Abstract:

This paper describes the implementation of a stable storage system which converts several fallible disk stores into a reliable device for storing data. It provides reliable reading and writing of data in a distributed UNIX* environment in spite of transient I/O faults, decay of physical storage devices and processor crashes. The implementation makes available to UNIX users a convenient way of using the facilities of a stable storage system by providing the abstraction of stable files and by maintaining the standard UNIX system call interface. It systematically handles abnormal situations by separating normal and exceptional processing in both the system description and implementation. This is achieved through the use of a fault tolerance design notation for the description of the system and the implementation of that notation using an exception handling package.

*UNIX is a Trademark of Bell Laboratories.

A Reliable Stable Storage System for UNIX

By

J.A. Anyanwu

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick

Number 191
January, 1984

© 1984 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, England.
ANYANWU, Josephine Ada

A reliable stable storage system for UNIX.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1984.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 191.)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 001.6442
U.D.C. 519.687 681.322.02

Suggested keywords
PROCESSOR CRASHES
STABLE STORAGE
UNIX

Abstract

This paper describes the implementation of a stable storage system which converts several fallible disk stores into a reliable device for storing data. It provides reliable reading and writing of data in a distributed UNIX* environment in spite of transient I/O faults, decay of physical storage devices and processor crashes. The implementation makes available to UNIX users a convenient way of using the facilities of a stable storage system by providing the abstraction of stable files and by maintaining the standard UNIX system call interface. It systematically handles abnormal situations by separating normal and exceptional processing in both the system description and implementation. This is achieved through the use of a fault tolerance design notation for the description of the system and the implementation of that notation using an exception handling package.

* UNIX is a Trademark of Bell Laboratories

About the author

Mrs. J.A. Anyanwu is now in her third year as a Ph.D. student at the Computing Laboratory of the University of Newcastle upon Tyne.
A Reliable Stable Storage System for UNIX

J. A. Anyanwu
Computing Laboratory
University of Newcastle Upon Tyne
NE1 7RU, U. K.

ABSTRACT

This paper describes the implementation of a stable storage system which converts several fallible disk stores into a reliable device for storing data. It provides reliable reading and writing of data in a distributed UNIX environment in spite of transient I/O faults, decay of physical storage devices and processor crashes. The implementation makes available to UNIX users a convenient way of using the facilities of a stable storage system by providing the abstraction of stable files and by maintaining the standard UNIX system call interface. It systematically handles abnormal situations by separating normal and exceptional processing in both the system description and implementation. This is achieved through the use of a fault tolerance design notation for the description of the system and the implementation of that notation using an exception handling package.

1. Introduction

The problems of tolerating faults in a distributed system is made more difficult by individual site crashes which may leave information stored locally in each processor or globally within the distributed system in an inconsistent state. Unreliable disk storage devices, which can suffer from physical decays also threaten the reliability of stored data. The use of a so-called stable storage system [1] is now accepted as one of the ways of maintaining the internal consistency of stored data in the presence of hardware failures. Such a system makes use of replicated physical hardware and carefully designed fault tolerance strategies in order to provide an abstract store for which the probability of failure can be regarded as negligible. This implementation provides a reliable repository for data which converts fallible disk storage into stable disks by tolerating anticipated faults that affect disk storage. It also provides crash recovery facilities for the data stored on such disks. These stable disks have the same actions as ordinary disk storage but with the property that no anticipated adverse

*UNIX is a Trademark of Bell Laboratories
events occur. As yet, there has been only limited experience with the implementation, use and evaluation of stable storage systems. Our implementation is partially an exercise in that direction. The implementation described here provides a simple way of providing the facilities of a stable storage system as an independent facility which is not embedded in an operating system. It addresses the problems of providing fault tolerance facilities in response to:

(i) processor crashes

(ii) random decays of physical storage devices, and

(iii) transient input/output faults

It tolerates disk crashes by the use of two disks which are not "fault-related" such that each acts as a backup for the other. This implementation differs from other stable storage systems in its provision of the abstraction of UNIX-like stable files rather than simple disk storage areas and in its systematic handling of abnormal situations arising from the use of disk storage devices. We are not aware of any other stable storage implementation that provides the abstraction of stable files which allows atomic reading and writing of variable length stable objects. In contrast to our approach most stable storage systems provide fixed size stable pages [1,2,3]. In these systems, operations of the stable storage system are atomic only if they involve just a single read/write operation on a fixed size page. However, users often have the need to write blocks of data of varying sizes atomically. Our implementation guarantees atomic read/write of variable length objects and our provision of the abstraction of stable files instead of stable pages gives the stable storage system an interface which is fully familiar to the UNIX user.

The stable storage system has been implemented on a UNIX based distributed system called UNIX United [4] which consists of a number of PDP/11 computer systems connected by the Cambridge Ring local area network. The facilities described in this paper are part of a prototype reliability subsystem associated with the Newcastle Connection software of our UNIX United system. Our implementation has been inspired by the fault tolerance system design methodology developed by Cristian [5]. An exception handling software package described by Lee [6] is used for handling detected exception occurrences.

