DISTRIBUTED DATABASE SYSTEMS: FAILURE RECOVERY PROCEDURE

Sang H. Son
Satish K. Tripathi

Computer Science Report No. TR-88-06
March 23, 1988

Distributed Database Systems: Failure Recovery Procedure

Sang H. Son

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Satish K. Tripathi

Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

Distributed database systems operate in computer networking environments where component failures are inevitable during normal operation. Failures not only threaten normal operation of the system, but they may also destroy the consistency of the system by direct damage to the storage subsystem. To cope with these failures, distributed database systems must provide recovery mechanisms which maintain the system consistency. In this paper, techniques used for recovery management of distributed database systems are surveyed. We first describe the basic notions including transaction, consistency, and commit protocols. Then, we discuss the types of failures that may occur in distributed database systems and the appropriate recovery actions.

1. Introduction

Distributed database systems(DDBS) are based upon computer networks where component failures are inevitable during normal operation. Failures not only threaten normal operation of the system; they can also destroy the database correctness by damaging the storage subsystem. To cope with such damages, DDBS must have recovery mechanisms which can maintain the correct state of the system.

Recovery in a database system means restoring the database to a state that is known to be correct after some failure has left the current state incorrect. The underlying principles on which recovery is based can be described by a single word: *redundancy*. The database can be protected by ensuring that its correct state can be reconstructed from some other information stored redundantly elsewhere in the system.

There are various types of storage media which are distinguished by their relative speed and resilience to failure. The first type of storage media is the *volatile storage*. It is the fastest type of storage, but the information residing in volatile storage does not usually survive system crashes. Examples of such storage are main and cache memory. The second type of storage is the *nonvolatile storage*. Information residing on nonvolatile storage usually survive system crashes. Examples of such storage are disk and magnetic tapes. Disk is usually used for online storage of the database while tapes are used for archival storage. Both are more reliable than main memory, yet subject to failure which may result in a loss of information. The third type of storage is called *stable storage*. Information on this type of storage is never lost in the practical sense. To implement an approximation of such storage, we need to replicate information in several nonvolatile storage media with independent failure modes, and update the information in a controlled manner.

To recover from different kinds of failures, database management system periodically makes copies of each recoverable object and keeps these copies in a safe place. For example, consider a database application program updating a data file. The data file is read from the disk to main memory, and then the write operation is executed. However, the updated information is not safe until the file is transferred back

to the disk, since if the system crashes after the update operation was executed but before the transfer operation is completed, the new value of the file is never written to disk and thus, is lost. Therefore, the main purpose of the recovery procedure is to allow an in-progress transaction to be undone by deleting all "unreliable" changes in the event of minor errors without affecting other transactions, and in the event of serious errors, to restore the database by loading it from the most recent archive copy and then to minimize the amount of lost work by redoing all the changes made since that archive copy was taken. This purpose is accomplished by the recovery procedure by periodically recording critical portion of the system state in a stable storage, and by continuously maintaining a log of changes to the state as they occur.

The recovery manager coordinates the process of system restart. In performing system restart, it chooses among three types of system startup.

(1) Warm start: the process of starting up the system after a controlled system shutdown. It does not involve any *undo* or *redo* operations at all. Recovery needs only to locate the most recent checkpoint record in the log. The address of that record was placed in the "restart file" when the checkpoint was taken. To reduce the risk of losing the restart file, it is common to store it in a stable storage.

(2) Emergency restart: the process that is invoked after a system has failed in an uncontrolled manner. It involves undo and redo procedures because some transactions were in progress at the time of failure and must be redone or undone to obtain the most recent correct state.

(3) Cold start: the process of starting the system after some disastrous failure that makes a warm start impossible. It involves starting again from some archive version of the database. Work done since that version was created is lost and have to be redone.

For general surveys on database recovery management, readers are referred to [VER78, KOH81, DAT83, KOR86, BER87]. Transaction management and its relation to the recovery management is well described in [GRA79]. Different operation policies and their implementation techniques are described in [STO79, EAG83].

2. Basic Concepts

In database systems, users access data under the assumption that the data satisfies certain consistency assertions. Most assertions are never explicitly stated in designing or using a system, and yet all users depend on the correctness of these assertions. A complete set of assertions about a system is called the *consistency constraints*. The database state is said to be *consistent* if the contents of the data objects satisfy all the consistency constraints.

