3-3-1986

Attributes of fault-tolerant distributed file systems

Frank Goetz

ROCHESTER INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND TECHNOLOGY

ATTRIBUTES OF FAULT-TOLERANT DISTRIBUTED FILE SYSTEMS

By
Frank M. Goetz

A thesis submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

James Heliotis, chairman

Chris Comte

Rayno Niemi

Title of Thesis: Attributes of Fault Tolerant Distributed File Systems

I Frank Goetz hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for comercial use or profit.

Date: March 3, 1986

Table of Contents

Key Words and Phrases

Distributed FIle systems, recovery, fault tolerance characteristics and mechanisms in distributed file systems.

Abstract

Fault tolerance in distributed file systems will be investigated by analyzing recovery techniques and concepts implemented within the following models of distributed systems: pool-processor model and user-server model. The research presented provides an overview of fault tolerance characteristics and mechanisms within current implementations and summarizes future directions for fault tolerant distributed file systems.

1. Introduction

File systems support basic information management functions which include system table management (eg. file directory maintenance), enforcing/implementing file access policy (ie. efficiency, accuracy, protection), resource allocation, and resource deallocation. Distributed File Systems manage files across multiple processors. In this context, system configuration may vary from loosely to tightly coupled architectures, which may support a homogeneous or non-homogeneous set of processors.

Distributed File Systems differ from distributed data management systems. Distributed data management systems deal with the *relation* and *semantics* of objects. In distributed file systems, semantics of file content is not important except, possibly, from a broad perspective (eg. file type of "executable" or "text" in a directory).

As the volume and need to share information grows within an organization or enterprise, the need for distributed file systems will increase. The centralized file system may have physical limits imposed due to physical address space. In a distributed file system, file storage requirements can be adjusted on an individual site basis and each site may independently be configured to increase storage requirements. Thus a distributed file system may be more flexible since system storage limit is related to the number of storage nodes which can be supported. A centralized file system must expand to accomodate a corresponding increase in information.

Although distributed file systems have been implemented on networks consisting solely of minicomputers/mainframes, the future trend will also include distributed file systems across networks of personal workstations.

System-wide file service support (eg. remote access of non-local files) is a key responsibility of distributed file systems. To provide *reliable* file service in a distributed environment, the file system must be capable of recovering from some level of faults.

1.1. Fault Tolerance

The ability to recover from crashes and system errors is a necessity of any file system. In a distributed file system the problem is more complex. In particular, systems that maintain multiple file copies must restore the "correct" version and resolve incomplete file updates during recovery.

As an example, let *T1* be the transaction on node A comprised of a read of file *F1* (*F1* is located on node C) followed by a write to file *F1*; let *T2* be the transaction on node B consisting of a read of file *F1*. If *T1* and *T2* are submitted to node *C* for processing simultaneously, the view seen at A and B will be dependent on the transaction processing order. Crashes as well as communication failures in the middle of transaction processing must be handled by the transaction processing algorithm. In the former case, the transaction processing algorithm must either delay the processing of one of the transactions or back one out; in the latter, the transaction must be backed out. The problem complexity increases when considering a crash during the concurrent processing of *multiple* transactions involving shared files. The capability of resolving these problems (resuming after 'undoing' undesirable side effects) to restore consistency in the file system implies a level of fault tolerance supported by recovery techniques.

Fault tolerance is defined as the ability of a system to operate correctly in the presence of failures. In this context, a failure is a user perceived occurrence of a loss of a service; an error, on the other hand, manifests itself as an occurrence of an undefined or unspecified state within a resource (eg. CRC error, checksum error) [Avizienis-Kelly]. Fault tolerance as it relates to distributed file systems includes failures such as disk head crashes and component errors as well as software failures due to deadlock. In this paper, fault tolerant attributes within distributed file systems encompasses those mechanisms which support the correct operation of transactions in distributed file systems.

1.2. Project Description

The goal of this project is to investigate fault tolerance in distributed file systems by analyzing recovery techniques and concepts implemented within two different models of distributed systems: the *pool-processor* model and *user-server* model. In addition to providing an overview of fault tolerance characteristics and mechanisms within current implementations, the research presents future directions for fault tolerant distributed file systems.

The next section provides background information and concepts pertinent to the subsequent analysis. Here, the distributed models of the two distributed file systems to be investigated will be described and the distinguishing characteristics of a 'distributed file system' will be explained. Next, an overview of failures which can occur in a distributed file system is presented. An introduction to recovery concepts for handling distributed file system failures is the last topic in the section. In section three, recovery criteria is established which is to be applied to the two distributed file systems. The section details these criteria which include: atomicity guarantees, exception handling, concurrency control, deadlock detection and recovery data management. In the last two sections, fault-

tolerant aspects of two distributed file systems are investigated using the criteria outlined in section three.

2. Key Concepts and Background

The distributed file systems to be investigated are implemented within the user-server and pool-processor models of distributed computation. Before investigating the salient fault-tolerant features in these systems it is necessary to present relevant concepts and background information. This section begins with a description of the distributed system models under study and then describes the characteristics of a distributed file system. Finally, after identifying failures that may occur in a distributed file system, the section concludes with an introduction to recovery concepts for providing distributed file system fault-tolerance.

2.1. Models of Distributed Systems: User-Server and Pool-Processor.

A distinguishing characteristic of the user-server model of distributed computation is the absence of a central control processor. In this model, multiple personal computers are connected via a network to perform common functions such as share data and send mail. Also connected to the network are dedicated/specialized processors which may perform such functions as filing, mail management and printing. Traditionally, these dedicated processors are referred to as *servers*. The user-server organization (shown below) is driven primarily by the cost of peripheral devices such as printers and mass storage. In this model, computing is performed locally.

The pool-processor model (illustrated below) extends the user-server model to provide more processing power to individual users. Whereas in the user-server model, each user has a personal computer, the assignment of computers to users is dynamic in the pool-processor model. Users are assigned processors as needed by the system; that is, dynamic scheduling is used to assign sub-tasks of an application from a pool of available processors. Similar to the user-server model, some processors may have dedicated functions (eg. one phase of a linkage editor) in the pool-processor model.

2.2. Distributed File Systems

The "information resource" within a computer system is usually referred to, collectively, as the *file system*. A *distributed file system* implies that the *implementation* of the file system employs a model of distributed computation. In general, distributed file systems can be characterized by the system-wide support of file services on remote as well as local files. Typical services provided by the file system include: creation and deletion of files, file renaming, file copy, file open, file close and file list capability. In addition, file systems provide for the symbolic naming

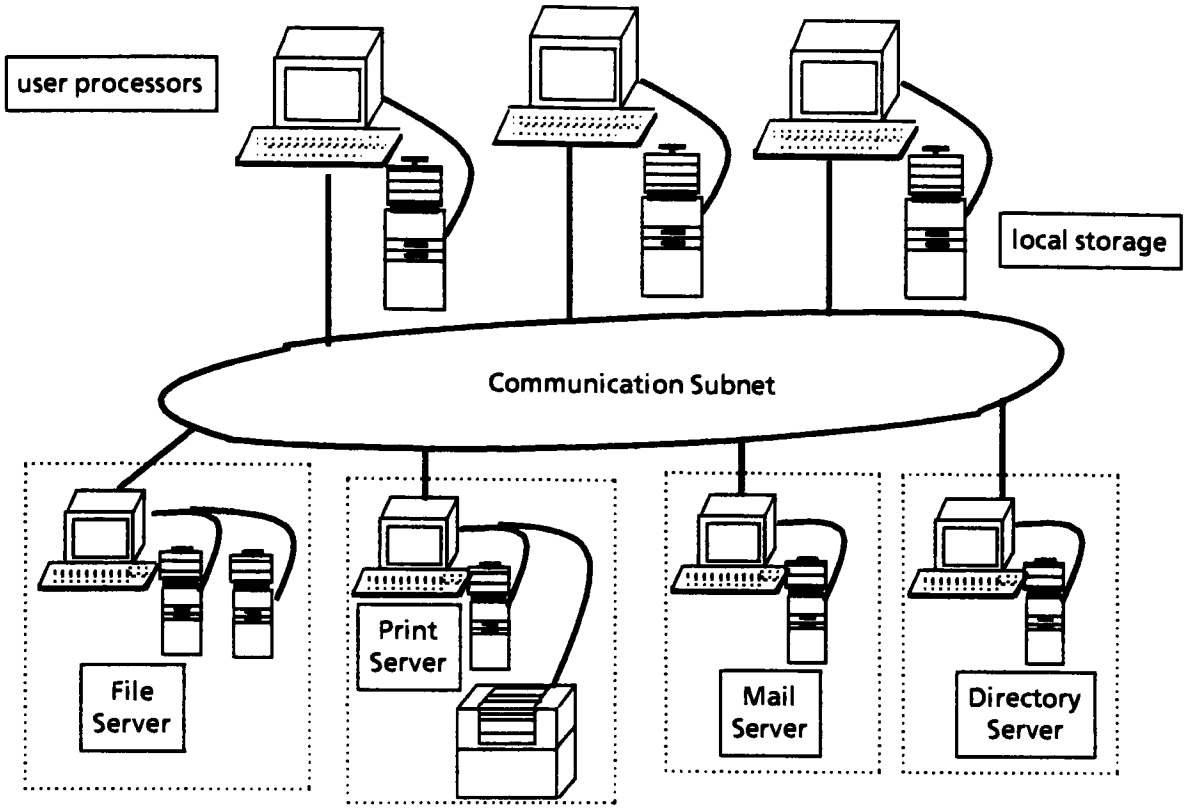

Fig. 2-1 User Server Model

of files (as opposed to the physical name) and support some level of access control for files. The goal in providing these services is to create a logical view of the file system in a friendly environment. Failures in a node or nodes of the system, if not resolved, may temporarily inhibit the correct operation of these services and ultimately produce errors in the file system. A summary of failures which may occur is discussed in the next section.

2.3. Failures in Distributed File Systems

To provide fault tolerance in distributed file systems, the level of fault resistance must be identified and the characteristics of system operation after failure must be defined. To this end, Garcia-Garcia-Molina [Garcia-Molina] has identified the following general classifications of failures: *node failures*, *communication line failures* and *malevolent failures*. *Node failure* is characterized by a "crash" in one or more nodes of the system. The recovery steps to be taken after a node failure usually depend upon whether or not processor status information was lost at

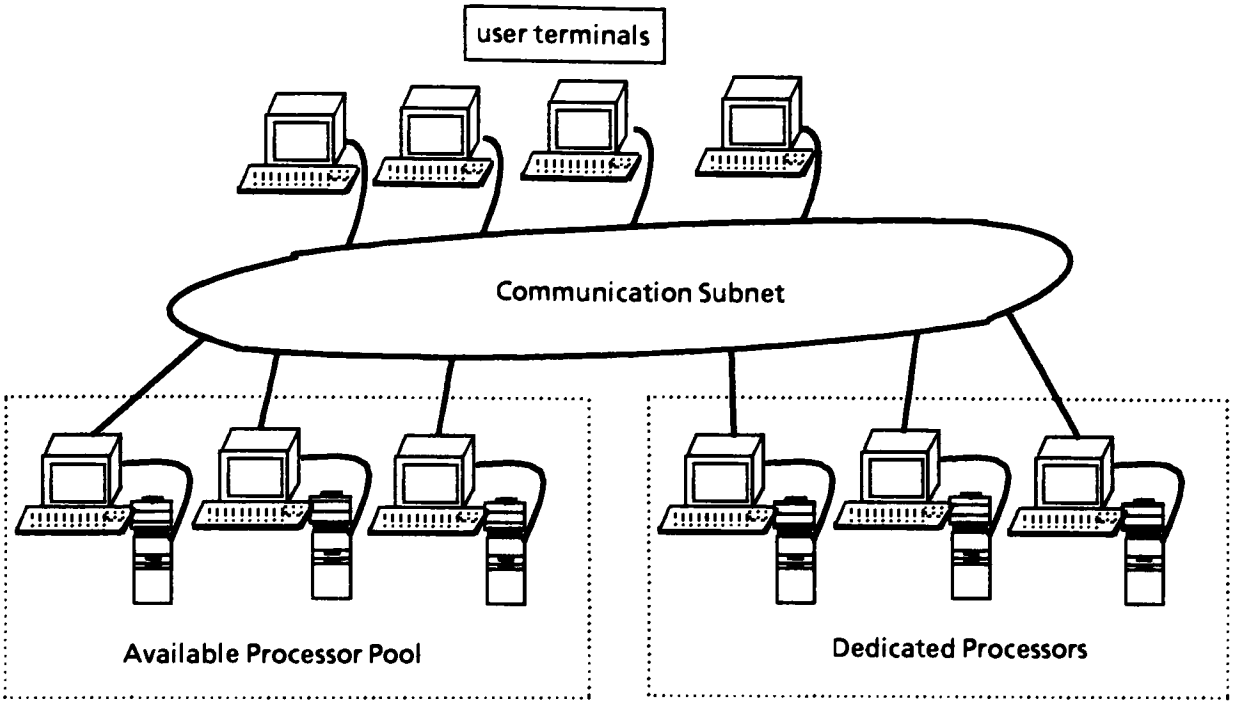

Fig. 2-2 Pool Processor Model

the node. *Communication line failure* between nodes may result in lost or damaged messages. Recovery in these cases depends upon the type of service provided by the communication network. *Malevolent failures* occur when the system acts in an unprescribed fashion. It is difficult to specify recovery algorithms for malevolent failures because they are very unpredictable and random. In addition, a system may exhibit *multiple failures* (more than one of the failures described) which possess an additional problem to the recovery algorithm designer in that a failure may occur while recovering from a failure. A failure which is detected by all nodes affected by the failure is called detectable; if a node continues normal processing in the midst of a failure which directly affects it, the failure is termed undetected.

Recovery protocols are implemented based on some assumptions on the characteristics of the failures. These assumptions are determined from knowledge of reliability of the underlying systems and components, cost of the overall system, as well as error occurrence probabilities. Consequently, the set of failures can be partitioned into two disjoint sets: those which are handled by the recovery protocol and those which are not. Lampson has identified the former set

of failures as *errors* and the latter set as *disasters* (ie. unrecoverable) [Spector-Schwarz]. Typically, malevolent failures and communication failures are not considered (usually low in occurrence and/or handled by subsystems). In terms of Garcia-Garcia-Molina's classifications *node failures* are usually considered 'errors' (hence recoverable) while *communication* and *malevolent failures* may be considered 'disasters' (unrecoverable). (This does not imply that all communication failures are unrecoverable. Failures unresolved by low-level communication recovery protocols may result in an unrecoverable failure at a higher level).