In the following section we give an overview of the system of which the stable storage system is a part. In section 3 we describe the reliability issues which the stable storage system addresses. Section 4 presents the fault tolerance design notation which is used to separate the standard and exceptional processing performed by a software component. It also describes the exception handling package which is used to implement that notation. The stable storage implementation is presented in section 5 and section 6 contains some performance measurements. In section 7 we discuss an alternate approach for implementing stable files in UNIX. Section 8 provides some concluding remarks. An example of how the fault tolerance design notation was implemented using the exception handling package is given in the appendix.
2. Overview of the System

This section gives a brief overview of the UNIX United distributed system [4] and describes how the stable storage system has been integrated into it. A UNIX United system is usually made up of a (possibly large) set of standard UNIX systems interconnected by a communication network. The naming scheme used by such a system joins together the naming structures of the individual UNIX systems into a single naming tree such that these component systems appear as directories in that naming tree. This enables a legitimate user on any of the UNIX systems to access files or devices of any other component system within the UNIX United framework as though these devices were part of the user's own system. Depending on the need of a computing environment, some of the computers in a UNIX United system could be used as "stable servers" which provide the services of a stable storage system. (Each computer could in fact provide both stable and ordinary disk storage facilities). Fig. 1 shows a possible position of a stable storage system in a UNIX United naming tree containing five UNIX systems U1, U2, U3, U4, U5.

```
(base)
  /
 /  
U1  U4
  /
 /  
U3  U5(system providing stable storage)
  /
 /  
sf  sf
 (stable file)
```

Fig. 1 An Example of a UNIX United Naming Tree

Following normal UNIX naming convention, names starting with "/" indicate that the name starts at a root directory and the symbol "/" is used to indicate a parent directory. The root for any process is at the UNIX system in which the user "logged in" unless the process changes its root with a "change root" command. From fig. 1 it follows that a user process on UNIX system U4 can access files in the stable storage system as "/U5/sf". A user on U2 can access these files as "/../U4/U5/sf". This implies that a user can access the stable storage system using standard UNIX system calls with the file-name being interpreted as a route through a naming tree, each element specifying the next branch to be taken. If a leaf corresponding to a stable file is reached, the appropriate stable operations will be invoked rather than the normal UNIX operations.

The UNIX United naming scheme is implemented by means of communication links and the inclusion of a software subsystem called the Newcastle Connection in each of the individual UNIX systems. This software subsystem is located between the UNIX kernel and the rest of the operating system and user programs. It intercepts system calls and determines
which of the calls are local and which are for remote UNIX systems. It also incorporates UNIX servers which accept calls that have been redirected to it from other systems. Communication between the Connection layers in the individual UNIX systems is performed by a remote procedure call mechanism [7].

The Newcastle Connection software sends any file access requests to the UNIX server in the appropriate UNIX system. In our present prototype implementation of stable storage for UNIX, it is the UNIX server in each system that distinguishes between ordinary files and stable files and invokes the corresponding operations. Once the stable storage system is invoked by the UNIX server, it assumes that the files to be accessed are legitimate stable files. If those files are not in fact stable files, the invocation will terminate abnormally by raising exceptions. This UNIX server is presently invoked only for remote file accesses. Thus each machine which contains our prototype stable storage system is regarded as functioning just as a "server machine" for the other component UNIX systems. It is not intended to be used by local user processes. An alternative method of incorporating the stable storage mechanism (we have developed) into a component UNIX system would allow processes in that system, as well as remote processes, to use stable storage. This would involve local as well as remote file accesses being interpreted. A fully general implementation would allow each component UNIX system to provide (local and remote) users with a mixture of conventional and stable storage. However, our present aim is to investigate the design of stable storage systems themselves, and our present method of incorporating the mechanism into a UNIX United system suffices for these purposes.

The stable storage system sits on top of the UNIX kernel and is regarded as a user process by the kernel. The relationship between the stable storage system, the Connection subsystem and the UNIX kernel is shown schematically in fig. 2.

```
user programs, non resident UNIX software
--------------
Newcastle Connection ---------- remote procedure call
                          |stable storage
--------------
UNIX kernel

--------------
user programs, non resident UNIX software
--------------
Newcastle Connection
--------------
stable storage
--------------
UNIX kernel

UNIX1               UNIX2

Figure 2: Software Subsystem Relationship
```
3. Reliability Issues

Various adverse environment activities can prevent a disk storage system from providing reliable service. Our physical disk storage is regarded as consisting of contiguous blocks of data. Access to the disk storage is provided by two functions: READ and WRITE. The actions performed by these operations are as follows: a successful read operation would read a block of data from a disk, and return the number of data bytes read. A successful write operation would change the existing disk state by writing the desired block of data on the specified disk. Unfortunately, due to processor crashes and physical decays of a disk storage system, a read/write operation will not always succeed.

(i) By a processor crash we mean any event which causes a processor to lose the contents of its main store.

(ii) We will say that there is a decay at an address on disk if we can not read from or write to that address. We will say that there is a transient decay, td, at some address, if initial attempts to read from and write to that address fail but a successful read/write operation is achieved within a predefined number of read/write retries.

In addition to such processor crashes and decays there are other abnormal input/output situations to deal with. Most of the troublesome problems are associated with the write operation. For disks without read-after-write capability, there is no assurance that the data has been correctly written. For example

(i) A write operation which returns successfully without changing the state of the disk is often not detected.

(ii) There is the problem of a write operation which writes to the wrong address.

(iii) A write operation which writes the wrong data to the disk also presents a problem.

(iv) There is also the possibility of a read/write operation which signals failure when the disk is not faulty. This failure is often due to transient I/O faults such as transient decays on disk.

These are some of the issues which our implementation addresses to provide reliable storage of data.