In DDBS the notion of database consistency has two aspects: (1) *internal consistency* of each data object, which requires that the database remains consistent within itself, and (2) *mutual consistency* of the replicated copies, which requires that all replicated copies must have same data values. The main goal of recovery management is to maintain both internal and mutual consistency of the database.

The consistency of the system state might be temporarily violated while modifying it. For example, in transferring money from one bank account to another, there will be an instant during which one account has been debited and the other not yet credited. This violates a constraint that the number of dollars in the system is constant. For this reason, the actions of a process are grouped into sequences called *transactions* which are units of consistency.

A transaction is the basic unit of user activity in database systems. Each transaction represents a complete and correct computation, i.e., if a transaction is executed alone on an initially consistent database, it would terminate in a finite time and produce correct results, leaving the database consistent. A transaction consists of different types of operations such as read, write, and local computations. Read and write operations are used to access data objects, and local computations are used to determine the value of the data object for a write operation. A transaction begins with a special BEGIN TRANSACTION operation and ends with either a COMMIT operation or an ABORT operation. COMMIT is used to signal successful termination; ABORT is used to signal unsuccessful termination.

We distinguish local and global transactions by the distribution of data objects that the transactions need to access. A transaction is *local* whenever the data objects it needs are stored at one site and the tran-

saction can completely run on that site. A *global* transaction is subdivided into a set of subtransactions T_i. Each subtransaction T_i is a local process to one site and runs under the control of a local transaction manager TM_i. The sites that have T_i of a transaction T are called the *participants* of the transaction. The coordinator C of a transaction T is the TM at the top of the control hierarchy which starts and finishes T by coordinating all cohort TM's participating in T. The transaction coordinator is not needed for local transactions.

It is important to note that, from the point of view of the user, transactions are *atomic*. In the *transfer* transaction, the user is not interested in the fact that two distinct updates must be made to the database; to the user, "transfer x dollars from account A to account B" is a single, atomic operation, which either succeeds or fails. If it succeeds, it is good. If it fails, then nothing should have changed. An output of a transaction is said to be *committed* when the output values are finalized and the new values are made available to all other transactions.

3. Commit Protocols

In DDBS, it is not so easy to achieve the atomicity of transactions because of the unanimity requirement. In order to ensure the atomicity property, all the sites participating in the execution of a transaction T must agree unanimously on the final outcome of the execution. T either commits at all sites, or aborts at all sites. It is the objective of a commit protocol to preserve transaction atomicity by collecting the status information of a transaction and making a unanimous decision based on the collected information. There are a number of different commit protocols that can be used. We present the most widely used commit protocol, called *two-phase commit protocol*, which is illustrated in Fig. 1. Other well-known alternative to two-phase commit protocol is the *three-phase commit protocol* [SKE81].

Let T be a transaction initiated by the coordinator C. In the first phase the coordinator sends "start transaction" messages to all the participants. Each participant individually determines whether it is willing to commit its portion of T or not, according to the result of the subtransaction it executed. If the decision is no, it adds a record <abort T> to the log and then responds by sending an "abort" (NO) message to

---

Protocol for coordinator

phase 1:

send "start transaction" messages to all participants
wait for votes from each participant

phase 2:

if all the votes are YES
then send "commit" to all participants and commit transaction
else send "abort" to all participants and abort transaction

Protocol for participants

phase 1:

"start transaction" is received
execute subtransaction and send vote to the coordinator:
YES to commit
NO to abort
wait for the final decision from the coordinator

phase 2:

receive either "commit" or "abort" and process it

Figure 1. The two-phase commit protocol.

C. If the decision is yes, it adds a record <ready T> to the log and forces all the log records corresponding to T onto stable storage. Once this is accomplished, it replies with "ready" (YES) message to C.

In the second phase the coordinator collects all the votes and makes a decision. Transaction T can be committed if C receives a YES message from all the participants. Otherwise, T must be aborted. Depending on the verdict, either a record <commit T> or <abort T> is added to the log and forced onto stable storage. At this point, the fate of the transaction has been sealed. Following this, the coordinator

sends either a "commit" or an "abort" message to all participants. When a site receives that message, it records it in the log and sends an "acknowledge" message to C. When C receives the "acknowledge" message from all the participants, it adds the record <complete T> to the log.