2.4. Recovery Concepts

According to Lampson, the goal of recovery is to hide the of effects of 'errors' on components; these new error free components provided by recovery he labels 'virtual components' [Lampson-Paul-Siegert]. After recoverable failures have been specified, strategies can be designed to support recovery based on the concepts of transactions and atomicity.

Transactions are used to group file requests together and, more importantly, to provide a mechanism to enable resolution of conflicting requests within the distributed file system. A transaction can be viewed as a logical group of distributed file system requests, involving files which may reside on one or more machines. Two characteristics of transactions which are important to recovery are *serializability* and *permanence* [Spector-Schwarz]. Transactions are serializable, in that transactions executing simultaneously have the same effect as if they would if executed one at a time. Permanence guarantees that updates will not be lost in a transaction which completes successfully. One other property usually required of a transaction is that results are only valid (eg. for use by other transactions) after its completion or cancellation. To assist in carrying out these guarantees, the distributed file system relies on atomic operations. *Atomicity* refers to the indivisibility of its action; atomic operations are uninterruptable operations which produce the correct requested result or no result. That is, a crash during the atomic write of a file B will result in either file B containing the update (the atomic write completed correctly before the crash) or, file B will remain unchanged.

Recovery protocols are based upon either forward or backward error recovery techniques. In backward error recovery, after detection of an error the 'consistent' state of the file system is restored through the use of checkpoints. In general, a checkpoint can be thought of as a specified point in processing at which a snapshot of the system state is taken (more on the information within this snapshot later) and saved in non-volatile storage. For example, in a transaction two checkpoints might be taken , say i, taken at t_0, and j, taken at t_1. Suppose an error is detected between t_0 and t_1 which the recovery protocol is capable of handling. The recovery protocol would simply restore the state of the transaction to the previous consistent state (from the non-volatile checkpoint information) at t_0 and resume from there. In systems employing forward

error recovery, no history information is used. Rather, corrective action is taken based upon implicit knowledge of the error type. For example, in a system employing redundant memory controllers, the forward recovery protocol on detection of an error may resolve the problem by simply designating the backup controller as the new primary controller (relegating the other to backup, presumably to be fixed).

2.5. Recovery as it Relates to Distributed File Systems

Fault tolerant features for local objects are well understood [Jegado] and have been incorporated into data management systems. Recovery in a distributed data management environment is usually limited to the transaction level. It is interesting to note that much of the recovery work that has been developed for distributed file systems has evolved from research performed for distributed data base systems. Providing an abstract (ie. beyond the transaction level) notion of recovery, as perceived by a user in a distributed file system, is more complex. In general, approaches to distributed file system recovery are implemented as part of the operating system or as an interface between the operating system and application layers.

3. Recovery Analysis Criteria

Recovery mechanisms within the distributed file systems under investigation will be presented and subsequently analyzed according to the topical criteria outlined below. The criteria is made up of aspects relevant to recovery in distributed file systems. This section will highlight the relevance of the five topical aspects to distributed file systems and will suggest strategies and mechanisms to handle each.

- atomicity guarantees
- exception handling
- concurrency control
- recovery data management
- deadlock detection

3.1. Atomicity Guarantees

For operations to be recoverable, they must be atomic. This aspect of the analysis criteria is concerned with the method(s) employed to guarantee this.

Atomic actions are an essential abstraction to fault-tolerant systems[Kohler]. The system guarantees that an action will be completed totally or not at all. In particular, a 'write' to a storage location will either produce the correct desired result or will not affect the location at all. Lampson and Sturgis have defined atomicity for both storage and communication types of operations [Lampson-Sturgis] in terms of a guaranteed set of states which will result from these operations. In this context, the action will either be: "completed and correct", "incorrect but detectable", or "not performed at all" In achieving these guarantees to promise recoverable atomic transactions, Lomet identifies the following implementation requirements [Kohler] paraphrased below:

- The atomic action must complete before its results are released.
- All updated objects in the atomic action can be returned to their previous values (before the atomic action commits).

The first requirement is related to the concept of commit point. The second requirement highlights the need for a 'transaction log' or a functional equivalent to undo the changes made prior to commitment. The commit point of a transaction can be thought of as the logical completion point of the transaction. A consequence of this is that the effects of a committed transaction can not be undone. In distributed systems, implementing these requirements have further implications, due to node failures and crashes (see [Garcia-Molina]). Because of this, the commit protocol design is an important element in any reliable distributed file system.

3.2. Exception Handling Mechanisms

Recovery can be broadly viewed as a process encompassing the following tasks: error detection, fault isolation, repair and system restoration. Exception handling in the context of this paper, refers to mechanisms which support the communication of errors (as well as general abnormal behavior) throughout the system to enable fault isolation and eventual repair. The capability of a system to provide for the processing of errors is determined by its exception handling mechanisms (if any). Thus, this aspect of the analysis criteria is concerned with schemes employed to provide exception handling.

Exception handling schemes in distributed systems can be based upon either centralized or decentralized control mechanisms. In centralized exception handling, a designated process detects errors throughout the system nodes and invokes recovery. In decentralized exception handling, errors are detected at the individual nodes.

3.3. Concurrency Control

This aspect of the analysis criteria is concerned with the mechanisms in place to handle simultaneous transactions. In distributed file systems, unrestricted concurrent processing of transactions can lead to an inconsistent file system. Suppose, for example, two transactions are submitted simultaneously (and allowed to run) for updating the same inventory file. In this example, the updates performed by the first transaction would be disregarded; the second transaction would always write the updated inventory file version. Hence, mechanisms are needed to protect against conflicts resulting from concurrent transactions within a distributed file system. The goal of concurrency control mechanisms is to maximize the amount of parallelism between transactions and minimize the number of conflicts between the concurrent transactions.

A record of transaction activity can be thought of as a sequence of reads and writes which operate on a subset of the file system. If each read of a file system data object is immediately followed by its write (update) then the transaction record (formally log) represents a serial sequence. A transaction log with "interleaved" reads and writes is only serializable if the reads and writes are to different file objects [Tanenbaum]. Tanenbaum summarizes a chief difficulty in devising concurrency control algorithms: "A good concurrency control algorithm should allow all serial logs, all serializable logs, and all other logs that leave the data base consistent. No such algorithm is known at present." [Tanenbaum]. Most algorithms [Tanenbaum] [Spector-Schwarz] employ locks as a means of achieving concurrency control. (This may be attributable to the rather simple implementation or perhaps due to use in concurrency control in operating systems). The general idea is to obtain a lock on individual file objects before accessing it (object, in this context refers to the unit of information handled by the file system: a byte, a block, a page, or the file itself, for example).

In supporting concurrent transactions, locks are often used as a concurrency control mechanism whereby a transaction may access an object as long as it possesses a lock for that object. The lock can either be granted or the transaction might be required to wait if there already exists a lock on the desired object. Two terms which are often used in discussing lock-based transaction commit protocols are well-formed and two-phase. Well-formed transactions are those which adhere to a policy of acquiring a lock for an object before accessing it; two-phase transactions do not release locks on objects before all needed locks are acquired. To provide consistent results among transactions, Eswaran, et. al. [Eswaran-Gray] proved that each transaction must be well-formed and two-phase. Although two-phase locking protocols are most common, Garcia-Molina [Garcia-Molina] notes that three-phase protocols have been required in some distributed

systems and suggests that n-phase protocols may also be required for more complex distributed systems.

A 'system-wide' lock is an example of a trivial concurrency control mechanism. In this case, at most one transaction can execute; this obviously offers the lowest degree of concurrency. One lock per file object is an improvement, but does not permit sharing of the distributed file objects in non-conflict cases (eg. three transactions which want to simply read the same file). This simple scheme can be extended to allow greater concurrency by relating a lock type to the intended access mode of the corresponding file object. Shared and exclusive lock types provide this capability. For example, multiple concurrent transactions can read a file object using a shared read lock; write locks are exclusive in that no other transaction may read or write the object while it has the lock.

In addition to the types of locks, the issue of the lock granularity must be addressed. A fine-resolution lock protocol implemented for attaining a greater degree of concurrency may be offset by slower transaction response time due to increased complexity of system monitoring functions. Some possible strategies, for example, include: file system lock; lock at file level; lock at byte level. An interesting scheme implemented in a Unix system employs locks at the driver level [Weinberger]. Here the goal was to support locks efficiently in the kernel. Weinberger relates the advantages with such a scheme: "the interface is well-defined and simple and since the implementation is at the system call level, requesting processes don't have to give up the CPU" [Weinberger]. A disadvantage of locks (including two-phase locks), however, is that deadlock could occur; since, in a distributed system, a transaction cannot acquire all locks needed in one instant, it may therefore be required to wait for the lock. If the distributed file system arrives at a state where, for example, each of two transactions is waiting for the lock held by the other transaction, deadlock may occur.

Kohler's basic requirements of concurrency control protocols [Kohler] for distributed data base systems can be stated in terms of distributed file systems:

- the protocol should maintain file system consistency.
- the protocol should require each transaction to operate in a finite amount of time.

The basic requirements while essential, do not adequately address reliability and performance aspects of concurrency control protocol design. Thus, Kohler developed an extended set of design goals which can similarly be adapted to distributed file systems [Kohler]:

- the protocol can tolerate node and communication failures.

- the protocol will support the required degree of concurrency.
- the protocol should be efficient in terms of processing time and in memory consumption.
- the protocol should not degrade appreciably as network traffic increases.
- the protocol should minimize restrictions placed on the organization of transactions.

3.4 Deadlock Detection

In a system where there is a potential of having to wait for resources, a deadlock situation can occur. This recovery analysis aspect represents the capability of a system to deal with deadlock.

Deadlock refers to process state(s) characterized by an infinite wait for an event. This, obviously has disastrous implications to a distributed file system. In a distributed file system, the infinite event could be an un-grantable open file request as part of a transaction.

If for example, transaction A contained the following:

```
lock(file1);
read(file1);
lock(file2);
read(file2);
unlock(file1);
unlock(file2);
```

and transaction B contains:

```
lock(file2);
read(file2);
lock(file1);
read(file1);
unlock(file1);
unlock(file2);
```

If transactions A and B are processed simultaneously deadlock could occur. Transaction A obtains a lock on file 1 and proceeds to read file1; at the same time, transaction B has obtained a lock on file 2 and reads file2. Next, A attempts to get a lock for file2 (this will not be granted since transaction B has not unlocked it yet); similarly, B is not granted a lock for file1 (since transaction A has not unlocked it yet). This is an example of deadlock because each transaction must wait for a resource (lock held by the other) that will never become available. From this simple example it is clear that some provisions for handling deadlock in a distributed file system are required. This aspect of the recovery analysis will focus on methods to deal with this.

In general, approaches to this problem can be classified as prevention, avoidance, or detection strategies. The goal of both prevention and avoidance strategies is to inhibit the possibility of deadlocks occurring. Prevention strategies impose tighter constraints on the system than avoidance strategies. Typically, prevention strategies remove the possibility of deadlock from occurring by imposing controls on resource (eg. locks) acquisition before a transaction can be run; avoidance strategies allow transactions to operate with the possibility of deadlock of occurring and avoid deadlock at transaction run time. For example, a prevention strategy could require that a transaction must acquire all locks needed before it can run. An example of an avoidance strategy might be one based upon timeouts in which a transaction is aborted if it has not completed before some timeout interval. It should be noted that avoidance strategies will not honor a resource request if it will result in a deadlock condition. Thus, prevention strategies offer high reliability at the expense of potentially decreased resource utilization; avoidance strategies incur increased runtime overhead but offer potentially better resource utilization. Lastly, in systems employing detection strategies, no prevention of the deadlock situation is automatically performed. Rather, the incidence of the deadlock situation is notified to a higher level system function to subsequently resolve. Havender has proposed three prevention strategies [Deitel]:

- resources requested in advance
- on denial of a resource request, the requestor must give up all resources held and re-request later.
- group resources into 'types' and impose an access order to the resource types (ie. must request resources in order).

Avoidance strategies can be implemented using timestamps [Spector-Schwarz]. Detection techniques typically detect the presence of a deadlock situation and defer deadlock handling to a resolver (eg. to remedy the situation by aborting a transaction). A timeout interval associated with a transaction can be used to detect a deadlock condition. However, a disadvantage with timeout mechanisms is that they may introduce a time variance dependency into transactions. Other distributed deadlock detection mechanisms which are more definite include: distributed deadlock detection based upon timestamps [Sinha-Natarajan] and distributed algorithms based upon the construction of wait-for dependency graphs [Ho-Ramamoorthy][Sugihara].

3.5. Recovery Data Management

The information necessary to recover from faults is dependent upon the recovery approach implemented. The focus here is on the logical organization of the recovery data as well as its use

within the overall recovery strategy. Thus, strategies and structures for handling recovery data and the unit of granularity of recovery will be studied. In backward recovery based systems, a history of transaction processing organized according to 'known' consistent points (ie. recovery points or "checkpoints") is required. On the other hand, forward recovery systems rely primarily on the status of the incomplete transaction at the point of the error and do not require the use of checkpoints. In general, backward error recovery resumes transaction processing from the previous consistent state information associated with the determined checkpoint; in forward systems, the effects of a failure are undone by performing inverse operations on the affected file. (eg. An error on a file delete request might be recovered through forward recovery by undeleting the file...assuming that the file was marked for delete but not yet expunged. Alternately, the forward recovery protocol might be to simply delete the file in such a case.).

Typically, stable or non-volatile storage is used to provide necessary reliability for information. Stable storage involves stringent protocols to guarantee that correct values will result from reads and writes (except when 'disasters' occur) [Lampson-Sturgis].

Recovery data management techniques in general can be classified as shadow-based or log-based methods. In shadow techniques, the "real" versions of objects are stored on stable storage; modifications within the transaction are made to a copy. The modified copy is transferred to stable storage as the "real" version when the transaction successfully completes. In this fashion, handling aborted or cancelled transactions is trivial because the "real" version is not updated with the copy in these instances. (Thus, the 'real' version always shadows the latest version). Alternately, a log can be used to provide a history mechanism.

