Abstract

The paper presents the recoverability features of a distributed file system which has been built as an extension of the UNIX* file system. The algorithm to achieve recoverability for the distributed files is discussed. The techniques implemented include backward and forward error recovery, global and incremental recovery data recording. The paper outlines how these techniques can coexist to provide the abstraction of recoverability to a user of the distributed file system.

Recoverability Aspects of a Distributed File System

By

M. Jagado

TECHNICAL REPORT SERIES

Series editor: Mr. M.J. Elphick

© 1981 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU.

*UNIX is a Trademark of Bell Laboratories.
JEGADO, Maurice

Recoverability aspects of a distributed file system.


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1981.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 160)

Abstract

The paper presents the recoverability features of a distributed file system which has been built as an extension of the UNIX* file system. The algorithm to achieve recoverability for the distributed files is discussed. The techniques implemented include: backward and forward error recovery, global and incremental recovery data recording. The paper outlines how these techniques can coexist to provide the abstraction of recoverability to a user of the distributed file system.

*UNIX is a Trademark of Bell Laboratories.
1. Introduction

The aim of the distributed file system described in this paper is to incorporate in the design of a real system some of the ideas on reliability and fault tolerance developed by a research project at the University of Newcastle upon Tyne, UK which is investigating the reliability and integrity of distributed computing systems [3,5,10,13]. Recent work documented in [13] and the availability of a Unix local network forms the basis for an experiment in distributed systems.

Schemes for dealing with the damage that has been assessed as existing when an error is detected are usually classified into backward and forward recovery techniques. Backward error recovery depends on the provision of recovery points [10], i.e. a means by which the state of a process can be recorded and later reinstated if required. The notion of a recoverable file system is used to mean that the state transformations performed on files can be recovered from by invoking the backward recovery capability of the system, providing the abstraction of recoverability to a user of the file system.

Although mechanisms to provide recoverability for local objects are well understood [1,6,9,14], mechanisms to provide recoverability in a distributed environment are more complex. In data bases the unit of recovery is considered to be a transaction [7]. However, the recovery capability of a distributed data base management system is generally not provided to support software fault tolerance but rather to recover a transaction when an access conflict is detected, or when system crashes occur for example.

In contrast, the Distributed File System (DFS) described here can be used to support software fault tolerance, for instance, for use with the recovery block scheme [11]. The DFS provides its users with the ability to establish nested recovery points. The grain of recovery is arbitrary and fixed by the user.

The DFS provides some protection against crashes. A crash is an undesired event which causes the state of volatile storage in one machine to be reset to some standard value [8]. The measures of crash resistance incorporated in the DFS are discussed.

The paper is arranged as follows: in the next section we describe briefly the standard Unix file system and state our objectives. The third section presents the specifications of the DFS. It introduces some preliminary definitions and explains the concurrency control mechanism. The global architecture of the system is described in section four and we examine in turn the detailed implementation of the system components with particular emphasis on the implementation of recoverability. The last section addresses some aspects regarding exception handling and crash resistance.

2. Objectives

Before stating our objectives, we give an overview of some of the operations supported by the standard Unix file manager [12]. In
response to an operation: open(filename, mode) the Unix File Manager (UFM) returns a file descriptor which has associated with it a read/write pointer. The role of the file descriptor is to identify the file in subsequent operations such as read, write, and close. The UFM simply considers a file to be a contiguous array of bytes. The read/write pointer defines the position from which a read or write operation begins in the array.

Our objective is to build an extension [3] of the UFM such that (i) it handles the distributed nature of the files, and (ii) it provides the abstraction of recoverability for the distributed files to the user processes.