4. Concepts and Notation

The need to provide effective fault tolerance [8] led to our separation of the standard and exceptional processing performed by a software component in both the system description and implementation. We use a fault tolerance notation to describe the stable storage system and an exception handling software package to implement that notation. This section describes some of the concepts used in this paper, the fault
tolerance notation and its implementation.

Our concept of an exception occurrence is similar to those of Melliar Smith and Randell [9] and Cristian [10]. The intended service which a procedure or component is to provide consists of making some internal state transition. This intended service can be specified by a binary relation $\text{INV}$ over the initial and final states. If the final internal state $s'$ is the intended outcome of activating a component $c$ in the initial state $s$, then we say that

$$(s', s) \in \text{INV}$$

An exception is said to have occurred if a procedure, when started in an initial state $s'$, terminates in some final state $s$, such that

$$(s', s) \not\in \text{INV}.$$  

A procedure either terminates normally or it terminates by signalling an exception. Once an exception is signalled, a handler associated with that exception is invoked, if such a handler was provided. If no handler was provided, the exception is propagated to the the enclosing exception context, and up the call-chain until either a handler for the exception is found or the highest exception context is reached. The highest context will either handle the named exception, or it will indicate to its caller the failure of the software component by converting the exception into a "failure" exception.

4.1. Notation

The notation which we use to describe the stable storage system is an adaptation of the notation used by Cristian [5]. Suppose $c$ is a command or procedure which may signal a set $E$ of exceptions. Then one can give the declaration:

\begin{verbatim}
proc c SIGNALS E  (PD)
\end{verbatim}

This is the procedure declaration construct, PD. It simply indicates that $c$ has two exit points: a standard one and an exceptional one. If $c$ is invoked in an initial state $s'$ and terminates in a final state $s$ such that $(s', s) \not\in \text{INV}$ then one can say that $c$ terminates using the normal exit point otherwise $c$ will be said to terminate using the exceptional exit point and an exception in $E$ will be signalled. Only exceptions which appear after a "SIGNALS" clause are visible outside a procedure. All other exceptions are internal exceptions which are detected and handled within the procedure itself.

The next construct is the exceptional continuation construct, EX. Let $H$ be a set of handlers associated with $E$. The construct

\begin{verbatim}
c[E::H]  (EX)
\end{verbatim}

says that if any invocation of $c$ detects an exception in $E$, then the standard continuation of $c$ is to be replaced by an exceptional continuation by invoking a handler in $H$. A handler in $H$ may be a (possibly
empty) sequence of operations and may itself signal an exception.

The R construct \( (N)c[\text{OTHERS}::\text{OH}; \text{E}::\text{H}] \) (R)

will be used as an abbreviation of the n-depth repetition
\( c[\text{OTHERS}::\text{OH}; \text{E}::c[\ldots c[\text{OTHERS}::\text{OH}; \text{E}::\text{H}].\ldots]] \)

The semantics of the R (repetition) construct are as follows: Suppose there is a special exception called OTHERS which is an element of \( E \), and a handler \( \text{OH} \) in \( H \), which is associated with OTHERS. Let \( n \) be an integer constant with value \( n \geq 1 \). Then if any invocation of \( c \) signals the exception OTHERS, the handler \( \text{OH} \) is immediately called. However, if the invocation of \( c \) signals an exception in \( E \) which is not OTHERS, then the handler action will consist of invoking \( c \) again. This repetition is continued until \( n \) successive \( c \) invocations persistently signal exceptions in \( E \), at which point a handler in \( H \) associated with that exception is invoked. Otherwise, if for some \( i, 1 \leq i \leq n \), an invocation of \( c \) terminates normally, then (R) terminates without further handler action being initiated and hence, without further retries. Although OTHERS is an element of \( E \), it demands special treatment when detected, namely, that the repetition loop be exited. This enables us to deal with exceptions which, when detected, indicate that further retries will be futile and that a handler is to be invoked without completing the repetition loop.

4.2. Implementation of Notation

The fault tolerance notation was implemented using the exception software package described by Lee [6]. This package is actually a set of macros for the C language. The basic structure of a program using the exception package is shown in Fig. 3.

```
BEGIN  /* beginning of an exception context*/
--------
if condition-true then  /*normal code*/
exc-raise(<exception name>)
--------
EXCEPTION  /* beginning of exception handlers*/
WHEN(<exception name>)
--------
WHEN(<exception name>)
--------
END  /* end of exception context*/
```

Fig. 3

The notation \([E::H]\) establishes an exception context and is implemented by the BEGIN and END primitives of the exception package. The EXCEPTION and END clauses of the exception package indicate the beginning and ending respectively of the handlers in \( H \) which are associated with an
exception context. After executing BEGIN, and if no exceptions are raised, the exception context is exited and control passes to the code following the END statement. If an exception is raised between the BEGIN and EXCEPTION statements, control passes to the appropriate WHEN clause, and the associated handling code is executed, at the end of which control passes to the END statement. The interested reader is referred to [6], which describes the package fully.

5. Stable Storage Implementation

Several unreliable physical disk storage devices which, we assume, are characterised by inherent unreliability due to electrical and mechanical interferences can, be converted into a reliable device for storing data by the implementation of a stable storage system. The stable storage system we have implemented is intended to provide the abstraction of reliable virtual devices with the property that transient input/output faults and decays are not visible to the user. This is achieved by implementing stable files and providing reliable atomic variable length read/write operations for accessing these files instead of the usual read/write operations for a disk storage device whose atomicity is guaranteed only if they operate on fixed size pages.