In a system using a log for recovery data, when a transaction must recover, the log is processed by successively undoing the effects of incomplete transactions, backwards through the log to a 'safe' spot (a recovery checkpoint). The transaction processing sequence resumes from here. Note that when logging is used, actions within transactions may operate on the real object. Only the log information must be kept in stable storage. Transaction log information may be organized using two general techniques for recording log entries. Linear recording implies that old and new object values are kept. In transitional recording only the changes are kept. An inherent problem with recording transitions is in obtaining appropriate inverse functions such that applying the inverse function to the change yields the old unchanged value. Using transaction logs does not by itself guarantee reliable recovery data. A question which must be answered is: "What if a crash or failure occurs during a write to storage?". In using shadow copies, the procedure is to write at commit. This concept is extended and forms the basis of the write-ahead protocol which consists of the following two steps [Kohler]:

- the undo entry is transferred to stable storage and then the updates are written.
- after writing undo and redo entries, the transaction is committed.

In studying recovery problems as they relate to distributed databases, Kohler has developed requirements of recovery data which indicate a need for recovery data at the file system level as well as at the distributed system level. Snapshots of the system state taken at checkpoints are used by backward error recovery techniques to resume from a fault. Global checkpoints in a distributed system are expensive in terms of synchronization monitoring (hard to synchronize), space (data is globally recorded), and time (processing time to take checkpoints). Distributed checkpointing is more efficient as each node takes an individual checkpoint. Ensuring that meaningful and consistent checkpoints are taken can be accomplished by using two-phase commit protocols in obtaining checkpoint data at process (transaction) interactions [McDermid].

4. Recovery Within a User-Server Distributed System: Xerox DFS

4.1. System Configuration and Overview

The Xerox Distributed File System consists of multiple *server* and *client* personal computers networked together by the Ethernet. Each personal computer has local storage and may operate independent of the network. The servers are actually dedicated personal computers (ie. not available for casual use) which are accessed by the client computers to perform such functions as: directory services (directory servers), mail services (mail servers), file services (file servers) and print services (print servers). Servers can be thought of as the 'providers' of an abstract information resource while clients can be thought of as interface processors which handle user interface functions and provide file system applications.

Network communication within the Xerox Distributed File System (XDFS) is provided by Ethernet, a local area network. Ethernet is categorized as a carrier-sense multiple access (CSMA) protocol which uses a shared coaxial cable as its medium for data transfer.

XDFS was designed to exist external to operating system software and was implemented in the Mesa programming language.

4.2. File System

A chief design goal of the system was to "allow flexible allocation of computing and storage resources to the filing service"[Mitchell-Dion]. Identity-based access control is provided by a directory server which handles the logical to physical file name mapping. A client, after authenticating the user, sets up a connection with the appropriate server which lasts for the duration of the transaction. XDFS provides facilities for filename to server location mapping, it is a client responsibility to present a file server-independent view of the XDFS to users.

Support for multiple server transactions is, in part, a direct consequence of the system architecture. That is, a transaction will minimally involve a file server and a directory server. In such a transaction, the initial server is referred to as the primary server and the other participating servers are termed workers. The primary server acts as transaction manager over the participating worker servers in completing the transaction. Using this technique, the XDFS provides support for multiple non-local file updates within a transaction.

A transaction consists of a series of file requests bracketed by *BeginTransaction* and *EndTransaction* types of commands. An *AddServer* command is issued by a client to access files on different servers. When a client issues an *OpenTransaction* command to the primary server, it returns a transaction identifier which, along with file identifiers, is used on subsequent file requests. Updates within a transaction are performed not directly on the actual files but instead to new data areas. When a *EndTransaction* is accepted by the primary server, the transaction is committed, the updates are made permanent by the server (or servers in the case of a multiple server transaction) involved, and the server-client connection is closed.

4.3. Recovery Mechanisms

4.3.1 General Description of Recovery

Before a file server carries out the requests specified in a transaction submitted by a client, the server forms a list describing the series of file operations in the transaction called an *intentions list*. The server transfers the intentions list to a form of non-volatile memory termed *stable storage* [Lampson-Sturgis]. Stable storage is an abstract object provided by software which relies on hardware detection of bad memory page writes. In the XDFS implementation, a normal file page is represented as two stable file pages. At commitment (when *EndTransaction* is issued by the client), the server carries out the intention list actions on the intended file in stable storage, then erases the list. In the event of a crash before commitment, the server recovers by processing the actions specified in the intentions list.

A transaction involving multiple file servers requires additional synchronization to prevent an abort from triggering a series of aborted transactions. To this end, the first server contacted (ie. the primary server) becomes the synchronize manager (coordinator) for the transaction and coordinates the activities of the other servers involved in four synchronized phases. In the first phase, each server writes a 'tentative' intentions list to stable storage. In the next phase, the names of the participating workers are included in an intentions list written by the coordinator to logically complete the transaction. The third phase is initiated by the coordinator

broadcasting the successful completion of the second phase (this prompts the workers to adjust their respective intentions lists to indicate that they are confirmed). In the fourth and final phase, the workers carry out their respective lists, then erase them. In the event of a server crash before commitment, all intentions lists are obtained and those previously confirmed are carried out. The primary server must be contacted for each of the remaining unconfirmed lists, before they can be processed.

XDFS contains mechanisms to recover from crashes as well as from aborted transactions. In providing crash recovery, stable storage is used in conjunction with crash recovery software. This guarantees that a server crash which occurs after a transaction commits (when *EndTransaction* is issued) will not affect its results. An incomplete transaction at the time of a crash "has the same effect as an abort" [Sturgis-Mitchell-Israel]. Recovery from aborted transactions involves intentions lists and multiple server synchronization policies.

Although XDFS guarantees consistent data (via stable storage) in the presence of crashes, it places the responsibility of maintaining consistency amidst exceptions on clients. If a server crash occurs before a *CloseTransaction* is issued, the updates described by the transaction are not performed; if a crash occurs after the *CloseTransaction* is issued but before it completes, the updates described by the transaction will be carried out successfully by server recovery software. When an exception occurs during a server file request, the server returns the exception code to the client instead of performing the file request (ie. the file remains unchanged). Thus, servers need only be concerned about *providing* the file information or a reason as to why it can not be obtained. Clients, on the other hand, know the logical information structure, and are therefore responsible for recovering from exceptions.

4.4. Analysis of Recovery Mechanisms

4.4.1 Atomicity Guarantees

XDFS guarantees a transaction will perform as a logical atomic action. Either the series of file reads and file writes encapsulated within the transaction will complete correctly or not at all (ie. no partial results). This strong guarantee holds amidst concurrent file access (see 4.4.3) as well as during crashes. Additionally, XDFS file operations associated with the intention list are atomic.

4.4.1.1 Issues and Alternatives.

Although XDFS guarantees that a transaction will complete correctly during server crashes and concurrent file requests, the responsibility for handling exceptions is placed on clients. A client

recovers from exceptions by redoing the entire transaction. Recovering from exceptions can be made more efficient if the client takes advantage of its local data cache to save file data values. After a server crash, a client can reread the data involved in the transaction and compare it with the (previously) cached values and if still valid the former results can be used without recomputing.

4.4.2 Exception Handling

XDFS servers can notify clients of exceptions through an "unsolicited message" [Sturgis-Mitchell-Israel]. This mechanism allows a server to broadcast a message to clients to notify them of broken locks (see 4.4.3) without the need for an acknowledgment. An unacknowledged message is not a problem; a transaction with broken lock(s) can proceed to commitment only after it explicitly issues ClearLock requests on the broken lock(s). On an individual basis, a server sends a result message to a client containing either the requested data or an error code. When an error is issued, it is the responsibility of the client to resolve the situation. In general, to recover from an exception or crash, the client redoes the transaction.

4.4.2.1 Issues and Alternatives.

The protocol followed handles all failures identically. That is, servers detect and report all exceptions to clients; clients must detect exception signals from servers and perform operations to recover from the effects of exceptions. Additionally, if a server runs out of resources, the server informs the client that it had to abort.

An obvious advantage to this scheme is its simplicity and generality in handling exceptions; there are no fault-specific processing requirements within the file system. Also, note that the XDFS approach is natural for the interfaces defined: a server returns a result in response to a request; since the client is already expecting a result, an "exception result" is a normal consequence. It is then the responsibility of clients to process exceptions after detection.

In looking for other possible exception handling schemes for suitability to the XDFS, the selection is limited to decentralized methods. This is attributable to the division of responsibility in XDFS with respect to exception handling. Servers comprising XDFS guarantee integrity of files and atomicity in transaction operations amidst server crashes; servers do not guarantee the validity of transactions which depend on their files in such instances. In XDFS, clients must take appropriate action in response to a server or client crash in order to ensure the correct completion of transactions. For example, simply redoing a transaction by rewriting previous values may not be correct if previous values which were used

to generate the data changed. Although centralized exception handling methods succeed in minimizing exception handling redundancy, they are not directly applicable to XDFS for several reasons. A centralized exception handler (eg. an 'exception server') would be required to detect server crashes as well as client crashes swiftly and efficiently. However, because of XDFS support of multiple server transactions, a client must be aware of all server crashes with which it is involved. This implies that such a centralized exception handler must either have detailed knowledge of all client-server relationships, or adopt a simple policy of notifying all (available) clients and servers of crashes. The amount and frequency of change of detailed knowledge required by the centralized exception handler renders the retaining of client-server relationships method impractical. Globally broadcasting all crashes, the second alternative, is inefficient: for now, all servers and clients in the system will be notified even though a server crash may only involve one server and one client. Because of this, it may be difficult to realize response and reliability requirements due to the communication overhead incurred. Furthermore, such a scheme would add complexity to client software.

Decentralized exception handling methods, however, are more suitable to the XDFS. The distributed exception handling mechanism employed in the Sirius/Delta project [Lampson-Paul-Siegert] is an example of one such scheme which could be adapted to the XDFS. Briefly, this scheme relies on an enhanced 2-phase commit protocol to assist in handling exceptions. During the intentions phase, the clients write a list of the other servers involved in the transaction; if any client notices that any of the servers in the set are down, it may message the other involved servers to determine whether or not to abort the transaction. The protocol only requires one client to abort for the transaction to be aborted. Note the similarity to the four-phased synchronization which currently exists within the XDFS (see 4.4.3). The Sirius/Delta scheme guarantees that if one server participating in a transaction commits, all of the other involved servers that are up will commit; if one server aborts, all operating servers will abort. The advantage of this scheme is faster response for transactions that are determined early in processing to be ready to commit or abort. However, this also results in increased data transmissions between servers and requires additional system memory for the servers (for server transaction status information).

4.4.3 Concurrency Control

Conventional locks in distributed systems are exclusive writer, multiple reader and usually are implemented over the entire file. In XDFS, a different approach is used which allows multiple writers of a file in a controlled fashion. The granularity of locks in XDFS is a range of sequential bytes in a file. The protocol implemented allows multiple readers and writers to operate on a file in a controlled fashion by using four-state locks: *Unlocked*, *Read*, *IntentionWrite*, and *Commit* (see p. 24). The protocol allows a transaction to proceed with *IntentionWrite* locks; it is only required that the *IntentionWrite* locks become confirmed before

the transaction is allowed to commit. Without any other lock removal strategy, a situation could occur in which multiple transactions are waiting to commit (due to *IntentionWrite* locks) resulting in long transaction processing times (potentially deadlock). The XDFS solves this by associating a timeout with each lock and allowing a client to break locks. When the timeout associated with a lock expires, the lock is considered *soft* since it is a potential candidate to become broken. A competing transaction can break a soft lock if the data associated with the lock is needed by the transaction (locks are managed by the server holding the data). In general, locks are cancelled in response to an allowed cancel request from a client and also at the end of a transaction. A lock held by a server may become broken if it is cancelled by another client desiring data within the range of the lock. A client cancel lock request is only allowed, however, if: a timeout on the lock has occurred, the current client transaction associated with the lock has not yet updated the data, and the range of data of the competing transaction's lock request intersects with the data range associated with the current transaction's lock.

Another service provided by the XDFS is to inform clients of broken locks (additionally, a client may request the list of broken locks from a server). In providing a broken lock clearance strategy, XDFS allows clients the option of voluntarily giving up these locks (presumably, in the case where the transaction does not depend on the locks) to enable the transaction to complete. To support this feature, for a transaction to commit now, all locks must either be confirmed or those which have been broken must be *cleared.* A client can proceed to commitment with a broken lock only after issuing a *ClearLock* request to the server controlling the lock. Note the distinction between clearing and cancelling a lock: after cancelling a broken *IntentionWrite* lock, the lock would transfer to the *Unlocked* state; after clearing a broken *IntentionWrite* lock, the lock remains in the *IntentionWrite* state ready to proceed to the *Commit* state. Through these capabilities, the XDFS enables blocked clients to continue as well as clients with broken locks, the potential to resume.

The ability to break locks under certain restricted conditions enables clients to support local data caches. This capability, which is advantageous for performance and data availability reasons, is sustained through the use of *IntentionWrite* locks.

The motivation for *IntentionWrite* locks can be more clearly seen from the lock-grant matrix of a typical multiple reader/one writer system [Lampson-Paul-Siegert] in Fig 4-1 below:

This lock mechanism clearly favors read intensive transactions. As more updates (write operations) take place, the probability of encountering conflicts with an existing *ReadLock* or *WriteLock* increases. Lock-grant status characterizing this behavior is represented by the last

Current Lock State	Next Lock State		
	Unlocked	ReadLock	WriteLock
Unlocked	allowed	allowed	allowed
ReadLock	allowed	allowed	not allowed
WriteLock	allowed	not allowed	not allowed

Figure 4-1. Multi-Reader/One-Writer Lock Grant Matrix

two entries of the last column [Lampson-Paul-Siegert]. When this occurs transactions are blocked, thereby delaying others from commitment. XDFS attempts to minimize this delay by postponing file data updates until commitment using an additional *IntentionWrite* lock.

The lock grant matrix for XDFS locks is as follows in Fig.4-2:

Current Lock State	Next Lock State			
	Unlocked	ReadLock	IntentionWriteLock	Commit
Unlocked	allowed	allowed	allowed	allowed
ReadLock	allowed	allowed	allowed	not allowed
Intention WriteLock	allowed	allowed	not allowed	not allowed
Commit	allowed	not allowed	not allowed	not allowed

Figure 4-2. XDFS Lock Grant Matrix

Under this protocol, transactions tend to operate in the upper left corner of the matrix which is characterized as pre-commitment transaction activity; lower-right corner reflects transaction lock status at or near commitment [Lampson-Paul-Siegert]. A transaction is allowed to operate more freely prior to commitment without having to worry about existing write locks. Note that only single *IntentionWrite* locks of file data are allowed (ie. an intention write lock on a range of data will not be granted if one already exists on any data which intersects with the desired range). The design choice of not supporting multiple write intention locks was made on the basis of handling conflicts (immediately or delay). Multiple intention write locks merely delays having to resolve conflicts; with single intention write locks, conflicts are handled as they occur. In XDFS, conflicts will resolve as transactions change their intention write locks to commit locks.