The DFS interface is designed to be as similar as possible to the UFM interface. A DFS user accesses distributed files located on different sites (remote or local) in the same manner as local files can be accessed through the UFM. However, the geographical distribution of the files is visible to the user. A user has to specify when a particular named file is in fact resident on a remote machine. In fig.1 we consider only a subset of the filing operations implemented since they are the most meaningful regarding recoverability.

dopen(machinename!fname, mode) returns fd: int
  mode = (Read, Write, R/W)
dclose(fd)
dwrite(fd, buffer, nbytes) returns nwritten: int

Fig. 1. Distributed Recoverable File Manager interface

The operation listed in fig.1 are prefixed by d (distributed). The exceptions that may be returned are not considered. The notation fd is a shorthand for file descriptor while fname stands for local filename. Those operations, constituting the DFS interface, are supported by a file manager which is called the Distributed Recoverable File Manager (DRFM). A file can be opened several times in possibly different modes by the same user. However, for the sake of simplicity of this presentation, we assume that a file can be opened only once within a transaction (section 3).

3. Specifications

3.1. Recovery structure

The terms establish recovery point (erp), discard recovery point (drp), and recovery region have the same meaning as in [3]. Recovery points can be nested and lead to nested recovery regions.

We define a transaction as a user program delimited by 'begin transaction' and 'end transaction' operations. A transaction consists of an arbitrary number of possibly nested recovery regions. A particular case occurs when within a transaction, no recovery point is established. This could correspond to a situation in which a user does not want to make use of recoverability, for example, for cost reasons. In
the rest of this paper we assume for simplicity that the transactions are of the form illustrated in the following figure.

```
beginr(username)
   erp()-------------------------------------
      
   erp()--
      .
      .
      .
      .
      .
      .
      .
      .
      .
   drp()--
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
   drp()-------------------------------------
endr()
```

**Fig. 2. Recovery structure in a user program**

The 'erp' operation returns a recovery point number (rpn) which permits identification of the recovery point. A user can invoke recovery to any active recovery point by the 'recover(rp)' operation. The recovery point rpn remains active after the 'recover(rp)' operation has been performed. The 'drp' operation discards the most recently established recovery point. We define commitment as taking place when the outermost recovery point is discarded, since after that time the state transformations performed on files cannot be recovered from by invoking the backward recovery capability of the system.

It is worth noting that the recovery discipline adopted does not enforce any particular structure with respect to the filing operations (Fig. 1). Constraints such as "a file must be closed within the recovery region in which it was opened" are not imposed. A user can perform filing operations independently from the recovery structure.

### 3.2. Concurrency control

The concurrency control of the DFS, in the presence of multiple users competing for the public files, relies on the use of locking to construct planned atomic activities [10]. The locks granted within a transaction are held until commitment to prevent the occurrence of the so-called "domino-effect". All locks are exclusive and the granularity of lock is the entire file. The reason for choosing such a granularity is mainly due to the fact that the UFM does not impose a structure on the contents of a file.

The locking mechanism is decentralized so that the lock of a file is maintained at the site on which the file resides. The locking operations are not made visible to the user. The DFS implements the locks internally. When a user performs a 'dopen' operation on a file, the DFS tries to grant the lock. If the lock cannot be granted an exception code: "lock not granted" is returned. In this case, the user can wait and retry, or invoke recovery explicitly. Since the locking operations are themselves recoverable, a deadlock situation can be solved by
recovery. However, no automatic provision for deadlock has been incorporated in the implemented system.

The locking strategy presented has both advantages and disadvantages, a main advantage being its simplicity. Alternative schemes are possible based for instance on unplanned recovery control [15]. The reader interested in the influence of different locking strategies on the degree of concurrency can refer to [4]. Note that the scheme presented enables a transaction to begin by opening all the files which might be required, thereby obtaining all the locks which might be needed during the execution of the transaction. Therefore a user can either follow this policy or choose to acquire the locks during the execution as needed.

3.3. Crash resistance

The DFS incorporates some protection against crashes. It ensures that any processor crash occurring in the distributed system before commitment leaves the system in a consistent state such that no file is left partly updated. The specific problems regarding crash recovery during the commitment operation itself are not taken into account.

The crash resistance feature of the DFS is achieved by keeping all the state modifications performed before commitment in volatile storage. Memory and temporary files are examples of such volatile storage, temporary files being deleted on system restart. Should a crash occur before commitment, the state of the files modified by a transaction is restored to the state that existed prior to their modification. In other words, the first recovery point established is made crash resistant.