The operations that constitute the interface to the stable storage system are organised as a set of server processes. These server processes are structured as two successive levels of abstraction, each level eliminating some set of undesired events associated with the disk storage. The first layer, called the transient layer, masks transient I/O faults. The second layer is the stable layer which uses the virtual devices produced by the transient layer to construct a better behaved set of devices by providing facilities for tolerating decays and crashes. The following section presents the basic information structure used by the stable storage system, namely stable files, and the stable operations which use these files.

5.1. Stable Files

A stable file looks to the user like an ordinary UNIX file. It is physically represented by an ordered pair of UNIX files held on two different disk storage devices. (The pair of files could have been chosen from the same disk storage device as long as care is taken to ensure that the files are not "decay related"). A stable file in our environment is read and written using standard UNIX system calls. A read operation reads from the first file and if that read operation is not successful, it reads from the second file. A write operation writes to each of the pair of files. The details of these operations are given in the section on "stable operations".

The problem we addressed was to provide the user with atomic variable length read/write operations which can be used to access data blocks of varying sizes and which incorporate crash recovery facilities for these variable length objects. Disk storage devices provide a weak atomicity property for fixed size pages such that a write operation to these pages is either written completely or not at all (except if a failure occurs while the disk’s write head is turned on). If such a
failure occurs, the data on the disk will be detectably bad and error
detecting codes which are written with every disk page will reveal this
fault when the page is read. Such physical hardware does not guarantee
atomicity of read/write operations on variable length blocks of data.
To provide this facility we implemented the abstraction of stable files,
and provided a means of crash recovery for these files. It is the means
of crash recovery that dictated the structure of the stable file.

In a fixed page environment, a stable page is usually represented
by two fixed size pages. If a failure should occur during a write
operation to the pair of pages, we assume that the pages will be in one
of the following states:

(i) Both pages contain valid data (even though the data might be
different).

(ii) One of the pages is detectably bad.

In case (i) if the contents of the two pages are different, crash
recovery would consist of copying one of the pages to the other. The
preference is usually to copy the first page to the other page so that
the most recent update is reflected. In case (ii) the good page is
copied to the detectably bad page.

Unlike the fixed page situation, a variable length write operation
can be interrupted by a crash resulting in only a part of an object
being written. It is usually not known which object was being written
when a crash occurred. After a processor crash, in a variable length
environment, the pair of objects forming a stable object will be in one
of the following states.

(i) Both objects have valid data (which may be different).

(ii) One of the objects is detectably bad.

(iii) One of the objects has invalid data but it is not known
which object it is.

In case (ii) a crash occurred while the disk write heads were on the disk
surface thereby making the data on the disk invalid. The data is detect-
ably bad and the error correcting codes will reveal this fault when the
data is read. In case (iii), a write operation to one of the two objects
was interrupted by a crash resulting in only part of the data being
written. The data on the disk is not detectably bad though the data is
in an inconsistent state. Since it is not known which object was being
written when a crash occurred, it follows that we do not know which
object contains inconsistent data. This is unlike the fixed page
environment where we assume that the data on the disk is either con-
sistent or it is detectably bad. This difficult situation where the data
is not detectably bad but in an inconsistent state does not arise in the
fixed page environment. In the variable length situation it is therefore
necessary to determine, for purposes of crash recovery, the consistency
status of each of the two objects that represent a stable object. The
fixed page solution which copies any one of the pages to the other page
when it is determined that the pages are not detectably bad is not suit-
able in the variable length environments. If we do not know which of
the objects is consistent, copying one of the objects to the other could
mean copying the inconsistent object to the consistent object thereby
making both objects inconsistent.

To solve this problem, we considered timestamping each update
operation. A timestamp would indicate to us the update operations that
belong together but will not necessarily enable us determine which
update operation was completed and which was interrupted. One approach
to timestamping every update operation would mean having timestamps
scattered in the file. We found this undesirable since we regard a file
to be a data entity which has a meaning to the user. Alternatively,
these stamps could be kept transparent to the user by storing them in
another file which in turn has to be made crash resistant. This would
increase the number of accesses to a disk storage device and hence the
time spent in carrying out each stable operation. This was considered
expensive and was not pursued further. Another possibility involved
circulating a token between the two files of a stable file such that
only the consistent file holds the token. This scheme was found unsuit-
able for our purposes because sometimes the two files representing a
stable file are both consistent.

We used instead what we called a "moving tag" to solve this prob-
lem. A "moving tag" is a concatenation of any small set of characters
such that the resulting string is assumed not to occur naturally in the
user's file. Each logical write operation writes a tag onto a stable
file after the successful completion of its operation. A subsequent
update request would overwrite the tag written by the previous one while
writing its data. The tag is therefore in effect removed by each logi-
cal write operation and reinstated always at the end of file. It was
the removal and reinstatement of the tag that gave it the name "the mov-
ing tag". This scheme has the desirable property that at any given
time, only one tag is found on a stable file and this tag is located at
the end of the file, as opposed to having timestamps on every block of
data. The stable read routine keeps the tag transparent to the user. A
user therefore sees a stable file as an ordinary UNIX file. After a pro-
cessor crash, a stable file which is inconsistent with respect to a
user's request would contain no tag. In such a case, the crash recovery
routine would be called to restore the consistency of a stable file. A
pictorial illustration of this scheme is given in fig. 4.

```
  a1  a2    a3  a4
  ------------|------------|--------
    data     data  tag    tag
  ------------|------------|--------
      b1      b2      b3
```