4.4.3.1 Issues and Alternatives

The variable granularity lock-based concurrency control method which provides clients with the capability to break locks, is developed in order to support caches of file server data at client nodes. This also provides a higher degree of concurrency at the expense of a more involved four-phased synchronized protocol for performing updates.

Intuitively, a less complex concurrency control protocol could have been implemented based upon file size lock granularity. Any such scheme, however, would tend to eliminate the capability to store caches of data at the local nodes. For now, a server would have to notify participating clients of broken locks whenever a transaction accesses a file being used by another transaction; in the XDFS strategy, notification of broken locks only occurs if the desired data intersects with the range held by the lock. As a result of reduced availability of data at the clients, system response would decline because of the decreased potential for concurrency.

In general, centralized concurrency control methods are not directly applicable to distributed systems due to demands for high availability and/or performance [Lampson-Paul-Siegert]. Other concurrency control mechanisms based upon decentralized control could have been employed such as multiple physical or logical clocks, use of a circulating privilege or a circulating sequencer [Lampson-Paul-Siegert].

Clocks are used in concurrency control to provide synchronization throughout the system. Each node has a clock which runs in discrete steps and is synchronized with the others in the network. The nodes participating in the transaction, process the next sequential step of the concurrency control protocol at each tick of the clock. In multiple logical clocks, synchronize messages are passed to every node ("via a connected graph") at a predetermined rate; a synchronize message contains a timestamp which is compared to the local process clock and if necessary made to be larger than the timestamp value (after adjusting for communication overhead). Similar to multiple physical clocks, multiple logical clocks are implemented as local counters. The local counters are incremented on an action/operation basis throughout the transaction. Overall synchronization (among the logical clocks) is achieved by including the logical clock value in inter-node correspondence; nodes adjust their logical clocks to be greater than the transmitted clock value.

4.4.4 Deadlock Detection

At the commit point of a XDFS transaction, all soft intention write locks within the transaction must become commit locks for the transaction to continue. This is accomplished by examining the state of the read locks also held on the same data. Any soft read locks on the same data are broken and the commit lock is granted. However, in the case of a soft *IntentionWrite* lock with at most one read lock not soft, the transaction is aborted. This scheme guards against a potential deadlock situation referred to as deadly embrace [Sturgis-Mitchell-Israel].

For deadlock to occur, a directed cycle must exist in the file lock demand graph [Ho-Ramamoorthy]. In XDFS, deadlocks are prevented through the use of timeouts and providing clients the capability of breaking locks. In a minimal case of a detected directed cycle in XDFS, suppose a transaction *A* is waiting to obtain information denied by a lock held by a transaction *B* and before *B* can proceed, it needs information in a file area currently locked by *A*. In this example, deadlock can be avoided within XDFS after either lock times out. For example, if the lock held by *A* times out first and *A* has read but not yet updated the data, *B* can then break the needed lock held by *A* thereby allowing *B* to proceed. Consequently, when *B* commits, the data required by *A* is unlocked and A can proceed. Since this capability is embodied within the individual servers and clients, this is an example of decentralized deadlock detection [Ho-Ramamoorthy].

4.4.4.1 Issues and Alternatives

Centralized detection schemes are not attractive to the XDFS for reasons similar to those given for centralized concurrency control methods in 4.4.3. There is also a technical problem with many of the concurrency control schemes that are based upon gathering correct remote status information (eg. for use in demand graphs). In such schemes, data required by the detection protocol may not be valid after collection because: 1) status can not be transmitted to the central control node instantaneously (implying transmission delay) and 2) local status can change randomly with high frequency.

In [Ho-Ramamoorthy] three schemes are presented for decentralized deadlock detection: a two-phase protocol, a one-phase protocol and a 'one-phase hierarchical deadlock detection protocol'. A brief summary of these schemes follows.

The two-phase detection protocol makes use of local resource status tables and polled information to construct a system-wide demand graph. Maintained at each node is a status table which contains, per originating process, entries for resources locked and resources waiting. The appointed control node constructs a system demand graph from the status collected from the local nodes. If no deadlock is found from this first poll then the system is not in deadlock and the protocol terminates. On the other hand, if deadlock is detected, the nodes are polled again and a sub-system demand graph is constructed by using only the transactions presumed to be in deadlock from phase 1. Should this also result in a directed cycle in the demand graph, the system is, in fact, in deadlock and the 'deadlock resolver' is notified. Otherwise the second phase indicates that the system is not in real deadlock and the protocol terminates. The policy for

determining the control node can be accomplished using a circulating privilege in which every node has a turn.

The one-phase protocol accomplishes deadlock detection in one communication cycle by relying on more data locally. In addition to the resource status table, each node must also manage a process status table which identifies transactions that are owned by processes at the node. In this protocol, all entries contain timestamp information which is communicated when appropriate (eg. a request for a remote resource will result in a local entry identifying the transaction, time, and whether it is *waiting* or *assigned* depending upon whether or not the resource is free; upon release request, the transaction identifier is searched in the process status table for the status entry of *assigned*; when found it is removed). To perform detection in this scheme, a node is again chosen as controller to collect all the resource and process tables. The controller first broadcasts a 'request for tables' message to all nodes then waits until all tables have been received as acknowledgment. Next, the controller constructs a demand graph using information which agrees in both the process and resource status tables (can be viewed as an intersection of relations over common fields). As a result of this step, no directed cycle implies no deadlock while a directed cycle signifies a 'real' deadlock situation in the system.

In the hierarchical detection protocol, the set of nodes over which the file system is distributed, is partitioned into clusters to facilitate a distributed form of deadlock detection. The protocol strives to detect deadlock by applying the one-phase protocol (above) within each cluster and then between clusters. Note that this protocol also employs dynamically designated control nodes at both the intra-cluster detection phase and the inter-cluster phase. Each local control node constructs a demand graph by performing the one-phase protocol within the cluster; this demand graph (of each cluster) is routed to the "inter-cluster control node" where an inter-cluster demand graph is assembled. The demand graph, which is only constructed for those entries which agree between the resource and process status tables, is similarly checked for any directed cycles. A directed cycle indicates that there is deadlock, otherwise there is no deadlock and the protocol terminates.

Any one of the three deadlock detection protocols presented above could be implemented within the XDFS structure. In XDFS, since accesses to the file system are accomplished through requests initiated by a client (on behalf of users) to one or more servers, deadlock must involve at least one server and one client. Since each node is dedicated as either a server or client (ie. server nodes and client nodes), it may be feasible to employ a deadlock detection process at each of the nodes in the system. This arrangement enables the capability to detect deadlock at a inter-transaction level as well as at a inter-node level. In applying this mechanism, any

deadlock process could become the controller for the current run of the deadlock detection algorithm.

The tradeoffs to be assessed in choosing the appropriate algorithm include: algorithm complexity, polling time involved, amount of memory required for status tables and, the traffic volume attributable to polling deadlock status information required by the controller (ie. the communication overhead). However, polling within the XDFS may be too inefficient to be practical since it relies on a contention-based communication network. That is, the time required by the controller to poll the clients may be fairly large due to the local network access time (time required by the controller to 'gain control'). In this case, the 1-phased hierarchical protocol may be the best solution. Polling is limited to smaller groups; inter-cluster polling may be smaller assuming an equitable partitioning.

The tradeoff as stated, is increased data that the controller nodes must manage. Within XDFS, timeout mechanisms are used to prevent deadlocks. Recall that a server may break a client lock under certain conditions. When this occurs, the server will also notify the clients involved. To adopt one of the deadlock detection methods described above, XDFS server and client responsibilities would change. No longer would a client rely on a lock break notification from a server; the timeout mechanism used by XDFS servers would not be present. Hence, servers would no longer be allowed to automatically break locks which have been set for a long time. Since clients are the originators of transactions, it would be logical to select a client as the controller for the deadlock detection protocol. (Servers could be used although this may be more difficult/inefficient due to transactions involving multiple servers). To avoid a cascading abort scenario once deadlock is detected, the controller would also have to decide how to resolve the situation (eg. cancel one of the transactions). Adopting one of the deadlock detection mechanisms described, however, has a major disadvantage. No longer can clients maintain local data caches of file data since this relied on notification of broken locks from servers.

4.4.5 Recovery Data Management

The protocol employed by the software to guarantee stable storage is to write each stable page sequentially in logically geographicaly separated areas of the disk. A brief overview of the concept of stable storage [Lampson-Sturgis] as developed by Butler Lampson follows.

The physical representation of stable storage described above, coupled with the following rules form the basis of the stable storage guarantee [Lampson-Sturgis]:

- a file page must have two correct stable copies
- the second file page is not written before the first write completes successfully.

The two cases to be handled by the recovery software upon detection of a bad write are either 1) the write error has been isolated to one stable page or 2) a write error did not take place during the write of either page. The first case represents a failure during either the first or second stable page write (after the first has completed successfully) and is resolved simply by updating the bad stable page with the correct stable page. In the second case, the failure was detected in-between writes (resulting in one correct updated page and one 'old' page) and is solved by updating the second stable page with the contents of the first stable page. (Since the first page 'leads' the second, this approach will guarantee that the file will be updated to 'most current').

An intention list contains all information regarding operations of a transaction and can be written to stable storage as an atomic operation. An important characteristic of intention lists is that processing the list partially several times then completing it has the same effect as completing it fully exactly once. This is attained by representing the write actions which comprise the intention list, as specific address value pairs; address is the stable storage address of the file contents to be modified, value is the new contents to write at address. Intentions lists requiring more than one page are written in reverse order: subsequent pages first, then the head page. Since an intention list does not exist without a head page, crashes occurring before the completion of the write will result in no "partial" intention list (ie. no list at all).

To recover from a server crash, the set of intention lists are assembled and those which are confirmed are processed. Tentative intentions lists must be confirmed with the coordinator for the transaction. As mentioned previously, all server-client correspondencè is either 'request-response (client-to-server initiated or server-to-client initiated) or 'unsolicited messages' (strictly server-to-client initiated). Because of this, a situation could occur where a server crashes after receiving a *EndTransaction* request but before generating an acknowledgment reply to the client. When the server recovers and issues a negative reply (which may be due to the fact that by now, the transaction is no longer active) the client must decide whether or not in fact the transaction completed. To handle this situation, XDFS servers maintain a 'most recent completed transaction' list. Clients can interrogate this list to see whether or not a transaction has actually completed.

4.4.5.1 Issues and Alternatives.

In general, the intentions lists form a log of transaction activity for the XDFS and can be thought of as checkpoint data. Checkpoint frequency is on a XDFS file request basis and the life of the checkpoint is for the duration of the transaction. From this perspective, checkpointing as it relates to the XDFS will be analyzed with respect to McDermid's three basic checkpoint requirements [McDermid].

System restore to a globally consistent state. From the above discussion it is evident that the stable copies provide a method of maintaining consistent file data.

Strive to minimize the amount of recovery data. XDFS recovery data consists primarily of intentions lists on a transaction basis. Recall that an intentions list is maintained on stable storage and contains pointers to updated file pages and the necessary operations to affect those updates for its associated transaction. Since this file contains just pointers to new pages (and not the updated pages themselves) and operation codes, this represents a small amount of data.

Support for incremental restore. In the case of crash recovery, only the incomplete intentions lists within XDFS are reprocessed by the servers involved. Clients recover from exceptions, however, by redoing the entire transaction. This usually requires more than rewriting previous values since data which depended on those values may no longer be valid (ie. may have changed since the crash). XDFS provides some assistance to clients to deal with this by allowing clients to break locks which have been set for a long time and by notifying clients of broken locks. These mechanisms enable clients to maintain local caches of server data. In the event of a server crash, a client can reread the data involved in the transaction and if unchanged (from previous cached value(s)), then the previously computed results are still valid and can be rewritten without recomputing.

In XDFS a log (intentions list) is maintained on a stable file which describes the updates to perform at commitment on the (non-volatile) file(s). An alternate approach to recovery data management within the XDFS could be based upon using shadow copies in which changes are made to file copies and at commitment the copies are made the real files. An obvious advantage of such a scheme is simplicity in both the method itself as well as in the recovery algorithms to support it. Using shadow copies, changes made to the 'current' copy are not made to the 'real' (or shadow) until commitment. During an abort or cancellation the 'current' copy is discarded. From a practical standpoint however, this is not realizable within XDFS. Since locks are implemented at the byte-range level, a potentially large number of file copies could be required for each 'real' file involved in a transaction.

5. Recovery Within a Pool-Processor Distributed System: The Newcastle Connection

5.1. System Configuration

A fault tolerant distributed file system (referred to as The Newcastle Connection) developed by University of Newcastle upon Tyne is implemented within the pool-processor distributed system model. System architecture configuration consists of a Cambridge ring network of Digital Equipment Corporation LSI 11/23 and PDP 11/45 processors each with approximately 10MB of local disk st. ge. Each node in the system runs the Unix operating system. Chief goals of the file system were to provide distributed support for Unix files and provide to users an abstract recovery capability [Jegado]. To support this, the Distributed Recoverable File System (DRFS) contains two unique aspects: an extension to the local Unix file system providing distributed access to files, and a general mechanism for the creation and deletion of recovery points. The DRFS was built as an application on top of Unix and was implemented in the C programming language.

Communication within the distributed system is accomplished through a cambridge ring network. The cambridge ring is characterized by point-to-point twisted-pair connections between nodes in the system and can support up to 255 nodes [Panzieri-Shavristava]. Active interfaces are employed in this token controlled ring to allow each node the ability to temporarily disconnect the input from the output and instead insert data. The token-based access control eliminates the need for a control node to initiate messages. In the cambridge ring, each sender is responsible for removing its messages from the ring.