Since any site at which T executed could fail at any time, the protocol is designed in such a way that T will be aborted in all cases prior to the time that all sites involved in the execution of T have recorded changes made by T in stable storage. The "ready" message is, in fact, a promise by a participant to follow the coordinator's order to commit T or abort T. The only means by which a participant can make such a promise is if the needed information is stored in stable storage. Otherwise, if the site crashes after sending "ready" message, it may be unable to make good on its promise.

We now describe how two-phase commit protocol responds to various types of failures.

(1) Failure of a participant.

If the coordinator finds another participant to substitute the failed participant, execution of T will be continued. Otherwise, T will be aborted. When a participant recovers from a failure, it must examine its log to determine the fate of those transactions that were in the midst of execution when the failure occurred. Depending on the type of records contained in the log, either *undo* or *redo* operation will be executed. This is discussed in detail in Sect. 5.

(2) Failure of the coordinator.

When the coordinator fails, a decision must be made as to whether to commit or abort the transaction by all the participants. Transaction T can be committed only if there exists an active site that contains a <commit T> record in its log. Transaction T can be aborted only if there exists an active site that contains a <abort T> record in its log. If neither of these two cases can be established, then the fate of T cannot be determined and this must be postponed until the coordinator recovers.

(3) Failure of a link.

When a link fails, all the messages that are in the process of being routed through the link do not

arrive at their destination intact. From the viewpoint of the sites connected throughout that link, it appears that the other sites have failed. Thus, previous methods apply here as well. If a link failure results in a network partition in which the coordinator and its participants belong to more than one partitions, it can be considered as a combined case of coordinator/participant failure, and should be responded as described above.

4. Types of Failure and Recovery Actions

The major categories of failure that DDBS may suffer and the associated recovery actions to handle those errors are summarized below. The details of each recovery procedure and the techniques used are discussed in the following sections.

The first type of failure is the *transaction failure*. Transaction failures may result from entering erroneous data values or erroneous processing. The adequate recovery action is to issue a warning message and reject the transaction causing the failure. Rejection of a transaction requires undoing all changes made to the database by the transaction. This process is called *rollback*.

An appropriate way to correct an error caused by a committed transaction would be to take the database to the state prior to the execution of the transaction, and redo all other subsequent committed transactions. This procedure is called *rollback-and-redo*. To correct the committed transaction errors, it is necessary to perform rollback-and-redo of all the committed transactions with the corrected transaction, or *database reconstruction* which is a process of loading an archival copy of the database and rebuilding the database to the most recent consistent state.

The second type of failure is the *system failure*. The system failures can be caused by a bug in the database management system code, an operating system fault, or a hardware failure (CPU, communication links, etc). In each of these cases, processing is terminated in an uncontrolled manner, and we assume that the contents of main storage are lost, but the database is not damaged. Transactions that were in progress at the time of the failure must be rolled back, since they did not complete. The database system *restart* will be executed by rolling back the database and rerunning the transactions that were active at the

failure point.

The third type of failure is the *media failure,* in which some portion of the secondary storage medium is damaged. There are several causes for such a failure, the most common of which are (1) head crash, (2) hardware errors in the disk controller, and (3) bugs in the routines for writing the disk. Although failures in which the contents of nonvolatile storage is lost are rare, the system nevertheless needs to be prepared to deal with this type of failure. The recovery process, in this case, consists essentially of restoring the database from an archive copy of the database and then using the log to redo all the transactions run since that dump was taken.

5. Transaction Failures

The rollback procedure is coordinated by the recovery manager. In rollback, any output messages the transaction has produced must be cancelled to make it appear as if the transaction had never started. During the execution of a transaction, any write operation is preceded by the writing of a new record to the log. Even though the updates are applied directly to the database, the information in the log can be used in restoring the state of the system to a previous consistent state when a failure occurs, since all the changes to the system state is kept in the log. This technique is called *incremental log with immediate updates*. Undoing changes involves working backward through the log, tracing through all log records for the transaction in question until the <begin T> record is reached. For each log record encountered, the change represented by that log record is undone by replacing the new value in the database by the old value from the log. Since the information in the log is used in reconstructing the state of the database, the system cannot allow the actual update to the database to take place before the corresponding log record is written to stable storage. Therefore the system requires that prior to executing a write operation, the log record corresponding to it be written onto the stable storage. This is called the *write-ahead log protocol.*

The rollback procedure can be simplified if updates of transactions can be done temporarily until it commits. During the execution of a transaction, all the write operations are deferred until the transaction partially commits. All updates are recorded on the log. When a transaction partially commits, a record

<commit T> is written to the log, and the information on the log associated with the transaction is used in executing the deferred writes. Since a failure may occur while this updating is taking place, the system must ensure that prior to the start of these updates, all the log records are written out to stable storage. Once this has been accomplished, the actual updating can take place. This technique is called *incremental log with deferred updates*. If the transaction is in progress when the failure is detected, or, if the system crashes before the transaction completes its execution, then the database is not affected by the transaction, and simple abort is enough to get rid of those temporary updates.

