# ROBUST DATA STORAGE IN A NETWORK OF COMPUTER SYSTEMS

Josephine A. Anyanwu

Ph.D. Thesis

Computing Laboratory
University of Newcastle Upon Tyne, England

November 1984

CONTENTS

## ABSTRACT

Robustness of data in this thesis is taken to mean reliable storage of data and also high availability of data objects in spite of the occurrence of faults. Algorithms and data structures which can be used to provide such robustness in the presence of various disk, processor and communication network failures are described.

Reliable storage of data at individual nodes in a network of computer systems is based on the use of a stable storage mechanism combined with strategies which are used to help ensure crash resistance of file operations in spite of the use of buffering mechanisms by operating systems. High availability of data in the network is maintained by replicating data on different computers and mutual consistency between replicas is ensured in spite of network partitioning.

A stable storage system which provides atomicity for more complex data structures instead of the usual fixed size page has been designed and implemented and its performance evaluated. A crash resistant file system has also been implemented and evaluated.

Many of the techniques presented here are used in the design of what we call CRES (Crash-resistant, Replicated and Stable) storage. CRES storage provides fault tolerance facilities for various disk and processor faults. It also provides fault tolerance facilities for network partitioning through the provision of an algorithm for the update and merge of a partitioned data storage system.

**ACKNOWLEDGEMENT**

I would like to thank Dr. T. Anderson, my supervisor, and Professor B. Randell for their encouragement and highly constructive criticisms and suggestions during my Ph.D. study. I would like to thank them for their critical reading of my reports and early drafts of this thesis. I consider myself extremely lucky to have been given the opportunity to benefit from their vast experience.

My special thanks are due to Dr. S. K. Shrivastava whose invaluable advice influenced some of the ideas presented in this thesis.

I would also like to thank Dr. L. F. Marshall for his advice during the implementation of the crash resistant UNIX file system. My initial tour of the UNIX kernel benefited greatly from his knowledge of that system.

# 1. INTRODUCTION

## 1.1. INTRODUCTION

As more and more computers are used in the automation of various essential services, the reliability of computer systems has become increasingly important. Most of the results of the processing which is carried out by computer systems are stored in some form of storage medium. Many computer programs get their input data from such media. A fault in any part of a computer system could easily be manifested as an error in the stored data at some point in the computation process. (We will rely on the intuitive meaning of faults, errors and failures in this introductory part of the thesis. A definition of these concepts which conforms to normal computer usage is given in the next chapter.)

Since programs often share data, an error in a single data item can be propagated throughout a computer system. Correct programs using such erroneous data could produce erroneous results. Users of computer systems expect that the data which is stored by such systems will be reliably stored. By this we mean that users who have entrusted their data to a computer system expect to be able to get that data when they want it.

Moreover, they also expect to get exactly the same data which they entrusted to the computer system and not a mutilated version of that data. Where such assurances cannot be given, users lose confidence in the computer system and could prefer to store their data in a more primitive but reliable and trusted medium. Hoare[Hoare75] described an error in a data bank to be potentially as catastrophic as an error in an ordinary bank which keeps money for its clients. The reliability of computer systems clearly depends heavily on the reliability of the data which are stored by these systems.

There are two complementary approaches to the provision of reliable computer systems. Avizienis[Avizienis75] calls these two approaches fault-tolerance and fault-intolerance. The fault intolerant approach aims at avoiding faults, and tries to eliminate all sources of unreliability from a computer system. It requires that efforts be spent in perfecting a computer system (hardware and software) before such a system is put into use. Hardware systems based on this approach use the most expensive high quality circuits in the construction of computers. There is no guarantee, however, that these circuits will never fail.

Software systems which use the fault intolerant approach aim at exhaustive testing of software modules so as to eliminate all errors. Program verification methods can also be used with the aim of proving programs correct. A program which has been proven correct is expected to behave always as specified. There has been reasonable progress in the area of program verification and correctness proofs. However, proofs can be faulty and even rigorously proven programs are executed on hardware systems which have not been proven correct. It is therefore difficult to ensure that such programs will always produce correct results. Likewise, extensive testing of software modules cannot guarantee the absence of errors except in very special situations. Fault intolerant approaches may therefore not be sufficient for the provision of highly reliable computer systems. There is the need to consider fault tolerant approaches as a complementary, rather than an alternative, method for providing high reliability.