5.2. File System

5.2.1 Overview of the Unix File System and DRFS

DRFS is organized in two 'layers': the Distributed Recoverable File Manager (DRFM) layer and the Local File Manager (LFM) layer. The LFM layer is built on top of the existing Unix file system. Each machine has a DRFM and a LFM. In general, the DRFM manages the user interface and is responsible for handling the distributed nature of files. An LFM handles the interactions with the underlying Unix File Manager (UFM) of the local processor. Access to files

on another machine is accomplished by prefacing the file path name with the machine name. Thus, the user must have a priori knowledge of the file-machine mapping in distributed file requests accessed through the DRFS. The DRFM services a distributed file request by generating a remote procedure call [Panzieri-Shavristava] to the appropriate LFM to perform the request. A general description of the Unix file system follows.

Unix employs a hierarchicaly organized file system which supports demountable volumes. Three types of files exist in the Unix file system: ordinary files, special files, and directories. Ordinary files are treated as unformatted collections of bytes with no structure imposed by the system. Directories, which provide the logical to physical mapping of files, are each treated as an ordinary file with access rights. Special files are used to provide transparent interfaces to peripheral devices attached to the system [Ritchie-Thompson]. File input and output is handled through system calls. For example, the 'open file' command requires a file name and access mode and returns a file descriptor to be used in subsequent acceses involving the file. The read command requires the file descriptor, a buffer pointer and the number of bytes to read and it returns the actual number of bytes read. Random access to files is supported through the lseek command which requires the file descriptor, an offset and base value and returns an index into the file. Finally the close operation uses file descriptor to close the associated file.

A directory type file contains a collection of file entries each of which consists of a file name and a file pointer (*i-number*). In addition to identifying the files in a directory this mechanism also provides a logical-to-physical mapping for file names. The *i-number* indicates the file's initial *inode* in the *ilist* which contains: user id, group id, protection mode, thirteen physical address pointers, file size, timestamps for creation, last use and last update, number of links to the file (ie. number of directories the file exists in), and the file type code (one of {directory, ordinary file, special file}). The thirteen pointers suffice to provide three layers of indirect mapping to fixed size blocks: the first ten pointers point directly to fixed size blocks; the next (indirect) pointer points to a fixed number of pointers, each of which point to the fixed size blocks; the twelfth pointer points to a fixed number of double-indirect pointers; lastly, the thirteenth pointer points to a fixed number of triple-indirect pointers.

Since a directory contains information on where to find a set of files (any of which may be a directory file type), the organization leads to a natural acyclic directed graph structure. The tree path of directory file names to a file forms its fully-qualified name. In the example below, /usrA/tmp/docs/doc1 refers to a document while /usra/tmp/docs refers to a directory containing the files doc1 and doc2.

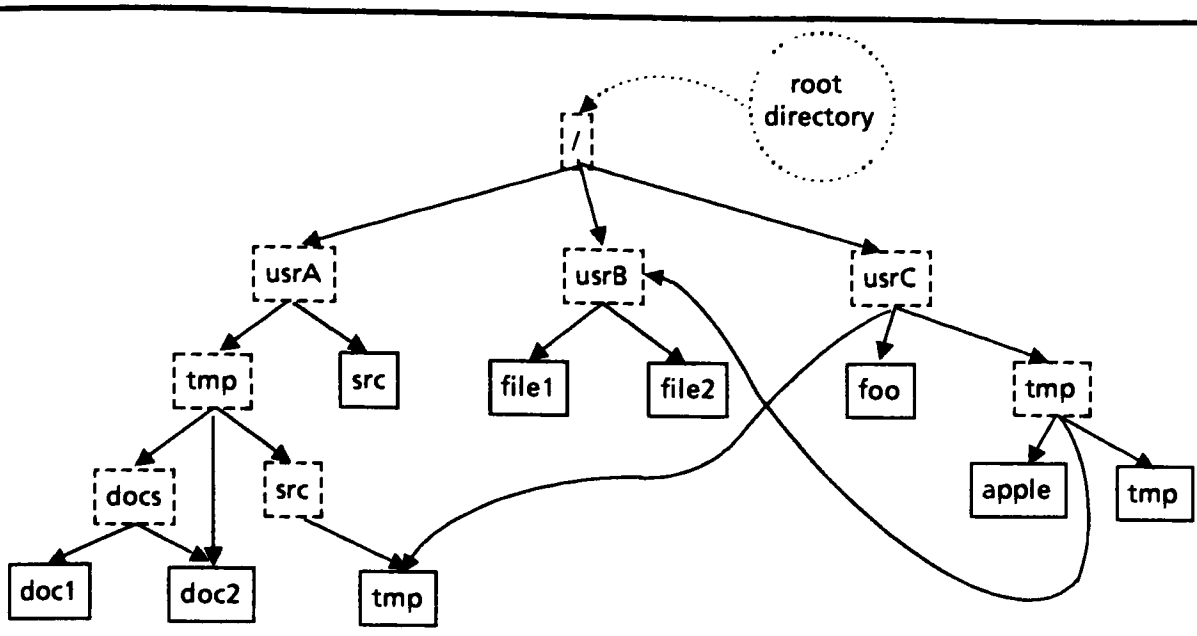

Fig. 5-1 Unix Hierarchical File Structure

5.2.2 Overview of Distributed File System

The DRFS utilizes distributed recoverable file managers situated at individual nodes to create the illusion of an extended Unix file tree structure. In this context, the logical structure of the file system can be viewed as a super tree with the collection of all root nodes forming a 'super-root'. Thus the DRFS gives the impression to each user that there is a 'higher' directory level (representing the super-root) which contains the set of machine names. From this set, each machine has its own local file system. The distributed recoverable file managers at each node are responsible for providing this logical view to users. Note that a DRFM does not deal explicitly with files themselves; it is only responsible for providing a distributed view of the file system as a whole. Each node also has a Local File Manager which interfaces directly to the Unix file system. Thus, DRFS is comprised of a DRFM, LFM and UFM process at each node of the file system (this structure is depicted in Fig. 5-2.).

On a local file request, a user could conceivably use the LFM services (or even use the Unix file services directly) local to the machine. The advantage of using the LFM, however, is support for recoverable Unix services. In fact the LFM is implemented as a local Unix File Manager with recovery capabilities. A user submits a series of DRFS requests to the local DRFM in the form of

a transaction. A DRFS transaction consists of a *BeginTransaction*, a series of file request operations and concludes with an *EndTransaction* operation. In the DRFS concept, a transaction is a useful user abstraction which represents a logical collection of file requests. (As it will be seen, recovery information is recorded with respect to the recovery region(s) of a transaction; retention of recovery information is associated with the life of the transaction.).

5.3. Recovery Mechanisms

DRFS provides mechanisms to enable fault tolerance within the distributed file system. These mechanisms include crash resistance for files and support for backward recovery within user transactions. Before the detailed mechanisms are presented, an overview of the crash resistance strategy as well as the recovery process at the user level is in order.

Crash resistance is provided by the DRFS by adopting a strategy whereby all pre-commitment changes are carried out on temporary files, and then deleting all temporary files on system restart after a crash.

From an operational perspective, recovering from an exception in a user transaction generally involves determining the appropriate recovery action to take (based upon exception type and severity) and, where appropriate, restoring consistency to some predetermined safe state using appropriate recovery information. In fact, these two steps are very related. As it will be seen, recovery information is collected by DRFS based upon user-defined sequences of file operations called recovery regions. Also, associated with a recovery region, a user may specify an exception handler which determines the appropriate recovery action(s) to perform for the file operations within the recovery region.

Next, the DRFS mechanisms necessary to support recovery at the user level are presented.

5.3.1 DRFS Recovery Mechanisms

Recovery mechanisms are in place at both the LFM and DRFM level to ultimately support a distributed recoverable file system. At the DRFM level, the basic logical information unit is a transaction; at the LFM level, the basic unit is a file operation (the DRFS logical structure is illustrated in Fig. 5-2.). LFM file operations are implemented as recoverable Unix file operations. Support for recoverability is provided to users from the DRFM through the ability to create and delete recovery checkpoints. Shrivastava's concept of inclusive recovery regions [Shrivastava81] is central to this recovery mechanism. Using recovery regions, user-specified ranges of transaction file operations (possibly nested), each of which are bounded within a recovery region are made recoverable. Recovery regions act in conjunction with the supporting exception signalling and handling mechanism to support fault tolerance within the DRFS.

A recovery region bounds file operations which are guaranteed by the DRFS to be recoverable. A recovery region is established using the atomic operations establish recovery point (*erp*) and drop recovery point (*drp*). In particular, users can construct a "recoverable transaction" by enclosing the operations of the transaction within a recovery region. Recovery regions can also be nested to provide a hierarchy of recovery levels. Problems in supporting nested recovery regions, however, include uniquely identifying multiple recovery points and deciding which recovery point (and when) to drop. Shrivastava solves these problems by associating a unique identifier (*rpn*) with each recovery point and employing a convention for discarding recovery points. On establishing a recovery point, a unique identifier is returned. In handling exceptions, this identifier enables the restore recovery point operation (*restore*) to restore the file system to different levels of recovery region entry points (ie. *restore(rpn)*).

When an exception is propagated from within the DRFS indicating that a file request could not be performed, the exception handler associated with the recovery region is invoked. There are two classes of exceptions signalled by DRFS components (ie. DRFM, LFM, UFM): failures and not-done exceptions. Not-done exceptions indicate that a request could not be performed; for these exceptions, recovery is attempted before propagating the signal to the next higher level. Failures are treated a bit different within DRFS; they indicate that consistency can not be restored. When this occurs to a DRFS distributed recoverable transaction, it is aborted, which causes the effects on files modified by the transaction to be undone. It is the not-done exceptions, therefore, which users may provide recovery for.

Within the DRFS, before a not-done exception is signalled, the exception handler associated with the operation must first undo all effects of the operation up to that point (via forward recovery as discussed in 5.4.5). Ultimately, the not-done signal is propagated to the user transaction where the exception handler associated with the recovery region containing the unsuccessful request is enabled to either take care of the malady or pass it on to a higher level (details to support this are discussed in more detail in 5.4.5). Thus, an exception handler can be associated with every recovery region to perform a *restore(rpn)* on the recovery point number (*rpn*) of the associated recovery region. In this manner the file state may be restored to its context at the beginning of the region (at the *erp* point). With recovery regions and exception handlers, a user can adapt the granularity of recovery to the application.

A typical recovery strategy implemented by a user may be to simply restore the file state to its prior state at the start of the recovery region (see Fig. 5-3.). Alternatively, a user may elect to simply abort the transaction (in this case, the *abort* command would be used). For other exceptions, a 'wait-and-retry' strategy could also be implemented.

As indicated, recovery is attempted by a DRFS component before issuing a 'not-done' exception (this is not the case of failures). It is interesting to note that restoring consistency in such instances is attempted through forward recovery. The mechanisms in place to support backward recovery to users were not used primarily because Unix does not support backward recovery but also for reasons of efficiency [Jegado].

In the next section, the DRFS mechanisms are analyzed with respect to the recovery criteria outlined in Section 3.

5.4. Analysis of Recovery Mechanisms

5.4.1 Atomicity Guarantees

Atomicity is present at several levels of abstraction within the DRFS. At the UFM, the Unix file operations are assumed atomic. This assumption enables the LFM to attempt to restore consistency in response to a not-done exception at the UFM interface through forward recovery. Similarly, the DRFM provides a degree of atomicity for file operations. If an exception occurs at the LFM interface, the DRFM attempts forward recovery on the LFM operation. Ultimately, transaction atomicity at the user level can be implemented using the backward recovery capability supported by the DRFS.

Mechanisms associated with guaranteeing atomicity within the DRFS include support for: crash resistance, user-defined recovery regions, and internal DRFS recovery mechanisms.

Crash resistance guarantees that if a failure occurs before commitment, updates to file(s) in a recoverable transaction will not be performed. A recoverable transactions is one in which the entire transaction is enclosed in a recovery region. A recoverable transaction has the property that, if a crash should occur (or propagated exception), the state of a file(s) modified within the