To execute the rollback procedure, there must be a process for actually undoing a change. Update procedures should include, not only a "DO" entry point, which is used to perform the appropriate action in the first place, but also an "UNDO" entry point, which is used to undo the effects of any such action at a later time. When the recovery manager encounters a particular log record in its backward trace, it simply invokes the UNDO entry point of the update procedure, passing it appropriate parameters.

It is convenient if the log can be kept on a direct access device because the recovery manager needs to be able to access log records in a selective fashion, instead of purely sequential fashion. However it is infeasible to keep the entire log permanently on-line. A feasible scheme is to write the active log as a direct access data set. When the data set is full, the log manager switches to another such data set and dumps the first to archive storage, usually tape. System R employs a variation of this approach, in which just one on-line log data set is used in a wrap-around fashion.

The process of rolling back a transaction is subject to failure. Such a failure will subsequently cause the rollback procedure itself to be restarted from the beginning. Therefore the recovery manager must be prepared to handle the situation where it is trying to undo an update that has already been undone in a previous incomplete rollback. In other words, the UNDO logic must be *idempotent* so that undoing a given change any number of times is the same as undoing it exactly once.

6. System Failures

When a system failure occurs, the recovery manager must restart the system. To do that, the recovery manager should be able to determine those transactions that need to be redone and those that need to be undone. In principle, it can be done by searching the entire log from the very beginning and identifying those transactions. There are two major difficulties with this approach:

(1) The searching process is very time-consuming.

(2) Most of the transactions that need to be redone have, in fact, already written their updates into the database and do not really need to be redone. Redoing them will cause no harm but will cause recovery to take longer time.

To reduce these types of overhead, The notion of a checkpoint is introduced. During transaction execution, the system not only maintains the log, but also periodically takes checkpoints. Generating a checkpoint is simple in principle. At certain intervals, the system collects current information in a stable storage. Taking a checkpoint consists of the following steps:

(1) Write a <begin checkpoint> record to the log data set.

(2) Output all log records currently residing in main memory onto stable storage, all modified data blocks to the disk, and the identifiers of all transactions active at the checkpoint.

(3) Write an <end checkpoint> record to the log data set.

After the checkpoint log records have been written, the recovery manager records the address of the most recent checkpoint in the restart file. This allows restart to quickly locate the checkpoint record. During restart, the <begin checkpoint> and <end checkpoint> records are a clear indication as to whether a checkpoint was generated completely or interrupted by a system crash. The recovery manager gets the most recent completed checkpoint record from the restart file, and uses it to determine both the transactions that need to be undone and the transactions that need to be redone in order to restore the database to a consistent state.

To explain the emergency restart logic, transactions are classified into five distinct categories with respect to the most recent checkpoint and the crash point as shown in Fig. 2. We assume that a system failure has occurred at time t_f, and the most recent checkpoint prior to time t_f was taken at time t_c.

- Transactions of type T_1 were completed before time t_c.
- Transactions of type T_2 started prior to time t_c and completed after time t_c and before time t_f.
- Transactions of type T_3 also started prior to time t_c but did not complete by time t_f.
- Transactions of type T_4 started after time t_c and completed before time t_f
- Finally, transactions of type T_5 also started after time t_c but did not complete by time t_f.

It is clear that, at restart, transactions of type T_3 and T_5 must be undone. It may not be obvious that transactions of type T_2 and T_4 must be redone. The reason is that although the transactions did complete before the failure, there is no guarantee that their updates were actually written to the database.