Fault tolerance is based on the use of redundancy. Such redundancy is used to detect the causes of unreliability in computer systems and to counteract their adverse effects. The causes of unreliability are expected to be present but reliable computing is achieved in spite of their existence. Redundancy in fault tolerant systems can be classified as (1) masking redundancy and (2) dynamic redundancy. In masking redundancy, redundant components are incorporated into a system in such a way that the effects of component faults are not visible to the environment of the component. Dynamic redundancy on the other hand is used for detecting the existence of errors in a system so that an appropriate form of error recovery can be invoked. The combined use of masking and dynamic redundancy enables the detection of errors in computer systems and the provision of continued service in the presence of faults.

## 1.2. AIM OF THESIS

This thesis is concerned with the provision of fault tolerance facilities in a decentralised computer system. It considers the problem of providing reliable storage of data in distributed environments so that the integrity of data is maintained and there is high availability of stored data. Specifically, it considers the problem of providing disk storage facilities that can tolerate various types of disk and processor faults and using these facilities as the basis for constructing highly reliable file systems. Our data storage system is designed to be used by a set of computers connected by a communications network. This thesis therefore also considers the problem of ensuring that the availability of files in the network is high in spite of various node and network failures.

Our emphasis is on how to make file systems reliable as opposed to the design of file systems per se. This is because there are many file systems already in existence and the design of such systems is much better understood than the problem of making these systems reliable.

There are now many distributed systems in which user data is accessible throughout a network irrespective of location. Examples of such systems are UNIX United[Brownbridge82], INGRES[Stonebraker76], XDFS[Sturgis80] and SDD-1[Rothnie80]. Our experiments were carried out in a UNIX United distributed system. A UNIX United system joins together several individual UNIX systems so as to give the illusion of a single UNIX system. Users in any UNIX system within the UNIX United framework can access data in the other UNIX systems as though the remote data were part of the user's own local system. An overview of the UNIX United system and its naming scheme is given in chapter 3.

Node and network failures have been recognised as major impediments to the provision of reliable storage in distributed systems. In addition, transient hardware faults and decay phenomena of disk storage devices could affect the integrity of stored data. Various solutions to some of these problems have been proposed. In chapter 2 we shall discuss some of the proposed solutions.

A mechanism known as stable storage[Lampson79] has been advocated as a useful paradigm for the construction of reliable data storage systems. This thesis investigates the use of the stable storage mechanism and its associated crash recovery facilities to help ensure that the probability of losing data which is stored on disk storage devices is negligible.

The provision of such a facility would be sufficient for the construction of a reliable data storage system if there were no need to provide reliable management of stable objects created by a stable storage system, and if such objects were always available. These problems lead us to extend the concept of stable storage to what we call CRES (Crash-resistant, Replicated and Stable) storage. CRES storage addresses the problem of maintaining the consistency of data as well as the management and availability of data objects. The next section summarises our approach to the construction of CRES storage.

## 1.3. SUMMARY OF APPROACH

Our approach to reliable data storage construction proceeds in three stages. Figure 1 shows the relationship between these three stages.

```
                  CRES Storage
  (Crash-resistant, Replicated and Stable Storage)
                          |
--------------------------|-----------------------------
 Crash-resistant  Crash-resistant   Crash-resistant
   File System      File System       File System
 (on computer 1)  (on computer 2)   (on computer 3)
        |                 |                 |
--------|-----------------|-----------------|-----------
   Stable Disk      Stable Disk       Stable Disk
  (on computer 1)  (on computer 2)   (on computer 3)
        |                 |                 |
--------|-----------------|-----------------|-----------
       /  \             /   \             /   \
     /      \          /     \           /     \
Physical  Physical  Physical Physical  Physical Physical
Disk      Disk      Disk     Disk      Disk      Disk
---------------------------------------------------------
```

Figure 1: Levels of Abstraction for CRES Storage

The first stage, which is described in chapter 3, provides virtual disks (called stable disks) that are much more reliable than their real counterparts. The abstraction of stable disks is used to maintain the consistency of data which is stored on disk storage devices in spite of transient I/O faults, decay of the storage medium and some effects of processor crashes. A novel algorithm for supporting more complex data structures instead of ordinary pages in stable storage design is proposed. This algorithm provides atomic objects of variable length instead of the usual fixed size pages. A stable storage system utilising this algorithm has been implemented and its performance characteristics evaluated.