A user may decide to terminate a transaction before it is completed. For this purpose, we provide an additional operation: 'abort' which results in the crash recovery procedure to be invoked.

4. Implementation

The implementation of the DFS is distributed. On each machine a unique local file server process serves the requests related to the files residing on that machine. This process is implemented on the form of a Local File Manager (LFM) which is built on the top of the UFM. The interface provided by the LFM is detailed further below. The set of active entities or processes present in the distributed system comprises the local file servers and the user processes. Note that a local file server process has an infinite life. The relationship between the components of the DFS are depicted in fig.3.
The DRFM and the LFM communicate via a message passing system. However, we can regard this communication as equivalent to remote procedure calls - the DRFM can call the procedures provided by the LFM.

Figure 3 shows that local files are accessed by the same mechanism as remote files, i.e via the LFM of the local machine. This feature has facilitated system testing. Most of the tests have been performed on a single machine, following which the decentralization to a physically distributed environment has not caused particular difficulties.

It should be noted that the Unix operating system and the PDP/11 do not support backward error recovery as a basic mechanism. Thus only the files handled by the DFS are recoverable; recovery for other objects local to a process is not provided.

In order to limit communication overhead between the DRFM and the LFM, a paging mechanism is implemented. With each file descriptor allocated to a user, the corresponding current page is maintained at the DRFM level while the LFM performs block transfers. A subset of the filing operations provided by the LFM is given below.
lopen(lfname, mode) returns fd: int
lclose(fd)
lwriteblock(fd, buffer, blockno)
Ireadblock(fd, #buffer, blockno)
lcreate(lfname, protection) returns fd: int
lunlink(lfname)

Fig. 4. Local File Manager Interface

The interface shown in fig.4 forms an internal interface between the DRFM and the LFM and is not visible to the user. The operations are prefixed by l which stands for local. The file descriptor (fd) returned in response to the 'lopen' operation is not the fd returned to the user. The DFS performs successive file descriptor mappings which are not considered in this paper.

We now examine in turn the structure of the DRFM and the LFM with particular emphasis on the implementation of recoverability.

4.1. Structure of the Distributed Recoverable File Manager

The DRFM is the "controller" of a transaction. It coordinates the LFM's whose services are requested during the execution of a transaction. The question which arises regarding recoverability is: How can recoverability be achieved for the distributed files? The scheme presented by Shrivastava [13] gives a solution to this problem by using the recoverability capabilities of the remote objects managers in a manner that models the inclusive recovery scheme [3].

The DFS follows this strategy. When a user establishes a recovery point, the DRFM broadcasts an 'establish recovery point' message to the LFM's which have taken part so far in the execution of the transaction. On reception of such a message, the LFM establishes its own recovery point thereby using its own recovery capabilities. Similarly when a user performs a 'recover' or 'drp' operation a corresponding 'recover', 'drp' message is broadcast.

As all the operations provided on the LFM interface are made recoverable, the DRFM does not have to maintain recovery information for the objects accessed through that interface. This greatly simplifies the implementation of recoverability in the DRFM since only the recovery information related to the objects maintained at the DRFM level has to be recorded.

We define a session as being the logical channel which links a DRFM to the LFM of a machine. A session is opened with a LFM when the machine name is referenced in the user program (fig.1). Only the first reference generates the opening of a session. The sessions are closed when the 'end transaction' operation is executed. When a session is opened, the DRFM sends for each active recovery point of the user, a corresponding 'erp' message to the LFM. The DRFM records incrementally the history of the sessions in a recovery cache type structure termed the session
cache.

The DRFM also maintains some information related to each file descriptor allocated to the user (current page, read/write pointer, local file server name...). This information represents a small amount of data and is recorded globally when a recovery point is established in a recovery cache type structure termed the file descriptor cache. However, the contents of the current pages are not recorded. When a recovery point is established the current pages which have been written are flushed onto the disk storage by calling the 'lwriteblock' procedure of the LFM. When recovery is invoked, the status of the current pages in the DRFM are set to: "toread", meaning that the pages will have to be read by calling the 'lreadblock' procedure of the LFM. It is worth noting that although the concrete state thus restored differs from the prior concrete state it satisfies the same abstraction [3,5].