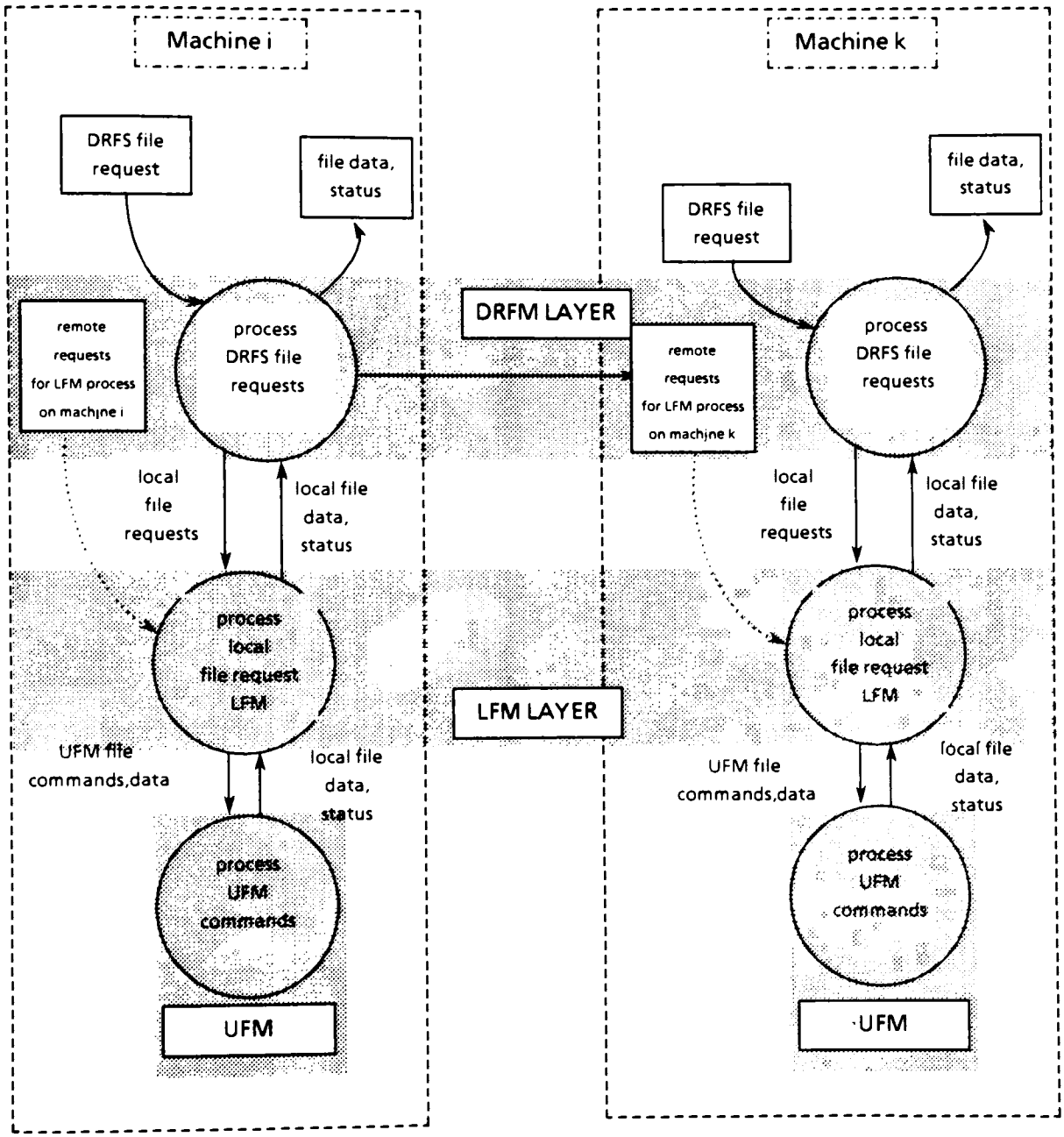

Fig. 5-2. Logical View of DRFS Structure

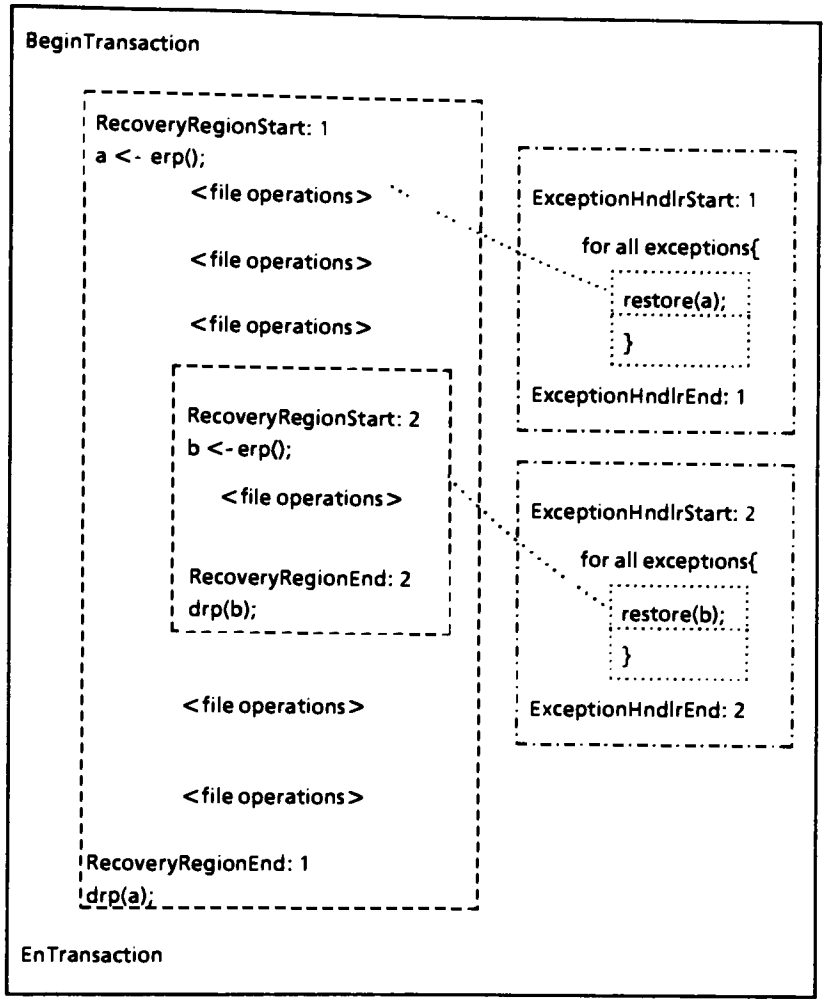

Fig. 5-3. A User Transaction Using Two Recovery Regions

transaction can be restored to the point established at the beginning of the transaction (ie. the start of the first recovery region).

5.4.1.1 Issues and Alternatives

Invoking recovery to the first recovery point for all user-level propagated exceptions, however, is an effective but inefficient way of restoring consistency since partially completed transaction results are lost. DRFS support for nested recovery regions within a transaction allows a more elegant recovery approach to be constructed. For example, using nested recovery regions, one

could enclose the non-local sequential file operations of a transaction within a recovery region. If a server exception should occur on the non-local file access, results prior to the exception may still be valid. The exception handler associated with the non-local access recovery region could decide to abort the transaction, continue processing (may be valid if the correctness of the transaction does not depend upon the data), or retry (server 'temporarily' down). Using this approach, a user may view a transaction as a collection of atomic file operation sequences.

The generic nature of the recovery mechanism as a tool for building fault tolerant file systems from recoverable file transactions is powerful. Although the interface is consistent and well defined, the primary responsibility of implementing recovery, however, is shifted from the system to the user (as with XDFS).

The DRFS approach to atomicity satisfies the stated requirements of atomicity (see 4.1). All modified values within the transaction can be restored to previous values by reverting to the first recovery point. Concurrency control and transaction coordination serve to guarantee that updates to a file are not made available until the transaction completes. File updates, performed by the LFM, are actually carried out on file copies; when the transaction commits, the copy is then made the original. In applying these mechanisms, the DRFS guarantees that a recoverable transaction will either complete (proceed to commitment) or not alter the previous results (the transaction is aborted resulting in the restoration of the first recovery point).

5.4.2 Exception Handling

Abnormal behavior at the LFM level which results in an inability to perform a requested file service is classified as 'failure' or 'not done' exceptions. Not-done and failure exceptions are distinguished by the ability to respond to problems in a predetermined orderly fashion. Not done exceptions are those errors which have been logically detected by the software and for which recovery is possible; failure exceptions are signalled when a fault occurs that can not be handled by any exception handler.

The DRFM and LFM each follow identical protocols for handling exceptions. When a not-done exception occurs, an attempt is made to restore consistency (and hence mask the event from higher level(s)) through forward recovery. If unsuccessful, the exception is propagated to the next higher level. No attempt to restore consistency is made for failure exceptions; these are always propagated to the next-higher level. In such a case, the DRFS aborts the transaction. If a crash should occur while processing a recoverable transaction, the file changes performed up to the time of the failure exist on a temporary file; all temporary files are subsequently deleted at system restart.

If an exception signalled by a system component goes uncaught by both the LFM and DRFM, the DRFS will "simulate a crash event by aborting the transaction" [Jegado]. To abort a transaction, the LFM discards the session information associated with the transaction (see 5.4.5) and sends an abort to the DRFM. This informs the DRFM to cleanup the transaction and broadcast an abort to the other LFMs involved in the transaction (as determined from the session cache). To recover from DRFM crashes, the protocol implemented is to leave the session connected and terminate when a new connection is made. (Jegado also offers a LFM session-abort timeout as another solution to the DRFM crash restart problem [Jegado]).

Using recovery regions, a user can layer (nest) levels of recovery. Each recovery region can either deal with an exception or pass it on to a higher level of recovery abstraction (eg. an outer recovery region).

5.4.2.1 Issues and Alternatives

In handling exceptions within the DRFS after detection, an obvious alternative approach could have been based upon the inherent backward recovery facilities provided to users (ie. recovery regions). Prior to issuing any exception, a component could restore the system to the previous consistent state by using the recovery region concept: a recovery point established at the beginning of the component handler would either be dropped if processing was successful or restored to invoke recovery in the presence of a not-done exception. However, since the Unix system does not provide support for backward error recovery, and because of the associated complexity of managing recovery information at multiple levels (ie. user level as well as LFM, DRFM levels), forward error recovery was used instead for handling exceptions within the DRFMs and LFMs [Jegado]. Also, since the UFM operations are assumed atomic, backward error recovery support for this interface would be inefficient; a more effective approach is to

The DRFM and LFM each follow identical protocols for handling exceptions. When a not-done exception occurs, an attempt is made to restore consistency (and hence mask the event from higher level(s)) through forward recovery. If unsuccessful, the exception is propagated to the next higher level. No attempt to restore consistency is made for failure exceptions; these are always propagated to the next-higher level. In such a case, the DRFS aborts the transaction. If a crash should occur while processing a recoverable transaction, the file changes performed up to the time of the failure exist on a temporary file; all temporary files are subsequently deleted at system restart.

If an exception signalled by a system component goes uncaught by both the LFM and DRFM, the DRFS will "simulate a crash event by aborting the transaction" [Jegado]. To abort a transaction, the LFM discards the session information associated with the transaction (see 5.4.5) and sends an abort to the DRFM. This informs the DRFM to cleanup the transaction and broadcast an abort to the other LFMs involved in the transaction (as determined from the session cache). To recover from DRFM crashes, the protocol implemented is to leave the session connected and terminate when a new connection is made. (Jegado also offers a LFM session-abort timeout as another solution to the DRFM crash restart problem [Jegado]).

Using recovery regions, a user can layer (nest) levels of recovery. Each recovery region can either deal with an exception or pass it on to a higher level of recovery abstraction (eg. an outer recovery region).

5.4.2.1 Issues and Alternatives

In handling exceptions within the DRFS after detection, an obvious alternative approach could have been based upon the inherent backward recovery facilities provided to users (ie. recovery regions). Prior to issuing any exception, a component could restore the system to the previous consistent state by using the recovery region concept: a recovery point established at the beginning of the component handler would either be dropped if processing was successful or restored to invoke recovery in the presence of a not-done exception. However, since the Unix system does not provide support for backward error recovery, and because of the associated complexity of managing recovery information at multiple levels (ie. user level as well as LFM, DRFM levels), forward error recovery was used instead for handling exceptions within the DRFMs and LFMs [Jegado]. Also, since the UFM operations are assumed atomic, backward error recovery support for this interface would be inefficient; a more effective approach is to

restore the DRFM by forward error recovery. Forward recovery within the DRFS is thus simpler since it relies on the recoverability of the LFM operations.

An exception handling strategy could have been based upon distributed detection of failures among nodes in a distributed system [Hosseini]. This scheme works well for errors at the node level but doesn't address (specifically) exceptions between processes (ie. transactions). This decentralized method of exception handling relies on a distributed implementation [McDermid] and strives to guarantee a correct and operational set of available sites. Each site tests itself and associated neighbors to determine a constant correct node set ('down' sites are subsequently avoided).

An alternate to the DRFS approach to exception handling could be based upon a form of centralized exception handling. Under this approach a control process could be active at any site in the system although only one process would be in control at any one time (ie. a rotating privilege). In this scheme, the central control process would monitor all DRFM and LFM processes in the network for exception conditions. In the event of an exception, the control process then activates the appropriate procedure on the corresponding processor (e.g. via remote procedure call) to handle the exception. Note that this centralizes the exception detection and signalling function. However, in addition to the increased overhead attributed to providing a dedicated process available at each node, this scheme may also require significant processor polling time (to obtain status information). Another drawback is the detail and amount of information and decision making the control process must perform (ie. level of exception reporting). The advantage of the approach taken within the DRFS is in its consistency in dealing with exceptions detected at both the DRFM and LFM and in its rather simple and efficient implementation. Each layer either processes the exception (ie. knows of the exception and responds to it) or it passes it on. Thus the tradeoff in this instance lies with a potential rippling effect of exceptions (which would have been avoided in the centralized case) versus a well defined interface at each layer (DRFM, LFM). The DRFS decision made in favor of the simple interface has the advantage of potentially being better suited to accommodate change at individual nodes (ie. can tolerate individual exceptions on a node by node basis). In the centralized approach it would be harder to accomodate individual exception conditions.

5.4.3 Concurrency Control

To handle concurrent file service requests on a file, exclusive locks are used. Locks may be acquired before use or on demand. The scope of the lock is the entire file and it lasts until commitment (or transaction termination due to willful or forced abort).

This protocol reduces the complexity of the recovery and crash resistance strategies. Problems associated with having to provide crash resistance and recovery data for concurrent transactions which operate on a common file simultaneously are eliminated.

5.4.3.1 Issues and Alternatives

The DRFS concurrency protocol is effective but does not promote a high degree of parallelism. This may be attributed to the fact that the main design goals of the DRFS were network-wide file accessibility and recoverability. Note that the implementation of the DRFS really lies in between the application layer and the operating system layer. Hence, providing concurrency control for transactions at a finer granularity may have significant performance impacts. A more efficient approach would be to incorporate new mechanisms into the kernel or use existing control primitives already in the kernel. A slight variant to this is the kernel-based transaction manager [Jessop]. Such approaches, however, would go against the underlying goal of not altering the Unix system [Jegado]. Possible alternatives which may be realizable at such a pseudo-applications layer include timeouts [Dolev], or finer-granularity file locks.

Page-level locks would offer a higher degree of concurrency. This would potentially allow multiple non-conflicting readers and writers of the same file. With this advantage comes the associated complexity of the concurrency control algorithm. The added complexity involves guaranteeing mutual exclusion between locks granted and in maintaining a correct and stable lock access mode to file page map.

A slight variant to this which fits directly into the UNIX structure is to implement locks at the driver level [Weinberger]. Lock management techniques could, for example, be incorporated into a disk driver which has the responsibility for controlling the physical disk for file data transfers. The reasons cited for such a proposal are: the interface is well defined and simple, and since the implementation is at the system call level, a process using the protocol "would not have to give up the CPU" [Weinberger]. In this scheme the UNIX *ioctl* system call would be used to acquire locks from a lock subdirectory within the device directory (eg. /dev/db-lock) and return the lock or an error (to be subsequently handled by the transaction). As Weinberger points out, this has the advantage of not requiring a separate monitor process for concurrency control [Weinberger] and therefore avoids queueing delays.

5.4.4 Deadlock Detection

Although deadlock detection is not performed by the DRFS, users can implement this capability using the recovery region concept since the UFM lock operations are made recoverable at the LFM level. That is, an exception raised from within the LFM signalled by a UFM component because a service cannot be performed can be used by a higher level to eliminate the condition. This can be done, for example, by aborting the transaction altogether or invoking recovery to a point prior to the deadlock condition (thereby freeing the resources).

5.4.4.1 Issues and Alternatives

In distributed file systems which implement concurrency control through locks, deadlock detection is necessary for providing fault tolerance. Some mechanisms to handle deadlock which are suitable to the DRFS, include decentralized detection schemes (see 4.4.4.1) and timeouts.