Fig. 4: Two logical write requests.
The stable storage system sometimes has to carry out several physical write operations in order to satisfy a user's single (variable length) logical write request. Each such physical write operation writes only fixed size blocks and the atomicity of these operations is guaranteed by the disk hardware. Fig. 4 shows two write system calls issued by a user. The first request which we call the A-update is to update the set of bytes starting at a1 and ending at a3. The stable storage system splits this request into three physical write operations (a1, a2), (a2, a3) and (a3, a4). The operation which writes (a3,a4) constitutes the writing of the tag. The second update request called the B-update is to update the set of bytes starting at b1 and ending at b2. In order to satisfy this request, the stable storage system carries out two physical write operations (b1, b2) and (b2, b3). The B-update overwrites the tag written by the A-update by starting its write operation at a3 instead of a4.

Let us consider the effects on fig. 4 of an adverse environment activity such as a processor crash which could interrupt a write operation. The problem is to ensure that the A-update and the B-update which are variable length stable write operations are atomic. The following are the various scenarios when the operations in fig. 4 are interrupted by a crash. A crash can occur after a2, a3, a4, b2 or b3. A crash is not expected to occur between ai and aj nor between bi and bj (where j = i + 1), since these operations are guaranteed to be written atomically by hardware. If a crash occurs after a2, a3 or b2, a stable file will be considered inconsistent since either a logical write operation has been started but not yet completed or a tag has not yet been written to indicate its completion. In such a case, the crash recovery routine which is invoked by the system manager after a crash will restore the consistency of a stable file. If a crash occurs before a1 or after a4 or b3, a stable file will be considered consistent since either a logical write operation has either not been started or it has been completed. In all cases, the atomicity of the variable length stable write operation is always guaranteed. How the crash recovery routine restores the consistency of a stable file is described in the section on "Stable Layer Implementation".

This scheme is most efficient when the stable file is an append-only data structure. To support random access write operations, the crash recovery routine would have to be extended to expect the tag anywhere within the body of a file, as opposed to the present implementation where the tag is at the end of the file. This extension is trivial but would increase the time spent for crash recovery operations. The other observation is that most disk hardware do not write across block boundaries. Consequently the writing of the tag will in effect constitute the writing of one physical block which is then overwritten by the next update operation.

5.2. Stable Operations

We shall now describe the implementation of stable operations using the notation of section 4.1. An example of how this notation was implemented using the exception handling package described in section 4.2 is given in the appendix. The stable operations are organised as two
levels of abstraction called the Transient and Stable layers.

5.2.1. Transient Layer Implementation

The transient layer implements the server processes which constitute the first level of abstraction of the stable storage system. This layer masks transient I/O faults. It consists of two procedures, Tread and Twrite. The operations on this layer use the primitives provided by the UNIX kernel to produce a better behaved set of operations by performing read-retries and by providing a read-after-write capability for their write operations. The procedures Tread and Twrite are invoked by the stable layer and are not intended to be invoked directly by users.

```plaintext
Proc Tread(fd:int; buf:array[...]) of char; nbytes:int) SIGNALS
    RD-FAIL, DISKERR, NOT-DISK:exception;
var fd: int;
begin
   (N)read(fd,buf,nbytes)[NOT-DISK::SIGNAL RD-FAIL;
    DISKERR::report, SIGNAL RD-FAIL];
end.
```

Procedure Tread masks transient read errors. The meaning of the read statement in the procedure Tread is the following: If a read operation fails, Tread would perform read-retries until a read operation succeeds, up to a maximum of N read-retries. The value chosen for N is determined by previous observation of the average latency period for transient faults on disks. If any of the read operations fails due to a fault that is not a disk fault (that is, if a NOT-DISK exception is detected), Tread terminates by signalling the read-failure exception, RD-FAIL, without making further read-retries. If all its retry attempts persistently detect a disk fault (DISKERR), then Tread writes an error report (intended for the maintenance engineer) and then signals read-failure.

```plaintext
Proc Twrite(fd:int; buf:array[...]) of char; nbytes:int) SIGNALS
    WRT-FAIL;
NOT-DISK,DISKERR,WRT-FAIL:exception;
var fd: int;
begin
   (N)write read(fd,buf,nbytes)[NOT-DISK::SIGNAL WRT-FAIL;
    DISKERR::report,
    SIGNAL WRT-FAIL];
end.
```

Procedure Twrite masks the effects of transient write errors. It repeatedly performs a write followed by a read until
(i) the value read is equal to the value written, thereby confirming that the data was written successfully, or

(ii) until it has attempted N write-read retries.

This provides a read-after-write capability for disks that do not have this facility. It masks the effects of bad writes, i.e., those which write wrong values to the disk. It detects write operations which write to the wrong address and write errors which do not change the disk state. If any of its write-followed-by-read operations fail, due to a fault that is not a disk fault, then Twrite will terminate by signalling the write-fail exception, WRT-FAIL. Furthermore, if all its retry attempts fail, it writes a report (intended for the maintenance engineer) and terminates exceptionally by signalling a write-fail exception.