The recovery manager starts with two lists, an undo-list and a redo-list, and determines which transactions must be undone and which redone. The undo-list initially contains all the transactions listed in

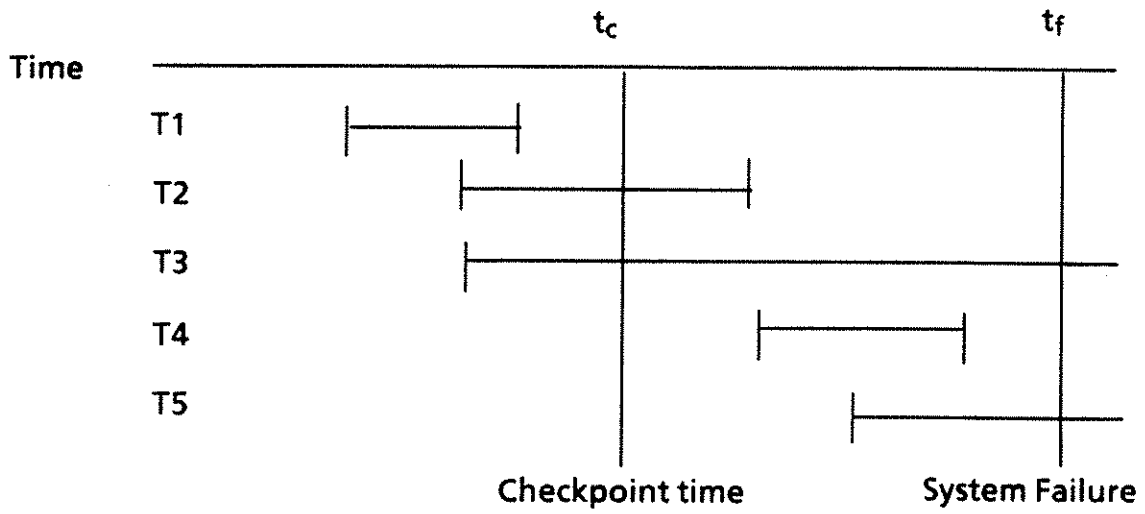

Figure 2. The five transaction categories

the checkpoint record; the redo-list is initially empty. The recovery manager examines the log to determine the transactions that started executing before the last checkpoint took place. Such transactions can be found by searching the log backwards to find the first <begin checkpoint> record that is associated with the last <end checkpoint> record, and then searching forward through the log, starting from the <begin checkpoint> record. If it finds a <begin T> record for a given transaction, it adds the transaction to the undo-list. If it finds a <commit T> record for a given transaction, it moves the transaction from the undo-list to the redo-list. When it reaches the end of the log, the undo-list and the redo-list identify those transactions that must be undone and those that must be redone. Then the recovery manager works backward through the log, undoing the transactions in the undo-list, and goes forward, redoing the transactions in the redo-list. No new work can be accepted by the system until this process is complete.

Update procedures may include a REDO entry point just as they include an UNDO entry point. The recovery manager is able to redo a transaction by tracing forward through the log records for that transaction and invoking the REDO entry point of the update procedure. REDO logic is required to be idempotent for the same reasons as UNDO.

An alternative to log-based crash recovery technique is *shadow paging*. Under certain circumstances, shadow paging may require fewer disk accesses than the log-based methods. The database is partitioned into some number of fixed-length blocks which are referred to as *pages*. These pages need not be stored in any particular order on disk. However, there must be a way to find the ith page of the database for any given i. This is accomplished using a *page table* that has one entry for each database page. Each entry contains a pointer to a page on disk. The logical order of database pages need not correspond to the physical order in which the pages are placed on disk.

The key idea behind the shadow-paging technique is to maintain two page tables during the life of a transaction: the *current* page table and the *shadow* page table. When the transaction starts, both page tables are identical. The shadow page table is never changed during the duration of the transaction. The current page table may be changed when a transaction performs a write operation. All read and write

operations use the current page table to locate database pages on disk.

Intuitively, the shadow page approach to recovery is to restore the shadow page table in nonvolatile storage so that the state of the database prior to the execution of the transaction may be recovered in the event of crash, or transaction abort. When the transaction commits, the current page table is written to nonvolatile storage. The current page table then becomes the new shadow page table and the next transaction is allowed to begin execution. It is important that the shadow page table be stored in nonvolatile storage, since it provides the only means of locating database pages. The current page table may be kept in volatile storage. If the current page table is lost in the system crash, the system recovers it using the shadow page table. Because any write operation changes only the current page table, it is guaranteed that the shadow page table points to the database pages corresponding to the state of the database prior to any transaction that was active at the time of the crash. Thus, aborts are automatic, and unlike the log-based techniques, no undo operations needs to be invoked.