When the recovery data is recorded incrementally, the standard algorithms of an object manager have to handle the recovery data during their execution. However this technique may be necessary to limit the amount of recovery data stored. The advantage of recording the recovery data globally versus incrementally resides in that the former technique enables a clear distinction between the standard algorithms of an object manager and the recovery algorithms (erp, drp, recover, abort).

In fact these techniques can coexist to make the recovery implementation more efficient. The DFS is an example of such a use of a combination of techniques. The LFM provides the abstraction of recoverability mainly by recording the recovery data incrementally as explained in the following.

4.2. Structure of the Local File Manager

A local file server process executes the recovery and filing operations supported by the LFM on request from the DRFMs. The LFM maintains an execution context for each session opened with a DRFM consisting of two instances of a recovery cache type structure termed respectively the undo cache and the commit cache. The undo cache records the operations to be undone when a user invokes recovery. The commit cache records the identification of the files which are not in a committed state. The commit cache does not contain redundant data.

We now examine how recoverability is achieved for some of the filing operations supported by the LFM. The creation and deletion of files by the operations 'lcreat' and 'lunlink' are mentioned for the sake of illustration and not considered in further detail.

1) lwritetblock: When this operation is performed, the original file is copied into a temporary file if no copy has been made previously. This copy is made in order to provide crash resistance (section 3) and is maintained as the up-to-date version of the file, that is, all updates are performed on this copy. When the copy is performed the file identification is recorded in the commit cache and the code {copy} in the undo cache.
The copy made would be sufficient for recovery purposes if recovery regions were not nested. As recovery regions can be nested the LFM, to update a block, proceeds by: (a) reading the contents of the block from the copy; (b) caching it in the undo cache of the session's execution context; (c) updating the block in place on the copy. To avoid redundancy of recovery data, a block is not cached if it has already been cached within the current recovery region. In order to limit the amount of recovery data held in the undo cache, the prior contents of the block are not recorded directly in the undo cache but in a temporary file of the LFM called block-pool; the undo cache retains only the code {lwriteblock}. The parameters associated with this code are: fd, blockno written, blockno allocated from the block-pool. The block-pool file is a shared data structure maintained by a monitor.

ii) lopen: This operation consists of two primitives operations which are (a) locking the file; (b) executing the open operation of the UFM. Hence the codes {lock, open} with appropriate parameters are recorded in the undo cache. If recovery is invoked, the file is closed and unlocked.

iii) lclose: This operation consists of (a) executing the close operation of the UFM; (b) unlocking the file. Since the unlock operation must be delayed until commitment, the unlocking is not performed but the file identification is recorded in the commit cache while the code {close} is recorded in the undo cache. If recovery is invoked the file is reopened. Note that the execution of the open and close operations of the UFM are performed on the copy of the file if a copy has been made.

Maintaining a copy of a file to make the 'lwriteblock' operation recoverable and crash resistant may appear an inefficient solution. An alternative scheme would have been to maintain only a copy of the blocks which have been written. A file block mapping [2] could then associate to a blockno, either the original block if the block has not been modified, or the copy of that block if an update has been performed. However, apart from the simplicity of implementation, the first solution has the following advantages:

- It optimises the normal execution of the 'lreadblock' operation. No file block mapping has to be performed since the copy contains the up-to-date version of the file.

- It enables us to adopt the same solution to make the 'lcreat' and 'lnlink' operations recoverable and crash resistant. The basic idea is to work on temporary files and not to perform state modifications in stable storage before commitment. For instance, a 'lcreat' operation results in a copy being created.

To summarize, the commit cache contains the identification of the files which are not in a committed state, while the undo cache contains the operation codes {lock, open, close, copy, writeblock} as necessary. A recovery cache is organised as a stack, which is subdivided into regions separated by barriers [1]. The algorithm executed by the LFM when recovery is invoked by a user process is given in fig.5.
1) Recover(rpn) =

2) Remove the barriers from the current recovery region to region rpn in the commit cache and the undo cache respectively.