5.2.2. Stable Layer Implementation

The stable layer implements the second level of abstraction of the stable storage system. It provides fault tolerance facilities for decays and processor crashes by using file replication. It also provides file replication transparency so as to conform to a uniform interface with UNIX system calls. The stable layer uses the better behaved operations provided by the transient layer instead of the ordinary operations provided by the UNIX kernel. It consists of the routines Sread, Swrite and Crec. Sread and Swrite are the routines which are used for reading from and writing to stable files on behalf of the user. These two routines are normally invoked by the user. However in our environment, the stable storage system is kept transparent to the user and the user invokes what he thinks is the UNIX read/write operation which in fact is the Newcastle Connection read/write operation. This operation activates the file server in the appropriate UNIX system which then invokes the stable storage system.

Proc Sread(fdp:int; buf:array[...],nbytes:int) SIGNALS SRD-FAIL;
SRD-FAIL,DISK1-BAD,DISK2-BAD:exception;
var fd1,fd2:int;
begin
  Tread(fd1,buf,nbytes)[DISK1-BAD::Tread(fd2,buf,nbytes)
  [DISK2-BAD::SIGNAL SRD-FAIL]];
end

The procedure Sread reads from the first disk by using the file descriptor fd1. If this is unsuccessful, it reads from the second disk using the file descriptor fd2. If the read from the second disk fails, Sread terminates by signalling a stable-read-fail exception, SRD-FAIL. As we shall see later, the activities of the crash recovery routine make the failure of the stable read/write operations rare.
Proc Swrite(fd1: int; buf: array[] of char; nbytes: int) SIGNALS
SWRT-FAIL; DISK1-BAD, DISK2-BAD: exception;
var fd1, fd2: int;
begin
    Twrite(fd1, buf, nbytes)[DISK1-BAD::SIGNAL SWRT-FAIL];
    Twrite(fd2, buf, nbytes)[DISK2-BAD::report];
end.

The procedure Swrite writes to the two disks, which have file
descriptors fd1, fd2. If the write operation to the first disk fails
however, the write operation to the second disk is not initiated and
Swrite terminates exceptionally by signalling a stable-write-fail excep-
tion, SWRT-FAIL. This helps ensure that not more than one disk can be
damaged following a crash during a write operation. However, one has to
be careful to ensure that when a stable write operation returns, the
data have actually been written to the disk and not buffered by the
operating system. This is achieved by either communicating directly with
the disk in "raw mode" [11] or by forcing the system write buffers to be
flushed after each write operation. Flushing the system's write buffers
was found expensive and undesirable. Our approach involves using disks
which can be divided into several virtual disks. These virtual disks
are treated as real devices by the operating system and as files by the
stable storage system. This enables us to obtain several files on one
physical disk while retaining the capability to address these files in
"raw mode" by transferring information between the user's core image and
the device without the use of the UNIX buffering mechanism.

Proc Crec(fd1, fd2: int) SIGNALS D1-LOST, D2-LOST, ALL-LOST;
D1-LOST, D2-LOST, ALL-LOST, WRT-FAIL, RD-FAIL: exception;
fd1, fd2, nbytes: int;
s1, s2: boolean initially false; /*decay switches*/
cetag1, cetag2: boolean initially false;
/*consistency status indicator*/
buf1, buf2: array[...] of char;
begin
    /*check for consistency tag*/
    if file1 contains a consistency tag, set cetag1 = True;
    if file2 contains a consistency tag, set cetag2 = True;
    if (cetag1 and cetag2) then
        /*both files good: upon reading if you encounter
detectably bad file, copy good to bad*/
    repeat
        Tread(fd1, buf1, nbytes)[RD-FAIL:: s1];
        Tread(fd2, buf2, nbytes)[RD-FAIL:: s2];
        if s1 /*file1 is detectably bad*/
           then Twrite(fd1, buf2, nbytes)[WRT-FAIL:: ALL-LOST]
           else Twrite(fd2, buf1, nbytes)[WRT-FAIL::ALL-LOST];
    until eof
    /*note that we assume that both disks will
not be bad at the same time*/
else if ctag1 then /*first file good; read from the first
file, write to the second */
repeat
    Tread(fd1, buf1, nbytes)[RD-FAIL::SIGNAL ALL-LOST];
    Twrite(fd2, buf1, nbytes)[WRT-FAIL::SIGNAL D2-LOST];
until eof
else if ctag2 then
repeat
    Tread(fd2, buf2, nbytes)[RD-FAIL:: SIGNAL ALL-LOST];
    Twrite(fd1, buf2, nbytes)[WRT-FAIL:: SIGNAL D1-LOST];
until eof;
end

The crash recovery routine Crec implements the crash recovery
facilities of the stable storage system. It is invoked by the system
manager to restore the consistency of a stable file. Its action is
applied to each file on a disk after a crash, before the system restarts
normal operation. The system manager also invokes this routine after
every time interval Tu, for maintenance purposes. The determination of
Tu depends on previous observation of the system, which will indicate
approximately how often a decay is expected to occur on disk. During
crash recovery, this routine first determines the consistency status of
each file by searching for the consistency tag at the end of a file. If
a stable file is found to be inconsistent, which means that one of the
two files representing a stable file has no tag on it, this routine
copies the consistent file (with a tag) to the inconsistent file. If
each of the two files contains a consistency tag, the crash recovery
routine will try to copy all readable blocks of data from the first file
to the second file. This copying of one consistent file to another is
for maintenance purposes, and helps to ensure that all blocks of data in
the two files are readable. If a block of data from the first file is
not readable (detectably bad) then the crash recovery routine would copy
the block of data from the second file to the first file and vice versa.
If Crec cannot complete its operations due to the existence of per-
mmanent decays, appropriate reports are issued to the maintenance
engineer who, we hope, initiates repair operations on the affected disk.