The consistency of a stable storage system does not guarantee the consistency of a file system which uses it because of some faults that can arise from certain file management activities. The second stage of our experiment, which is described in chapter 4, is thus concerned with the reliable management of data stored on stable disks. This stage implements a crash resistant file system by ensuring that the use of data management facilities (such as buffering mechanisms employed by file systems) do not result in data inconsistency when a processor crash occurs. A crash resistant UNIX file system which is based on the abstraction of stable storage has been implemented by making modifications to the UNIX kernel. An evaluation of the performance of this implementation has also been carried out.

A reliable data storage system should ensure that data items are accessible in spite of node and network failures. The third stage of this work, which is the subject of chapter 5, replicates the reliable files which are implemented by the second stage on different computers so as to increase the availability of these files. The use of data replication requires that the mutual consistency of replicas be maintained. Network partitioning resulting from the failure of the communication network could interfere with the preservation of such consistency. We present an algorithm for the update and merge of a partitioned data storage system which differs from work done by others in this area. Algorithms which provide a merge of partitioned networks require that either the semantics of all operations on data are known or that application programs declare their readset and writeset in advance. (The readset and writeset of an application program are those data items which the program reads from or writes to.) Other algorithms depend on the ability to obtain a global serialisable schedule of the combined set of transactions from all partitions in a network[Wright83]. In contrast, our algorithm dynamically determines the readset and writeset of each partition, so that conflicting update operations can be detected. Its merge protocol enables resolution of conflicts before an actual merge operation is carried out.

Before we proceed with the discussion of reliable data storage construction, we survey in chapter 2 the various faults which affect the consistency of data and measures and mechanisms for maintaining data consistency in distributed systems. The three stages in our data storage construction will be presented in chapters 3, 4 and 5 respectively. Chapter 6 suggests areas for future study and presents some concluding remarks.

# 2. DATA CONSISTENCY IN DISTRIBUTED SYSTEMS

## 2.1. INTRODUCTION

Systems which store data usually have a set of constraints which must be maintained. Data stored by such systems are considered to be consistent if the set of predicates which prescribe the systems constraints are satisfied. In order to discuss data consistency therefore, it is important to examine the faults which could violate these constraints.

Our concepts of faults, errors and failures are based on the definitions by Randell[Randell78] and Anderson[Anderson81]. We consider a system to be a set of interacting components which is designed to provide a specified service. In providing this service, a system makes state transitions from an initial state to a final state. A sequence of state transitions which takes a system from an initial state to a final state so as to provide the specified service will be referred to as a correct sequence of transitions.

A failure occurs when a system does not provide its specified service. A system is said to contain an error when it makes an incorrect state transition such that subsequent state transitions cannot lead to the provision of the specified service. A fault is the underlying cause of an error.

We follow the model of errors in[Lampson79] and classify errors as either expected or unexpected. A system is in a normal state if it contains no errors otherwise it is in an erroneous state.

```
            System
             / \
            /   \
      Normal  Erroneous
      State     State
                 / \
                /   \
          Expected  Unexpected
           Errors     errors
```

Figure 2: System State Classifications

Figure 2 gives a pictorial representation of system states based on this classification of errors. We consider a fault which affects data to be any event which makes stored data irretrievable or which alters the contents of stored data to a state which is different from that specified by the user as stated in the system constraints.

Recovery of data objects aims at restoring data in a system to a usable state by eliminating errors. Concurrency control mechanisms are used to ensure that interference between user update operations do not result in data inconsistency. Discussion of recovery techniques can be found in [Verhofstad78], [Gray81], [Kohler81], [Haerder83]. In [Bernstein81] Bernstein presents a survey of concurrency control schemes. This chapter surveys the general area of data consistency, of which recovery and concurrency control form a part. It examines data consistency with respect to

(1) Consistency in the presence of concurrent access to shared data.

(2) Consistency in the presence of component faults

(3) Mutual consistency between replicated resources.

This method of survey has the advantage of giving a unified approach to concurrency, recovery and replication issues with respect to their roles in the maintenance of data consistency.

The organisation of this chapter is as follows. The next section introduces distributed systems concepts and problems. Section 2.3 discusses the causes of inconsistency in stored data. In section 2.4 measures which are used to maintain the consistency of data in spite of faults are discussed. Section 2.5 presents some concluding remarks.