Any of the three protocols outlined in 4.4.4.1 could be applied to the DRFS. The tradeoffs which must be analyzed in determining the most desirable protocol include: required response of the detection protocol and amount of memory available at each node for deadlock detection. The 2-phase protocol involves 4*n polling overhead (transmit + reply for a n-node system). The one-phase protocol avoids an additional communication polling phase but also requires additional data management overhead (for now an additional table is required) at each node. The hierarchical method offers potentially the best response (since deadlocks detected at the cluster level can be notified at once), however it may not be efficient or practical for distributed file systems employing a small number of nodes.

A timeout mechanism could be integrated into DRFS to provide a higher degree of reliability in detecting and recovering from deadlocks. This alternative, however, assumes that a deadlock situation is time bounded (ie. the timeout must be longer than the longest expected transaction processing time). Also, the timeout approach imposes timing behavior on transaction operation which must be addressed.

5.4.5 Recovery Data Management

Upon notification of a recovery point from a DRFM, an LFM establishes a recovery point locally. In this respect, the protocol models distributed checkpointing [McDermid]. As previously stated, the effects of a file transaction are undone (that is, all updates to a file within the recovery region of the transaction) by restoring the first recovery point.

Principle DRFS recovery data structures involved in providing backwards recovery to user transactions are: the session cache, the commit cache and the undo cache. The session cache contains file specific information on a user basis; there is one entry per user file. The session cache is managed by the DRFM and is updated as files are opened/closed. A LFM manages undo and commit caches for each session. Operations to be undone are contained in the undo cache while file identifiers representing files involved in uncommitted transactions are held in the commit cache. Crash resistance is provided (even using nested recovery regions) by performing updates to individual file copies at the LFM layer. Updating a block within a file involves first reading the block from the copy (a copy of the entire file is created on the initial update) into the undo cache then writing the updated block to the file copy. Instead of transferring the entire block to the undo cache, the pointer to the area in the 'block-pool' where the block is copied is written to the undo cache. This procedure is required to handle file updates within nested recovery regions. At commitment, the undo entries are dropped and the commit cache is processed by determining the file changes required to affect the updated file state, and then carrying them out on the actual (disk) file. As Jegado suggests, a successful transaction involving a file update would be performed in non-volatile storage in three steps. First, the file copy would be renamed to the original file. Next, the original file is deleted and the storage released. Finally, the lock on the newly updated file (former copy) is unlocked [Jegado]. Since the file updates are performed to the actual file on disk, a consequence of the procedure is that the effects cannot be undone after commitment.

At the LFM layer, a recovery point is established either in response to a file system request (ie. local recoverable Unix file operation) or upon reception of an 'establish recovery point' message broadcast from the DRFM. To ensure reliability in communicating between the DRFM and LFM layers, the establish, drop and restore recovery point operations are implemented using a reliable procedure call mechanism [Panzieri-Shavristava].

5.4.5.1 Issues and Alternatives.

Recovery data in DRFS is managed using a distributed checkpointing strategy. On a transaction basis, a DRFM directs the LFM(s) involved to record local recovery data for each recovery region. An alternative to this approach would be to use global checkpointing in which periodic snapshots of the entire system state are taken collectively. Using this method, a DRFM for example, would collect recovery information from all LFMs to form a single system-wide checkpoint for the transaction. Obviously, this approach is very costly in terms of time and space. An advantage to global recording, however, is the ability to centralize recovery functions

and hide such details from 'object managers' (ie. when using incremental recording, 'object managers' must provide their own recovery functions). The DRFS scheme, however, does satisfy McDermid's [McDermid] basic requirements of checkpoints.

System restore to a globally consistent state. Restoring the system to the first recovery point established undoes all effects of the transaction beyond that point. When recovery is invoked by a user through a local DRFM, each LFM involved in the transaction will process its respective undo and commit caches accordingly. Consequently, the set of files involved in the transaction are restored to the state at the time the recovery point was established.

Strive to minimize the amount of recovery data. The alternative to incremental recording of recovery information is global recording. This would increase the DRFM memory requirements for recovery data, since each would be required to periodically capture the entire state of files involved in transactions it manages. As McDermid suggests, minimizing the amount of data can be accomplished through minimizing checkpoint frequency or by storing data which has been stable for awhile. In either case, it is difficult to prevent obtaining some non-stable checkpoint data through global checkpointing.

In addition, the implementation of the LFM block update procedure strives to minimize data held in the undo cache by not storing blocks for the current recovery region which already exist in the undo cache [Jegado].

Support for incremental restore. Recovery data managed by the DRFM is recorded incrementally. Each LFM in turn incrementally records information in the undo and commit caches on a recovery region basis. Because of this, restoring a set of files to previous correct states after an exception, involves only the file(s) involved within a recovery region.

DRFS utilizes both logging and shadow techniques of recording recovery data due to the complexity associated in supporting nested recovery regions.

The objective of crash resistance in DRFS is to prevent adverse effects of a crash on distributed files in a transaction. This feature is provided by performing transaction updates to volatile file copies and following a protocol which initially restores all volatile memory to nominal values

then deletes temporary files at system startup after a crash. Crash resistance as described is only provided for transactions using the recovery region mechanism (ie. a file copy is made at the first recovery point established). Thus, when a crash occurs before commitment, all file copies representing the uncommitted updates are subsequently deleted thereby maintaining the previous (non-volatile) file states.

As Jegado indicates, the recovery data management scheme can be made more efficient by using a more efficient file representation such as representing a file as a sequence of pointers to page blocks and then on update copy only blocks which have changed [Jegado]. The advantage of this scheme is potential speed increase due to smaller file copy sizes; the disadvantage is that now pointer to page file mapping is needed.

6. Future Directions

In this section new concepts and approaches to problems relevant to fault tolerant distributed file systems will be explored. It is the intent of this section to provide a general overview of these concepts and to relate their general applicability to the issue of fault tolerance in distributed file systems.

6.1 Nested Transactions

Nested transactions as proposed by Moss [Moss], impose a hierarchical structure to dependent transactions. In this scheme, a transaction *A*, which immediately surrounds a transaction *B*, is called the parent of transaction *B*; transaction *B* is termed child. Thus a child has only one parent defined as the most immediate transaction which encloses it. A constraint on a child is that it must complete before the parent (either normally or by abort). The scheme provides for the parent to obtain the locks held by a completing child. It is also important to note that a parent cannot interfere with a child; it can only interact with external transactions--parents or those not directly related. A transaction commits when the transaction "root" completes, that is, when the outermost transaction (ie. the transaction with no parent) completes. Nested transactions provide synchronization between parents as well as between children. Children interact with parents but not among themselves). Similarly since parents cannot interfere with children, parents are mutually exclusive. This structure imposed by the method enables both parents and children to each be run concurrently.

Nested transactions, as explained above can become a great tool in synchronizing concurrent aspects of transactions. Transactions can be structured into logical dependent relationships, (ie. parent-child relationships) then the composite set can be executed in a guaranteed way.

Not only does this have an influence on atomicity within distributed file systems, but this also serves as a structuring tool to build "reliable distributed systems" by constructing applications as fault-tolerant atomic actions. The lock inheritance rule allows a high degree of concurrency between parents and children. This scheme may also be found to be extensible as a method of providing a structured, layered exception handling mechanism similar to DRFS.

Nested transactions provide a novel way of handling recovery through multiple copies. Since the scheme relies on shadow copies there is potential for an increasingly large number of copies of a file to develop throughout the life of a transaction. The protocol employed simplifies this by having parents inherit the unique copies of children (ie. a file copy already possessed by a parent is discarded). The abort of a child does not automatically cause its parent to abort. Rather, the effects of the child are undone by the system automatically. A parent can therefore perform recovery as required on behalf of one or more aborted children.

6.2 Transaction Kernel

The transaction kernel idea [Spector-Schwarz] takes a macroscopic view of atomicity within the distributed system. The concept is developed with the goal of applying the reliability aspects of transactions as underlying constructs for programming reliable distributed systems. To support this in a generic context, shared abstract objects and associated dependency sets are used. Semantic knowledge of the abstract object types used within a transaction is applied to generate a dependency set which describes the abstract type processing order.

This scheme provides atomicity as a generic system guarantee to be used in constructing distributed applications. The goal of this approach is to embody those mechanisms necessary to implement a generic transaction manager into a kernel for efficiency. Obviously, this has advantages for concurrency control, exception handling and deadlock detection.

A transaction kernel at each node in a distributed file system would hide the recovery implementation details from programmers. Each kernel would be responsible for providing recovery for transactions. In this manner, transaction processing could not only be reliable but could also provide better performance. Increased performance may be realizable since some recovery data functions can be handled more efficiently from within the kernel (as opposed to the applications layer). Since all resource information is available to the kernel in most instances, errors detected in association with these can be monitored more efficiently. Note that the increase in potential size (perhaps complexity) of the kernel may be offset by increased speed in detecting the exception (less levels to go through) as well as less redundant exception handling mechanisms. (ie. where multiple transactions each check the same exception).

Similar advantages could be attained for concurrency. A transaction kernel making use of semantic knowledge could make better decisions on aborting/delaying transactions to promote a

greater degree of concurrency. As the authors point out, this implies the knowledge and implicit existence of dependencies within and among transactions. For example, consider two transactions each wishing to perform an update on the same file. Suppose also that the transaction updates are disjoint; that is, the range of bytes modified by one transaction does not intersect with the range of bytes modified by the other transaction. Under simple file size locking rules one of the transactions would be forced to wait until the other completes (due to commitment or abort). In a transaction kernel making use of semantic knowledge, both transactions would be allowed to operate concurrently. Note that although this seems very much like the XDFS byte-level locking scheme, there is a significant difference. XDFS provides support to *users* for byte range locking and does not exploit any knowledge of inter-transaction semantics. To attain the transaction kernel functionality in terms of the XDFS scheme, users of the system must know in advance the 'minimal update range' within affected files in a transaction and 2) must issue file system requests in accordance with this range. In a transaction kernel these decisions and details would be hidden from the user; the kernel would be presented with a set of transaction update requests and it would decide if any of the updates could be run concurrently.

However, there are also problems and disadvantages to a transaction kernel. There is obviously the problem of determining precisely the semantic relationship between (and within) transactions. Also, as the semantics for specifying a transaction changes, so must the kernel. (This may occur, for example, to provide more function to users or to take advantage of new hardware features.). A transaction kernel may be also harder to adapt to specialized applications (eg. unique recovery requirements). Another disadvantage is additional kernel overhead: functions which were once distributed (ie. transaction concurrency control and recovery) are now centralized creating a potential bottleneck. (This could be overcome, however, with a distributed OS/kernel implementation).

6.3 User Structured Atomic Applications

Shrivastava has developed a mechanism for building atomic user applications [Shrivastava82]. The goal of this scheme is to achieve greater performance by allowing other transactions to use results from other uncommitted transactions. This method is based upon the philosophy that outputs of transactions are not committed until they are used (eg. as inputs to other transactions). This method also assumes that user applications are structured as atomic operations in this manner; that associated with an application is consistency-checking software and that dependencies are recorded for recovery purposes (ie. a cross reference of files needed and updated by transaction). Conceptually there are many advantages to structuring user applications (eg. transactions or groups of transactions) as atomic operations. One problem is in

supporting this efficiently: for now transactions can be considered to operate serially. To minimize this potential waiting time, Shrivastava has proposed the ability to release transactions before commitment. This method is based upon the observation that "commitments imply dependency". In this fashion, an output of a transaction 'action' can be thought of as 'committable', that is, available for other transactions. For example, processing a transaction *B* say which relies on multiple inputs from transaction *A* (eg. updates to files), does not have to wait until all of *A*'s updates are finished; B can begin to operate as soon as any of its required inputs becomes available. This method works well when transactions run to completion. However, when a transaction aborts this may in fact invalidate incomplete results used as input to other transactions. Consequently, the net effect may be a series of aborts triggered by a single transaction abort. On the other hand, if a transaction *X*, which relies on (and received) inputs from a transaction *Z*, proceeds to commitment (ie. provides results used by another transaction) and completes before *Z* aborts, then the changes made by *Z* would have to be recovered. The transaction(s) which used results from Z, must then also be recovered, and so on. Thus, an abort triggered by one transaction could result in a series of recovery invocations on committed transactions. An example may help to illustrate this. In the figure below, transaction *V* starts at t_0 and lasts until t_2; transaction *Y* uses results from *V* at t_1; both *Y* and *V* are committable at t_2 but not committed (used by another transaction)until t_3. *Y* can be recovered up to t_3, however, after t_1 *V* will also have to be backed out.

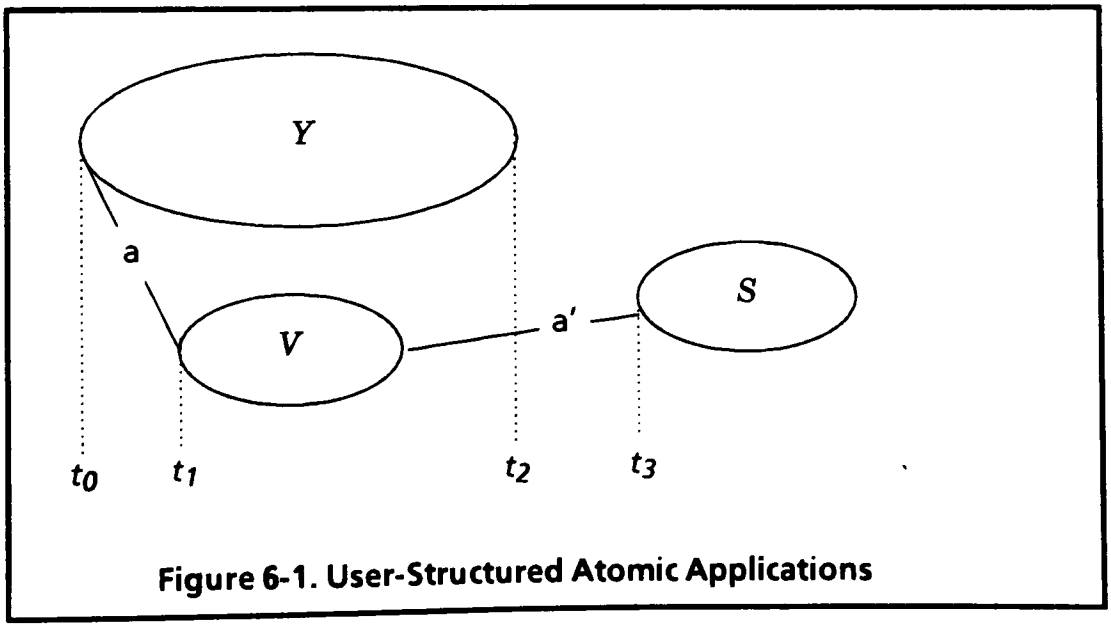

Figure 6-1. User-Structured Atomic Applications

6.4 Other Mechanisms

Bhargava [Bhargava] has investigated "optimistic" concurrency control methods. These methods are based upon the philosophy that transactions can be run concurrently until commitment; at this time they must be validated. This approach postpones inter-transaction conflict detection on the hope that conflicts will not occur frequently. To support this, a transaction passes through local and then global validation. Local validation is performed at each node in the system; global validation is done for each locally validated transaction. Both validation phases require communication with the other nodes in the system to maintain an accurate status of the collection of semi-committed and committed transactions in the distributed system.