3) Process the undo cache as follows:

   while (pop(operation) != barrier) {
     switch(operation) {
       case lock: unlock.
       case open: close.
       case close: open.
       case copy: delete copy.
       case wblock: write the prior value of the block. free(blockno allocated from block-pool).
     }
   }

4) Process the commit cache by removing the entries until the barrier is encountered.

5) Set current recovery region to rpn.

   **Fig. 5. Recovery Algorithm of the Local File Manager**

   At commitment time, the entries in the undo cache are discarded and the commit cache is processed as follows. For each file identification recorded in the commit cache, the LFM determines what the state of the file after commitment should be. It then performs the necessary commit operations to effect the state transition from the actual state of the file to the post commitment state in stable storage. For instance, the commit operations related to a file which has been successively opened, updated, and closed are: rename the copy to be the original file name, delete the original file, free the blocks allocated from block-pool, and unlock the file.

4.3. Synthesis

   The concrete representation of the DFS is composed of a hierarchy of modules which have been termed the DRFM, LFM, UFM (fig.3). In the following diagram, we show part of the interactions involved between those modules when a 'dopen' operation is performed in a user program.
5. Exception handling – crash resistance

The purpose of this section is to answer the question: What happens if the occurrence of an exception is detected in the DRFM or the LFM? First of all, we stress the fact that neither the DRFM nor the LFM makes use of backward error recovery for itself since the Unix operating system does not support backward error recovery as a basic mechanism (section 4). When the occurrence of an exception is detected in one of these modules, an explicitly programmed handler is executed. The role of this handler is to attempt to mask the occurrence of the exception and to proceed with the execution if the masking is successful; the module being then strongly tolerant to the occurrence of the exception [5]. If the masking is unsuccessful, the handler attempts to restore the consistency of the module by forward error recovery and to propagate the exception or the failure exception to the calling module. The attempt to restore the consistency may be unsuccessful, the module remaining in an inconsistent state. In that case, the exception is converted into a crash event and the abort procedure of the module is invoked. Note that we do not specifically consider communications failures. Those failures result in exceptions being raised by the lower level message passing system (fig.3) and are treated in the same way.

In the following, we present some characteristics of the crash recovery protocol without giving the complete specification. When an exception within the LFM is converted into a crash event, the LFM executes the abort procedure which consists of cleaning the session’s execution context and returns an 'abort' answer to the DRFM. On reception of such a message the DRFM triggers the abort procedure which consists of cleaning up the transaction locally and broadcasting an 'abort' message to the other local file servers recorded in the session cache. On reception of an 'abort' message the LFM executes the abort procedure if the session is not already aborted. Should a local file server crash, the exception raised by the message passing system in the DRFM is converted into a crash and the abort procedure triggered. The protocol implemented is fairly simple but does not cope with exceptions raised during commitment.

6. Conclusion

In this paper, we have discussed the recovery features of a distributed file system, detailing successively the structure of the system components. It has been shown that a combination of techniques such as global and incremental recovery data recording can make the
implementation of recoverability more efficient. The fact that the system is geographically distributed has not complicated the design since local files are accessed in the same manner as remote files. The access is made through the message passing system interface which "hides" the inter-machine dependencies. The system has been written in the high level language "C" and is currently running on two PDP11/45. It is to be extended to a broader local network in the near feature. The overall structure of the system is simple and in this respect sticks to the objectives of development of the first version.

A number of extensions are possible: (i) to investigate other concurrency control mechanisms; (ii) to improve the basic set of facilities provided in order to give to a user the same facilities as the Unix file system and to measure the actual overheads incurred.

Acknowledgments

A special debt of gratitude is due to P.A. Lee who launched the project by an internal communication and who continuously helped the author during the design and implementation. The author remains grateful to F. Cristian for his valuable advice in modular design and his help in the production of this paper. Thanks are also due to F. Panzieri, B. Randell, S.K. Shrivastava, and G. Wood for their useful comments.

This work has been carried out with the support of the Institut National de Recherche en Informatique et Automatique of France and the Science Research Council of Great Britain.

References