## 2.2. DISTRIBUTED SYSTEMS

The subject of distributed computing systems is relatively new and there are several definitions of such systems. We consider a computing system to be distributed if it consists of a collection of computers (loosely or tightly synchronised) which communicate by means of a communication network and in which there is no shared memory among individual computers. Some of the problems which arise in distributed systems are due to the physical separation of the components and the possibility of heterogeneity among such components. Some of these problems are concerned with providing a uniform naming scheme for objects, implementing an interprocessor communications scheme, synchronising the activities of the various computers and providing a mechanism to handle new sources of errors which might arise due to the distribution of processes and data. For ease of discussion we shall consider data items in a distributed system to be grouped into objects, and activities which manipulate data will be referred to as tasks.

## 2.3. CAUSES OF DATA INCONSISTENCY

This section discusses the major causes of inconsistency in stored data, namely update interference, component faults and causes of mutual inconsistency between replicated data.

### 2.3.1. Update Interference

Interference between concurrent update operations is a fault arising from interaction between tasks in a computer system. If tasks are correctly carried out separately one at a time, then in the absence of faults, the output produced by each task will be consistent with the user's specification. However, if several tasks are performed concurrently and make use of common resources there is the possibility that the activities of various tasks on shared data could interfere in such a way that outputs are produced which do not conform to specification. Concurrency control schemes such as those presented in [Bernstein81], [Rosenkrantz78] are used to prevent such interference between tasks. Bernstein and Goodman [Bernstein81] illustrated the concurrency control problem by giving examples of anomalies which could arise in on-line electronic fund transfer systems. In such systems, a customer is allowed to read data, perform some computations and then write the results in the data storage system. The lost update problem and the retrieval of intermediate data are examples of problems which could arise in these systems.

**Example 1: The Lost Update**

Suppose that two customers A and B are accessing a bank account X at the same time. Suppose that account X contains one thousand dollars ($1000). Let A and B execute the following algorithms:

```
 A's Algorithm           |  B's Algorithm
                         |
Read X into XA;          |  Read X into XB;
     /* X = 1000*/       |      /* X = 1000*/
Add 10 to XA;            |  Add 100 to XB;
    /* X = 1000*/        |      /* X = 1000*/
Write XA into X;         |  Write XB into X;
   /* X = 1010*/         |      /* X = 1100*/
```

Figure 3: The Lost Update

Let us assume that the instructions on the same line are carried out in parallel. Since A and B have deposited $10 and $100 respectively into account X, the expected final value of X should be $1110 which would reflect both update operations. From figure 3, it can be seen that only one update operation will actually be reflected (either A's or B's update operation). The other operation is lost. Concurrency control mechanisms are used to ensure that such user interference do not result in a data storage system having a state which is inconsistent with the user specification.

**Example 2: Retrieval of Intermediate Data**

A problem can also arise when data which is in an intermediate state is retrieved by a user. Suppose that a customer maintains two accounts, a deposit account S and a current account Y which contain $1000 and $200 respectively. If a user A is transferring $100 from the deposit to the current account while a user B is printing the total balance in the bank account (the sum of the amounts in the deposit and current accounts), the algorithm executed by A and B could be the following:

```
A's Algorithm               |  B's Algorithm
                            |
Read S into SA;             |
     /* S = 1000*/          |
Subtract 100 from SA;       |
     /* S = 1000*/          |
Write SA into S;            |
     /* S = 900*/           |
                            |  Read S into SB;
                            |       /* S = 900*/
Read Y into YA;             |  Read Y into YB;
     /* Y = 200*/           |       /* Y = 200*/
Add 100 to YA;              |  Print sum of SB and YB;
    /* Y = 200*/            |      /* sum = 1100*/
Write YA into Y;            |
   /* Y = 300*/             |
```

Figure 4: Intermediate Data Retrieval

Let us again assume that instructions on the same line are carried out in parallel. From figure 4 it can be seen that user B read the contents of the bank account while A's fund transfer was still in progress and before it was completed. User B therefore retrieved an erroneous version of the contents of the account even though the final value placed into the bank account by A was correct.

There are many circumstances in which interference between user update operations could occur. The examples given here simply illustrate two of the many situations which could arise.

User interference due to concurrent update operations could leave stored data in an inconsistent state or give an incorrect view of data to the user. The use of proper concurrency control mechanisms help eliminate such anomalies.

### 2.3.2. Component Faults

We consider a distributed system to consist of five basic components namely storage, processors, communication, software and users. This section discusses faults which affect these components and how component faults affect stored data.