In Pepin [Bouchet], the developers designed and implemented a novel approach to handle deadlock based upon a lock-request limit. In this scheme, a transaction would be allowed a 'maximum' number of attempts (re-requests) to acquire a lock before it was aborted. It seems possible to extend this method to allow for dynamic conditions within the file system. Such an adaptive extension could, for example, make use of internode traffic (adjust the re-request interval accordingly) as well as transaction site priority (vary the number of retries on a site basis).

Lower memory costs could make the allocation of more memory for recovery functions feasible. This may result in the ability to perform checkpointing on a more frequent basis. An additional benefit gained from more frequent checkpointing is the potential for an increase in recovery performance speed. Additional buffer memory at each node may enable each node to restrict (or confine) the damage due to errors at the node as Rohmdane suggests [Romdane].

Cheaper processors could enable the realization of distributed file system architectures utilizing redundant processors or co-processors. A redundant processor scheme would be useful for recovering from processor crashes; in such a case, the redundant processor would become the active processor. Transactions would be hidden from the effects of performing the orderly switch of processors. Alternately, a dedicated watchdog recovery co-processor could be designed to handle all recovery data management functions for the processor. This recovery co-processor may have associated with it storage for recovery data. The recovery co-processor would interact not only with its associated processor to manage the recovery information for the node, but it would also interact with the other recovery co-processors in the distributed file system. A recovery co-processor has the responsibility of not only soliciting and storing recovery information for transactions affected locally, but it must also assist in recovery on notification

from the associated main processor. Processors then must notify other participating processors in the case of recovery (or they could be required to determine it for themselves) and interrogate the co-processor for the recovery data.

7. Summary

In this paper, characteristics of fault-tolerant distributed file systems have been presented. Problems encountered within a distributed file system were first summarized and described. A set of recovery aspects to support fault tolerance in distributed file systems was defined. This topical recovery criteria was subsequently applied to two current distributed file systems implemented within different models of distributed computation: the Xerox Distributed File System and The Newcastle Connection. An independent analysis of each of the distributed file systems followed. The analysis consisted of an overview of the distributed system configuration, an overview of the file system, a general description of recovery mechanisms used within the distributed file system and an investigation of the recovery mechanisms with respect to the recovery criteria previously identified.

In order to develop a fault tolerant distributed file system, we should have detailed knowledge of the faults to be handled by the system, determine those faults which are errors (ie recoverable) and those which are disasters (ie. those for which recovery is not possible), determine the set of errors for which software mechanisms are required (ie. not handled by hardware). Software mechanisms can be compared and evaluated using the proposed recovery criteria.

It is difficult at best to formulate generalizations for approaches to fault tolerance in distributed file systems. However, a few observations regarding the framework for applying this work in establishing a fault tolerant strategy for a distributed file system are in order.

Developing a fault-tolerant strategy for distributed file system should be thought of as a process consisting of:

- determine the inclusive set of faults to be handled at some level by the system
- of this set determine those which are errors (ie. recoverable faults) and those to be considered disasters (ie. errors for which recovery is not possible)
- determine the set of errors which can be handled via hardware mechanisms
- identify software mechanisms to handle the remaining set of errors (those not handled --- hence masked--by hardware subsystems) with respect to the following criteria:
    - concurrency control
    - atomicity
    - recovery data management
    - exception handling
    - deadlock detection

It was the intent of the paper to provide an overview of recovery attributes necessary to support fault tolerance in distributed file systems. In addition to providing a general description of two implemented distributed file systems, the paper served not only to describe the implemented recovery capabilities of each within the framework of the recovery analysis but also provide alternative approaches.

This work has summarized various approaches to each of these criteria, indicated relative strengths and weaknesses of each, and has offered alternative approaches. Finally, it is hoped that this research has succeeded in providing an overview of attributes for fault tolerance within distributed file systems and serves as a constructive aid in planning and implementing fault tolerant distributed file systems.

8. Glossary

abort	used to terminate a transaction under abnormal conditions.
atomic operation	an indivisibile operation which produces no side effects; never produces *partial* results.
backout	term applied to getting out of an incomplete (canceled or aborted) transaction.
cancel	terminate a transaction under normal processing (eg. resolve deadlock etc.).
commit	point at which effects of transaction cannot be undone; effectively completes the transaction.
consistency	without errors
crashes	abnormal failures in the system resulting in loss of status and correct operation of control algorithms.
deadlock	a terminal process state initiated by an infinite wait for an event.
defer	to put off the decision to commit to a later time.
failure	characterized in termination of correct operation in a component or system; may be detected or undetected.
recoverable	attribute which signifies results after failure are defined; ability to operate in the presence of failures.
recovery point	software check point established which causes the saving of status and other necessary information; used in the event of a crash.
recovery region	concept devised by Shavristava which spans from the point where a recovery point is established to the point at which it is released.
redo	performing a transaction again, presumably after cancel or abort.
rollback	after a crash, the process of restoring state to last valid state by proceeding back through a transaction log.

transaction	a collection of operations grouped to form a specified request
transaction log	data structure containing a history of status and other necessary recovery information
two-phase lock protocol	a protocol which implements atomicity; characterized by a *intention phase* and a *commit phase*.
undo	process of restoring state of objects modified within a transaction to prior state.
update	modify objects in the distributed file system.

9. Bibliography

[Adiba]

Adiba, M., Andrafe, J. M., Update Consistency and Parallelism in Distributed Databases, *2nd International Conference on Distributed Computing Systems (IEEE)*, April 1981, pp 180-187.

[Avizienis-Kelly]

Adiba, M., Andrafe, J. M., Update Consistency and Parallelism in Distributed Databases, *2nd International Conference on Distributed Computing Systems (IEEE)*, April 1981, pp 180-187.

[Bannister]

Bannister J., Trivedi, K., Task and File Allocation in Fault-Tolerant Distributed Systems, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 103-111.

[Bhargava]

Bhargava B., Resiliency Features of the Optimistic Concurrency Control Approach for Distributed Database Systems, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 19-32.

[Birrell]

Birrell, A., Levin, R., Needham, R., Schroeder, M., Grapevine: An Exercise in Distributed Computing, *CACM* 25:4, April 1982, pp. 260-274.

[Bouchet]

Bouchet, P., Chesnais, A., Feuvre, J. M., Jomier, G., Kurinckx, A., PEPIN: An Experimental Multi-Microcomputer Data Base Management System, *2nd International Conference on Distributed Computing Systems (IEEE)*, April 1981, pp 211-217.

[Breitwieser-Kersten]

Breitwieser, H.; Kersten,U., Transaction and Catalog Management of the Distributed File Management System DISCO, *5th International Conference on Very Large Data Bases*, IEEE, N.Y., N.Y., 1979.

[Deitel]

Deitel, H. M.; An Introduction to Operating Systems; Addison-Wesley Publishing Co., Inc.; 1983.

[Devor]

Devor C., Elmasri R., Rahimi S., The Design of DDTS: A Testbed for Reliable Distributed Database Management, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982.

[Dolev]

Dolev D., Strong H. R., Distributed Commit with Bounded Waiting, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 53-60.

[Eswaran-Gray]
Eswaran, K. P., Gray, J. N., Lorie, R. A., Traiger, J. L., The Notions of Consistency and Predicate Locks in a Database System, Communications of the ACM. November 1976, Vol. 19, No. 11, pp 624-633.

[Feridun]
Feridun, A. M., Shin, K. G., A Fault Tolerant Multiprocessor System with Rollback and Recovery Capabilities, *2nd International Conference on Distributed Computing Systems (IEEE)*, April 1981, pp 283-292.

[Gertner]
GertnerI., Gordon R. L., Experiences with Exception Handling in Distributed Systems, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 144-149.

[Hammer]
Hammer M., Shipman, D., Reliability Mechanisms for SDD-1: A System for Distributed Databases, *ACM Transactions on Database Systems*, 5:4, December 1980., pp 431-466.

[Hosseini]
Hosseini, Seyed Hossein, Fault Tolerance in Distributed Computing Systems and Databases,Thesis (Ph.D), University of Iowa, 1982.

[Ho-Ramamoorthy]
Ho, Gary S., Ramamoorthy, C. V., Protocols for Deadlock Detection in Distributed Database Systems, IEEE Transactions on Software Engineering, November 1982, Vol. SE-8, No.6., pp 554-557.

[Janicki]
Janicki, R., On the Design of Concurrent Systems, *2nd International Conference on Distributed Computing Systems (IEEE)*, April 1981, pp 455-466.

[Jegado]
Jegado, M., Recoverability Aspects of a Distributed File System, *Software Practices and Experiences*, vol. 13 no. 1, January 1983.

[Jessop]
Jessop W. H., Noe J. D., Jacobson D. M., Baer J. -L., Pu C., The Eden Transaction-Based File System, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 123-125.

[Kant]
Kant K., Efficient Local Checkpointing for Software Fault Tolerance, *Operating Systems Review*, Volume 17, No. 2, April, 1983.

[Kim]
Kim K. H., Issues in the Design of Fault Tolerant Distributed Systems, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 123-125.

[Kohler]
Kohler W., Overview of Synchronization and Recovery Problems in Distributed Databases, *Proceedings of the 21st Conference on Distributed Computing*, September 1980.

[Lin]

Lin W. -T. K., Nolte J., Read Only Transactions and Two Phase Locking, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 85-93.

[Lampson-Sturgis]

Lampson, Butler, Sturgis, Howard; Crash Recovery in A Distributed Data Storage System, Xerox Technical Report, Computer Science Laboratory, Palo Alto Research Center, Palo Alto, CA., 94304.

[Lampson-Paul-Siegert]

Lampson, B. W., Paul, P. M., Siegert, H. J. (ed.), Distributed Systems - Architecture and Implementation, An Advanced Course, Lecture Notes in Computer Science, Springer-Verlag, 1981.

[Lunn]

Lunn, H., Bennet, K., A Highly Reliable Distributed Filestore Directory System, *2nd International Conference on Distributed Computing Systems (IEEE)*, April 1981, pp 299-307.

[Maeda-Yamazaki-Tanaka]

Maeda, A.; Yamazaki, I.; Tanaka, A., A Distributed File System in EPOS, *Proceedings of the 21st Conference on Distributed Computing*, September 1980.

[Martella]

Martella G., Pernic B., Schreiber F. A., Distributed Database Reliability Analysis and Evaluation, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 94-102.

[McDermid]

McDermid, J. A., Checkpointing and Error Recovery in a Distributed System, *2nd International Conference on Distributed Computing Systems (IEEE)*, April 1981, pp 271-282.

[McLellan]

McLellan, P., The Design of a Network Filing System, *CST-12-81*, Department of Computer Science, University of Edinburgh, November 1981, pp 299-307.

[Mitchell]

Mitchell, J., Dion, J., A Comparison of Two Network-Based File Servers, *CST-12-81*, CACM, 25:4, April 1982, pp 233-245.

[Mitchell-Dion]

Mitchell, J. G.; Dion, J., A Comparison of Two Network-based File Servers, *Communications of the ACM*, vol. 25 no. 4, ACM, April 1982.

[Garcia-Molina]

Garcia-Molina H., Reliability Issues For Completely Replicated Distributed Data Bases, *Proceedings of the 21st Conference on Distributed Computing*, September 1980.

[Moss]

Moss, J. E. B., Nested Transactions and Reliable Distributed Computing, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 33-39.

[Mukerji-Kieburtz]
Mukerji, J.; Kieburtz, R. B., A Kernel for Supporting a Distributed File System, *Proceedings of the 21st Conference on Distributed Computing*, September 1980.

[Panzieri-Shavristava]
Panzieri F., Shavristava S. K., Reliable Remote Calls for Distributed Unix: An Implementation Study, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 127-133.

[Ritchie-Thompson]
Ritchie, D. M., Thompson, K., The UNIX Time-Sharing System, The Bell System Technical Journal, July-August 1978, Vol. 57, No. 6. part 2. pp. 1905-1930.

[Romdhane]
Romdhane M. B., Courtois B., Error Confinement/Data Recovery in Distributed Systems, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 11-18.

[Seifert]
Seifert M. H., Structuring Distributed Software for Fault Tolerance, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 126.

[Shavristava82]
Shavristava S. K., A Dependency, Commitment and Recovery Model for Atomic Actions, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1982, pp 112-119.

[Shrivastava81]
Shrivastava, Santosh Kumar., Structuring Distributed Systems for Recoverability and Crash Resistance, *IEEE Transactions on Software Engineering*, vol. SE-7 no. 4, July 1981.

[Sinha-Natarajan]
Sinha, M. K., Natarajan, N., A Distributed Deadlock Detection Algorithm Based Upon Timestamps, *4th International Conference on Distributed Computing Systems (IEEE)*, May 1984, pp 546.

[Spector-Schwarz]
Spector, Alfred Z.; Schwarz, Peter M., Transactions: A Construct for Reliable Distributed Computing, *Operating Systems Review*, Volume 17, No. 2, April, 1983.

[Sturgis-Mitchell-Israel]
Sturgis, H.; Mitchell, J.; Israel, J., Issues in the Design and Use of a Distributed File System, *Operating Systems Review*, vol. 14 no. 3, July 1980.

[Sugihara]
Sugihara, K. Kikuno, T., Yoshida, N. Ogata, M., A Distributed Algorithm for Deadlock and Resolution, *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, IEEE, October 1984, pp. 169-176.