Our fault assumptions are that there will be no more than one decay
on the same disk within a time interval Tu and that no more than one of
the disks is bad at the same time. If we can assume that necessary
repairs will be effected within Tu time of detection of faults, then the
stable storage system can be said to provide "failure-free" disks. This
is because once one disk becomes bad, the second disk can not (by our
assumption) become bad within a time interval Tu. Within this time
interval, the crash recovery routine Crec would have been invoked by the
system manager to correct any damage due to transient decays and
crashes, and repairs would have been effected by the maintenance
engineer. The practicality of the assumption that repairs can be
effected within a suitable time interval Tu can certainly be questioned.
However it might not be unreasonable in certain environments.
5.2.3. Other Stable Operations

We also found it necessary to implement a stable version of some operations that use disk storage. The existence of these stable operations is however transparent to the user. The following routines were implemented:

Stable-Open, Stable-Close, Stable-Lseek, Stable-Creat.

These routines are called by the stable storage system in response to the users' open, close, lseek, and creat system calls. Interested readers are referred to UNIX documents for a more complete understanding of the Open, Close, Lseek and Creat system calls.

6. Performance Measurements

Some initial tests were carried out to assess the performance of the stable storage system. The tests were performed on a PDP 11/45 running V7 UNIX and using RK05 and RLO1 disks. The aim was to compare the disk access times for ordinary disk storage and stable storage. The time spent in reading/writing 50k bytes of data from an ordinary file and from a stable file were recorded and compared. Data blocks of varying sizes ranging from 64 to 2048 bytes were used. The UNIX facility "time" was used for obtaining time measurements to the nearest millisecond. Table I contains some performance figures. These figures are averages calculated from the results of a number of experiments.

<table>
<thead>
<tr>
<th>UNIX read</th>
<th>stable read</th>
<th>UNIX write</th>
<th>stable write</th>
<th>block size</th>
<th>total no. of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.40 sec</td>
<td>20.60 sec</td>
<td>20.40 sec</td>
<td>84.880 sec</td>
<td>64 bytes</td>
<td>50k bytes</td>
</tr>
<tr>
<td>10.60 sec</td>
<td>10.80 sec</td>
<td>10.80 sec</td>
<td>44.740 sec</td>
<td>128 bytes</td>
<td></td>
</tr>
<tr>
<td>5.60 sec</td>
<td>5.80 sec</td>
<td>5.80 sec</td>
<td>24.600 sec</td>
<td>256 bytes</td>
<td></td>
</tr>
<tr>
<td>2.580 sec</td>
<td>2.60 sec</td>
<td>2.600 sec</td>
<td>13.620 sec</td>
<td>512 bytes</td>
<td></td>
</tr>
<tr>
<td>1.340 sec</td>
<td>1.360 sec</td>
<td>1.360 sec</td>
<td>13.120 sec</td>
<td>1024 bytes</td>
<td></td>
</tr>
<tr>
<td>0.720 sec</td>
<td>0.740 sec</td>
<td>0.720 sec</td>
<td>12.680 sec</td>
<td>2048 bytes</td>
<td></td>
</tr>
</tbody>
</table>

Table I
Data Access times for UNIX and stable read/write operations

We observed that the time required by stable read operation is approximately equal to the time required by the UNIX read operation when exceptions are not encountered. For the write operation, the access time ratio of stable write operation to UNIX write operation is approximately 1:4. The 1:4 ratio is due to the fact that the stable write operation uses a better behaved write operation provided by the
transient layer instead of the ordinary UNIX kernel primitive. This provides
the facility for writing atomically variable length stable objects
and a read-after-write capability for each write operation to the two
files that make up a stable file. Fragmentation of larger blocks of
data into 512 byte blocks by the transient layer makes the stable write
operation much slower than the ordinary UNIX write operation when block
sizes which are larger than 512 bytes are used. The 1:4 ratio is gen-
erally maintained if the transient layer operations read and write
larger blocks of data. In our environment, a weak atomicity of opera-
tions which write 512 byte blocks is provided by the operating system
and we needed to base our operations on that facility and hence the
fragmentation of larger blocks.

From these figures it can be seen that applications which perform
mostly reading operations pay only a very little price for using the
stable storage system. On the other hand, for applications which are
writing most of the time, using the stable storage system could account
for an increase in overheads. Systems differ greatly as to the extent
to which disk operations dominate the average workload. Moreover accu-
rate figures on the ratio of read to write system calls are not
presently available, so it is difficult to estimate how large an over-
head the use of the stable storage system will be. Wyeth [12] in his
simulation of a recursive cache mechanism analysed the references to
shared resources in a set of sequential programs and his figures show
that read operations occur three times as frequently as write opera-
tions. Applying his results to disk storage would suggest that the
number of read accesses will generally be fairly high. For such appli-
cations the use of the stable storage system will rarely be noticed on
the user level.