To commit a transaction, the system must do the following:

(1) Ensure that all database pages in main memory that have been changed by the transaction are output to disk.

(2) Output the current page table to disk. Note that it must not overwrite the shadow page table since the system may need it for recovery from a crash.

(3) Output the disk address of the current page table to the fixed location in stable storage containing the address of the shadow page table. This overwrites the address of the old shadow page table. Therefore the current page table has become the shadow page table and the transaction is committed.

If the crash occurs prior to the completion of the step 3, the system revert to the state prior to the transaction. If the crash occurs after the completion of step 3, the effects of the transaction will be preserved.

Shadow paging offers several advantages over log-based techniques. The overhead of log-record output is eliminated, and recovery from crashes is significantly faster since no undo or redo operations are needed. However, there are drawbacks to the shadow page technique:

- Data fragmentation.

  For performance reasons, related database pages need to be kept close physically on the disk. This locality allows for faster data transfer. Shadow paging causes database pages to change location when they are updated. As a result, either we lose the locality property of the pages, or we must resort to more complex, high-overhead schemes for physical storage management.

- Garbage collection.

  Periodically, it is necessary to find all the garbages created by shadow paging, and add them to the list of free pages. This process, called *garbage collection,* imposes additional overhead and complexity on the system.

In addition to these drawbacks, shadow paging is more difficult than logging to adapt to systems that allow several transactions to execute concurrently. In such systems, some logging is typically required even if shadow paging is used. System R, for example, uses a combination of shadow paging and a logging scheme [GRA79].

7. Media Failures

A media failure is a failure in which some portion of the database storage medium is damaged. For this kind of failures, we need an archive dump of the entire database saved periodically in some stable storage. To restore the database from a media failure, the most recent archive dump is used. When the operator is informed of the failure, he allocates a new device to replace the one that failed, and loads the database on to the new device from the most recent archive dump, and then uses the log to redo all transactions that completed since that dump was taken. Let us examine the operation of taking an archive dump in more detail.

An archive dump must represent a consistent state of the system. Hence, it must be started when the system is in a consistent state, and that consistent state must remain unchanged until the dumping process is completed successfully. Archive dumping must be performed at intervals to minimize the costs of performing dumps and the costs of recovering the database. The cost of recovery depends upon the time spent by the system in recovering after an error is detected. The time and cost of recovery increase with the number of changes performed on the database between the last dump and the time the error was detected. Thus, if the dump intervals are very small, too much time and too many resources are spent in dumping; if these intervals are too large, too much time is spent in recovery. To reduce the time required to dump, some systems support the dumping of just those pages that have been updated since the last dump (incremental dumping). Such facilities are essential if the database is very large.

In DDBS, archive dumping mechanisms should include a coordination scheme to generate a globally consistent dump. Therefore dumping can be classified into three categories according to the coordination necessary among the autonomous sites. These are (1) fully synchronized, (2) loosely synchronized, and (3) nonsynchronized. Fully synchronized dumping is done only when there is no active transaction in the database system. In this scheme, before writing a local dump, all sites must have reached a state of inactivity. In a loosely synchronized system, each site is not compelled to write its local dump in the same global interval of time. Instead, each site can choose the point of time to stop processing and take the dump. A distinguished site locally manages a dump sequence number and broadcasts it for the creation of a dump. Each site takes local dump as soon as possible, and then resumes normal transaction processing. It is then the responsibility of the local transaction managers to guarantee that all global transactions run in the local dump intervals bounded by dumps with the same sequence numbers. In nonsynchronized dumping, global coordination with respect to the recording of the archive dump does not take place at all. Each site is independent from all others with respect to the frequency of the archive dump and the time instants, when local archive dumps are recorded. A logically consistent state of the system is not constructed until a global database reconstruction is required.

One of the problems of archive dumping methods is that the processing of transactions must be stopped for dumping. Maintaining transaction inactivity for the duration of the dump operation is undesirable, or may not be feasible, depending on the availability constraints imposed by the database system.

Archive dumping can be performed exclusively as part of the commitment of transactions and subtransactions. This scheme has the advantage of not having a separate archive dumping mechanism, but may have problems if the number of transactions allowed is too large or if it is necessary to keep archive dumps for a long time.