#### Storage Faults

By storage, we mean secondary storage such as disk storage devices. (Volatile storage will be considered to be part of a processor.) Typical commands to a disk storage device would be to (1) read data from disk and (2) write data onto disk. There are also the issues of addressing and locating data on disk storage devices.

Unfortunately, it is not always possible for operations on disk storage devices to succeed. The physical storage medium could decay, making retrieval of data impossible. A disk could suffer a head crash caused by mechanical or electromagnetic faults in a disk arm or drive, thereby destroying user data. Transient errors caused by electromagnetic fluctuation could temporarily alter the state of stored data. These faults make disk storage insufficiently reliable for storing critical data. A mechanism which provides tolerance for storage faults is discussed in chapter 3.

## Processor Faults

Faults in a processor lead to processor failures. A processor may fail in two ways. The first is often referred to as a processor crash. When a processor crash occurs, the processor stops processing and the contents of volatile storage are lost. The loss of information in volatile storage could lead to inconsistency in long term data which is stored on disk storage devices. Consider our electronic fund transfer example of figure 4. In A's algorithm, a processor crash could occur after the "write SA into S" instruction and before the "write YA into Y' instruction is executed. The system could attempt to naively recover from the processor crash without ever executing the rest of the bank transfer statements. The value of the bank account which is stored on disk would then be erroneous.

The other mode of processor failure is often regarded as malicious failure of a processor. In this case, the processor does not stop but continues operation although its behaviour is unpredictable and could be malicious in the sense that faulty components within the processor continue to provide unspecified service. Such malicious behaviour could destroy stored data by writing invalid data to storage or overwriting already stored data. NMR schemes such as the triple-modular redundancy scheme[Lyons62] can be used to mask such malicious hardware failures. This approach involves the use of several replicated hardware modules and voting on their results so that the majority result can be chosen. Schneider[Schneider83] also describes an abstraction called "fail-stop" processor which provides tolerance for such processor failures.

**Communication Faults**

The communication subsystem delivers messages between users. There are two standard operations on messages namely, "send" and "receive". Figure 5 shows the various possible outcomes when messages are sent.

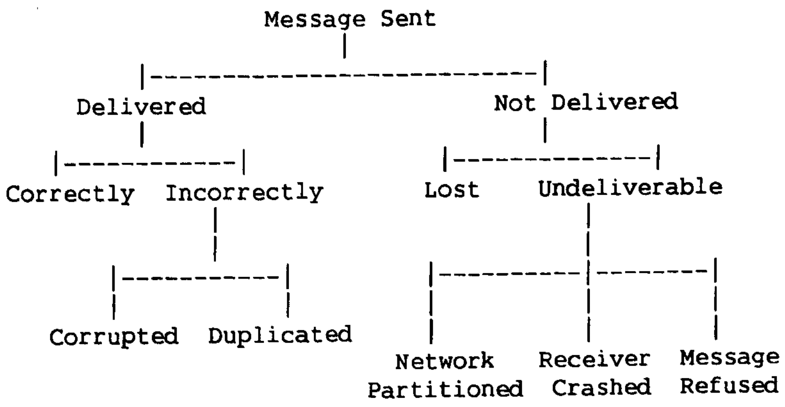

Figure 5: Possible Outcomes for Messages

The effect on stored data of incorrectly delivered messages or undelivered messages is that either bad data is written into storage or data which is expected to be written into storage is not written. Unless such faults are detected, stored data will be left in an inconsistent state. Remote procedure call mechanisms and similar protocols are used to ensure that messages are properly delivered. Algorithms for error detection and message retransmission schemes are also used to tolerate some adverse effects of communication failures. The failure of the communications network, followed by continued processing in multiple partitions makes it difficult to maintain the mutual consistency of replicated copies of resources in a distributed system. Communication failure also prevents the completion of distributed tasks by making it impossible for the various parts of a distributed task running on different machines to be properly synchronised. Algorithms for handling network partitioning address these issues.

## Software Faults

Software constitutes a major proportion of the complexity in computer systems. Thus although it is usually assumed (rightly or wrongly) that the hardware is correctly designed it is extremely difficult to guarantee that a complex software system does not contain design faults. Such a guarantee requires the ability to enumerate all possible failures modes (if any) of the software modules and all possible interactions of such modules. This is very difficult even for a reasonably small software subsystem. In the presence of software faults, the behaviour of a computing system is unpredictable and data can be damaged. Fault tolerance for unexpected software faults is difficult to provide. The problem, as Randell[Randell78] pointed out, is "how to tolerate faults which are unanticipated and unanticipatable". Two approaches, namely the recovery block scheme and N-version programming, have been developed for providing tolerance for such faults. These schemes will be discussed later.