7. An Alternate Approach

This implementation avoided a direct use of the UNIX file system by
implementing stable files as virtual disks which are treated as devices
by the operating system. Our aim was to provide an independent facility
which is not embedded in the operating system so as to keep the imple-
mentation portable. An alternate approach would be to provide robust-
ness while using the UNIX file system by making the UNIX file system
itself crash resistant. This would require kernel modifications in
order that system file management information (such as i-nodes) can be
accessed and made stable. We would also then have to deal with the
problems posed by the UNIX buffering mechanism which forms the block-
device interface. One such problem is the issue of delayed write opera-
tions. If a crash occurs, there could be logically completed but physi-
cally incomplete I/O in the buffers and these would be destroyed by a

Such an implementation would have the advantages of utilising
fully the facilities provided by the UNIX file system. It will however
be less portable than the present implementation. We are currently car-
rying out such an exercise and the result of that experiment is the sub-
ject of another paper.
8. Conclusion

This prototype stable storage implementation has provided a facility which helps maintain the consistency of data stored on disk storage devices. Through the use of systematic handling of abnormal situations it has provided simple, reliable and efficient stable operations which can be used to build arbitrarily large atomic actions needed at the transaction level. The provision of the abstraction of stable files extends the domain of reliability facilities available to the UNIX user. As was indicated earlier, it would be possible to incorporate it into UNIX United in such a way as to make it a more generally available facility. It is hoped to investigate such issues in due course.

Acknowledgements

Acknowledgement is due to B. Randell who advised that this implementation be carried out and whose critical comments led to the improvement of this paper. Acknowledgement is also due to F. Cristian who allowed me to benefit from his design methodology for fault tolerant systems. The work described in this paper has benefited greatly from the advice of T. Anderson. Discussions with L. F. Marshall and S. K. Shrivastava have been of great help in this implementation.
References


Appendix

Implementation of the Fault Tolerance Notation

This section shows how the fault tolerance notation was implemented using the exception package described in section 4.2. This is done by presenting the implementation of the procedures of the transient layer, procedures Tread and Twrite, using this package and a Pascal-like language for readability. The exception handling package does not allow the use of break or return statements within an exception context. We however used break and return statements below whenever we found that they improved readability. Exc-signal statements are used to indicate exceptional return of a procedure and Return statements are used to indicate normal return.

Proc Tread(fd: int; buf: array[... of char; bufsize)
const bufsize = 512; /*size of buffer */
type answer = (goodread, diskerror, ...);
var result: answer;
retryno, maxretry:int;
BAD-DISK, OTHERS, OP-FAIL: exception;
begin
BEGIN
/*beginning of exception context*/
repeat
result = read(fd, buf, bufsize);
if result = goodread then break /*exit from loop*/
else
  if result = diskerror /*disk error detected*/
    then exc-raise(BAD-DISK)
    else exc-raise(OTHERS); /*error not disk error*/
EXCEPTION
WHEN(BAD-DISK)
retyrno = retryno + 1;
WHEN(OTHERS)
write reports, "error not disk error";
break;
END /*end of context */
until retryno >= maxretry;
if result = goodread then return(result)
else exc-signal(OP-FAIL);
end.
Proc Twrite(fd: int; buf: array[...] of char; bufsize)  
const bufsize = 512;  
type answer = (goodread, goodwrt, diskerror, ...);  
var resultw, resultr: answer;  
ok-wrt: boolean initially False;  
BAD-DISK, BAD-RAW, OP-FAIL, OTHERS: exception;  
begin  
BEGIN  
    /* beginning of context */  
    repeat  
        resultw = write(fd, buf1, bufsize);  
        if resultw = goodwrt then  
            /* do the read-after-write */  
            begin  
                seek to beginning-of-block;  
                resultr = read(fd, buf2, bufsize);  
                if resultr < > goodread  
                    then exc-raise(BAD-DISK);  
                    /* can't read written values */  
                    if buf1 = buf2 then  
                        begin ok-wrt = True;  
                          break;  
                        end  
                else exc-raise(BAD-RAW);  
                /* detects good write that writes wrong values */  
            end  
        end if resultw = diskerror then exc-raise(BAD-DISK)  
        else exc-raise(OTHERS);  
        EXCEPTION  
        WHEN(BAD-DISK)  
            retryno = retryno + 1;  
        WHEN(BAD-RAW)  
            /* bad read-after-write */  
            retryno = retryno + 1;  
            lseek to beginning of block  
            /* necessary since write operation was successful */  
        WHEN(OTHERS)  
            write reports "not disk error";  
            break;  
        END  
        /* end of context */  
        until retryno >= maxtry;  
        if ok-wrt then return(resultw)  
        else exc-signal(OP-FAIL);  
    end.  

One of the difficulties encountered in separating normal and exceptional processing in this implementation was that the fault tolerance notation does not yield to a direct implementation by the exception handling package. We also always have the need to return a result object both during normal and exceptional termination since the caller of a function in our environment quite often interrogates and uses this returned value. The concept of a procedure terminating in more than one
way and returning result objects differing in number and type is not addressed by the exception handling package. One of the reasons for this is that the ADA language exception handling model on which this exception package is based does not have parameterised exceptions, through which exceptional result objects can be returned (as in the CLU language, for example).

Another difficulty was the use of exception contexts, within which a designer is allowed to terminate the execution of a block by raising exceptions but not allowed to terminate normally by issuing a return statement. Despite these difficulties, we found that the separation of normal and exceptional processing and the clean control structure which the exception package gives, greatly enhances the reliability of our implementation.