A backup database can be created by pretending that the backup database is a new site being added to the system [ATT84]. An initialization algorithm is executed to bring the new site up-to-date. The major drawback of this method is that the backup generation does interfere with update transactions.

A different approach based on a formal model of asynchronous parallel processes and an abstract distributed transaction system is proposed by Fischer et al. [FIS82]. Instead of viewing the actual state, the global dumps view a state that could result by completing all of the transactions that are in progress when the global dump begins. This approach is called *non-intrusive dump* in the sense that no operations of the underlying system need be halted while the global dump is being executed.

An archive dumping mechanism which provides a practical procedure for an efficient implementation of the non-intrusive dump approach is proposed in [SON85]. In this mechanism, each site saves the state of the data objects to generate a local dump. To construct a globally consistent dump, the updates of a transaction must be either included in the checkpoint completely, or not included at all. This can be achieved by dividing transactions into two groups according to their relationships to the current dump: *included-transactions* (INT) and *excluded-transactions* (EXT). The updates belonging to INT are included in the current dump while those belonging to EXT are not included. Timestamps are used to determine the membership of the set of INT and EXT. When an EXT commits, updated data objects are not stored in the database as usual, but are maintained as *committed temporary versions*. An EXT can be reclassified as an INT if it turns out that the transaction must be executed before the current dump. This is

called the *conversion* of transactions. A transaction-consistent state of the database is always maintained on stable storage. At a dump time, only the portion of the saved state which has been changed since the last dump is replaced. When a dump completes, next one begins with only a negligible interruption of service to delete old versions that have been maintained.

Two main properties of this archive dumping method are global consistency and reduced interference, both of which are crucial for achieving high availability of DDBS. Global consistency of each dump improves the availability of the system by enabling a fast recovery from media failures. However, system availability must not be impaired during normal operation. The analysis results show that this archive dumping method successfully satisfies these two conflicting requirements [SON86].

8. Recovery Strategies

In a centralized database system, basic techniques of recovery are undoing all "unreliable" changes of transactions, and restoring the database by loading it from the most recent archive copy and then redoing all changes made since that archive copy was taken. This kind of work must be done also in DDBS. However, recovering the database is more complex in DDBS for reasons including the following:

(1) In DDBS, it is necessary to consider *partial operability*. Partial operability is the property of DDBS that allows a group of sites that are operational and can communicate with each other to process the incoming transactions local to that group.

(2) Coordination is necessary to generate correct archive copy as well as to reconstruct the correct state by recovery managers.

The primary concern of recovery from system failures is to maintain the consistency of the database (recall that the system failures can leave the database in an inconsistent state). There are two strategies in recovery from system failures: *backward error recovery* and *forward error recovery*.

In backward error recovery, processing of new transactions is stopped when an error is discovered, and the database state is rolled back to a previously saved state known to be consistent. In forward error

recovery, the error detection and recovery are concurrent with the execution of transactions. The basic difference between the two strategies is whether the partial operability is allowed or not. Backward error recovery does not achieve partial operability. Even though forward error recovery is more complex than backward error recovery, it would be better to provide forward error recovery capability in DDBS because it exploits the advantage of distributed systems.

In forward error recovery, all the information the crashed site needs for recovery from system failures, is the update information of transactions which were executed after the site crashed. One simple way of getting the information is to request all other sites to send the appropriate portion of the log when the crashed site restarts. To increase the availability of the necessary recovery information, a guaranteed delivery mechanism is used in the database system SDD-1 in the form of the RelNet [HAM80]. Guaranteed delivery ensures that a message sent to a failed site is received by that site upon its recovery, even though the sending site may be down at the time of recovery. If there is no guaranteed delivery mechanism like RelNet, each site must provide a mechanism to store the necessary recovery information for the crashed site and deliver it when the crashed site is being recovered. This recovery information may need to be replicated for robustness. These kinds of mechanisms have appeared in the literature with different names such as *recovery array* [CHO80], *missing update information file* [EAG83], and *update history* [SHA78].

The number of sites that must send information to the crashed site depends on the operation policy of the system. If the operation policy is such that a transaction can be executed provided at least one copy of each referenced data object is available, then the crashed site needs to check all the sites that have copies of replicated data objects. On the other hand, if the partial operation policy is such that there is one master copy for each data object, then the crashed site only needs to check the master copy site for each data object.