### User Faults

It can be said that human beings constitute the most unpredictable component of a computer system. However, the user faults which we are concerned with are those related to the submission of wrong input data and operator faults (such as mounting the wrong tape on a tape drive). When wrong data is not detected, a computer system makes correct state transitions but produces outputs which are not consistent with the system specification. This is because the system started in an erroneous initial state.

Error detection mechanisms can sometimes detect invalid data at the interface between two interacting components. This is often referred to as interface checking. There is no guarantee that wrong input data can always be detected by such checks. Propagation of such data could result in wrong data being written into a data storage system and thereby lead to overall system failure. By this we mean that the system of which the computer and the user are components, fails.

### 2.3.3. Unsynchronised Update of Replicated Data

Technological advances in communication network design have led to the development of operating systems in which user data is distributed around a network. In such systems, it is not unusual for a user in one machine to access data on other machines in the network or to have a task distributed among several machines. However data required by a user may not be accessible due to a network failure. To cope with this problem, many systems replicate data on different machines, so that if the machine which stores the original copy of user data is not available due to node or network failure, any accessible replica can be used instead.

However, the replication of data could result in mutual inconsistency between replicated objects. It is necessary that the mutual consistency of replicas be maintained. Some algorithms designed for this purpose are described in [Ellis77], [Mullery75], [Thomas78]. However, maintaining such consistency in the presence of network failures is difficult due to the inability to synchronise the autonomous update operations on replicated objects in non-communicating sites. Mutual inconsistency between replicas could result in users obtaining a view of data which is not consistent with the system specification.

## 2.4. MEASURES FOR MAINTAINING DATA CONSISTENCY

This section discusses some software fault tolerance measures and techniques which are used to maintain the consistency of data in spite of the sources of unreliability which have been discussed in the previous sections. Measures and mechanisms which are used to maintain data consistency in the presence of (1) Concurrent accesses to shared data, (2) Component Faults, and (3) Replicated Data will be discussed.

### 2.4.1. Shared Data Access Control

One of the techniques used in maintaining the consistency of data is the control of access to shared data. Many algorithms exist for synchronising accesses to such data so as to avoid interference between user updates. Most of these algorithms have been proposed as concurrency control schemes [Bernstein81], [Rosenkrantz78], [Papadimitiou84], [Bhargava82].

Concurrency control is concerned with ensuring that potentially interfering user accesses do not leave stored data in an inconsistent state. Bhargava[Bhargava82] outlined the three basic approaches to the design of concurrency control algorithms as:

(1) The use of the "wait" synchronization technique. This requires that a task needing to access a resource has first to acquire it exclusively. All other tasks which need the resource wait until the resource is released. One approach to implementing a "wait" synchronization method is to implement a locking scheme [Eswaran76], [Schlageter76], [Stonebraker79 ]. A locking scheme locks any resource in use by a task and releases it when it is no longer needed by the task. There are various different locking strategies, one of the best known being two-phase locking (2PL). The major disadvantages of locking strategies are the bottlenecks created by tasks waiting for resources and the reduction in concurrency resulting from the use of such schemes.

(2) The use of timestamps, circulating tokens and tickets [LeLann78], [Bernstein80] are approaches which can be mapped onto locking strategies. These approaches order the activities of a system by allowing only the task which has the highest (or lowest) timestamp, or which is in possession of the ticket or

token, to access shared resources. These schemes differ from locking strategies in their method of ordering activities.

(3) The use of backout strategies is often referred to as the optimistic approach to concurrency control[Kung82]. Tasks are allowed to proceed freely with respect to accesses to shared data despite the risk of errors being introduced. Before update operations are committed, conflicts are detected and resolved. In resolving conflicts, recovery strategies such as those which are discussed in the next section are used. The operations of some tasks may have to be undone in this resolution process.

### 2.4.2. Software Approaches for Tolerating Component Faults

This section discusses some software approaches for tolerating component faults. It does not deal with each component fault individually but instead discusses general strategies and mechanisms which can be used to tolerate such faults. Recovery from errors constitutes the major approach to the maintenance of the consistency of data. Different strategies are used depending on whether faults are anticipated or unanticipated.