REFERENCES

ATT84 Attar, R., Bernstein, P. A. and Goodman, N., Site Initialization, Recovery, and Backup in a Distributed Database System, IEEE Trans. on Software Engineering, Vol. 10, No. 6, Nov. 1984, pp 645-650.

BER87 Bernstein, P., Goodman, N. and Hadzilacos, Concurrency Control and Recovery in Database Systems, Addison Wesley Publishing Company, Reading, MA, 1987.

CHO80 Chou, C. and Liu, M., A Concurrency Control Mechanism and Crash Recovery for a Distributed database System, Proc. International Symposium on Distributed Databases, North-Holland Publishing Company, INRIA, 1980, pp 201-214.

DAD80 Dadam, P. and Schlageter, G., Recovery in Distributed Databases Based on Non-synchronized Local Checkpoints, Proc. Information Processing 80, North-Holland Publishing Company, Amsterdam, 1980, pp 457-462.

DAT83 Date, C. J., An Introduction to Database Systems, vol.2, Addison Wesley Publishing Co., Massachusetts, 1983

EAG83 Eager, D. and Sevcik, K., Achieving Robustness in Distributed Database Operations, ACM Trans. on Database Systems, Vol. 8, No. 3, Sept. 1983, pp 354-381.

FIS82 Fischer, M. J., Griffeth, N. D., Lynch, N. A., Global States of a Distributed System, IEEE Trans. on Software Engineering, Vol. 8, No. 3, May 1982, pp 198-202.

GOO83 Goodman, N., Skeen, D., Chan, A., Dayal, U., Fox, S., and Ries, D., A Recovery Algorithm for a Distributed Database System, Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 1983, pp 8-15.

GRA79 Gray, J. N., Notes on Database Operating Systems, Operating Systems: An Advance Course, R. Bayer (eds), Springer-Verlag, N.Y., 1979, pp 393-481.

GRA81 Gray, J. et al., The Recovery Manager of the System R database Manager, Computing Surveys 13, June 1981, pp 223-242.

HAM80 Hammer, M. and Shipman, D., Reliability Mechanisms for SDD-1: A System for Distributed Databases, ACM Trans. on Database Systems, Vol. 5, No. 4, Dec. 1980, pp 431-466.

KOH81 Kohler, W. H., A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems, Computing Surveys, Vol. 13, No. 2, June 1981, pp 149-183.

KOR86 Korth, H. and Silverschatz, A., Database System Concepts, McGraw-Hill Book Co., New York, 1986.

KUS82 Kuss, H., On Totally Ordering Checkpoints in Distributed Databases, Proc. ACM SIGMOD, 1982, pp 293-302.

MON78 Montgomery, W., Robust Concurrency Control for a Distributed Information System, MIT TR-207, December 1978.

SCH80 Schlageter, G. and Dadam, P., Reconstruction of Consistent Global States in Distributed Databases, Proc. International Symposium on Distributed Databases, North-Holland Publishing Company, INRIA, 1980, pp 191-200.

SEV76 Severance, D and Lohman, G., Differential Files: Their Application to the Maintenance of Large Databases, ACM Trans. on Database Systems, Vol. 1, No. 3 Sept. 1976, pp 256-267.

SHA78 Shapiro, R. and Millstein, R., Failure Recovery in a Distributed Database System, Proc. Spring COMPCON, 1978, pp 66-70.

SKE81 Skeen, D., Nonblocking Commit Protocols, Proc. ACM SIGMOD Conference on Management of Data, 1981, pp 133-142.

SON85 Son, S. H. and Agrawala, A. K., A Non-Intrusive Checkpointing Scheme in Distributed Database Systems, 15th International Symposium on Fault-Tolerant Computing, Ann Arbor, Michigan, IEEE, June 1985, pp 99-104.

SON86 Son, S. H. and Agrawala, A. K., Practicality of Non-Interfering Checkpoints in Distributed Database Systems, Proceedings of IEEE Real-Time Systems Symposium, New Orleans, Louisiana, December 1986, pp 234-241.

STO79 Stonebraker, M., Concurrency Control and Consistency of Multiple Copies in Distributed INGRES, IEEE Trans. on Software Engineering, Vol. 5, No. 3, May 1979, pp188-194.

VER78 Verhofstad, J., Recovery Techniques for Database Systems, ACM Computing Surveys, Vol. 10, No. 2, June 1978, pp 167-195.