#### 2.4.2.1. Fault Tolerance Strategies

Forward and backward error recovery schemes constitute the two main software fault tolerance approaches for tolerating anticipated and unanticipated faults, respectively.

**Forward Error Recovery**

Fault tolerance for anticipated software faults is generally provided by the use of forward error recovery techniques, such as exception handling schemes [Goodenough75], [Cristian82]. These schemes aim at detecting and correcting errors before these errors can cause extensive damage to data. They provide efficient and specialised recovery facilities. Such scheme are not designed to work when unexpected faults occur.

Reliability in software systems requires that programs give a well-defined response to all requests for service. Detecting and coping with unusual conditions (even though expected) makes programs complicated and unreadable. An exception handling mechanism provides a linguistic facility which permits a more organised method of detecting and coping with errors. Let us refer to the processing which should be carried out when an expected error (such as overflow in an arithmetic computation) occurs as abnormal processing. Exception handling mechanisms provide a clear separation between the normal and abnormal processing performed by a software component, thereby making the system simpler, more readable and therefore more reliable. A number of recent systems (such as [Liskov79], [Defence80], [Liskov83]) incorporate exception handling facilities.

**Backward Error Recovery**

Fault tolerance for unanticipated software faults usually aims at restoring a faulty system to an earlier state which is presumed to be "error-free". It does not aim at detecting specific errors, since it does not know what faults to expect and hence what errors might have been generated. It only needs to know that something has gone wrong.

One of the ways of coping with such faults is the use of backward error recovery schemes. This approach requires that state information (recovery data) be saved periodically. In the event of an error, a system is restored to a previous error-free state and restarted from that state in the hope that the state preceded the occurrence of the fault. A thorough survey of recovery techniques which are based on the use of state restoration has been carried out by Verhofstad [Verhofstad78].

### 2.4.2.2. Fault Tolerance Mechanisms

This section discusses some of the mechanisms which are used in software systems for fault tolerance purposes.

## Recovery Block Scheme

The recovery block scheme [Horning74], [Randell75] provides a means of error detection and recovery for unanticipated software errors. It uses redundant software modules such that when one module fails, control is passed to an "alternate". Errors are detected by the use of an acceptance test which tests the outputs produced by a module with respect to the system specification. In the event of an error, a system is restored to a previous "error-free" state (recovery point) before control is passed to an alternate module. The modules in a recovery block scheme should ideally not be "fault related". By this we mean that such modules are expected to be independently developed so as to minimise the risk of common faults. This scheme provides an effective means of handling unanticipated software errors and for providing continued service in the presence of such errors.

## N-Version Programming

Another mechanism which can be used to tolerate unanticipated faults in software systems is the use of N-Version programming[Avizienis77]. In this scheme, N different versions of a program are executed at the same time. Their results are compared and the result which is submitted by the majority of the versions is chosen. This scheme can tolerate any unanticipated fault which does not violate the N-Version control structure. However, if the majority of the program versions are in error, the erroneous result will be chosen.

## Atomic Transactions

Gray[Gray81a...] defines a transaction as a set of Read and Write commands with the following properties:

(i) Consistency: Transactions only make correct state transitions.

(ii) Atomicity: Either all of a transaction's commands are carried out or none is.

(iii) Durability: The effects of a completed transaction cannot be abrogated.

Transactions can be used as units of consistency and recovery. The atomic property of transactions is used to keep stored data consistent. Maintenance of data consistency by transactions is based on the use of backward error recovery. A transaction aims at carrying out all update operations or none of them. It therefore provides UNDO capabilities so as to eliminate the effects of partially completed operations. Transaction mechanisms are typically provided for database systems, but the concept can apply to any form of data storage system.

A transaction transforms an already consistent database to a consistent database since the effects of uncompleted operations are undone. One of the limitations of the transaction scheme is that a considerable amount of work might have to be undone if the transaction cannot be completed. This is particularly undesirable in long-lived transactions where days or possibly months of work might be undone.

Transactions and other schemes which use backward error recovery can be combined with forward error recovery schemes so as to provide a highly reliable and consistent computer system. The use of forward error recovery helps minimise the probability of system failures by providing fault tolerance for specific faults. This in turn reduces the frequency of "UNDO" operations required in a system.

The construction of a stable storage system is an example of the use of forward error recovery and a limited form of backward error recovery to provide a disk storage system which is resilient to (it is hoped) all faults affecting disk storage. It specifically tolerates anticipated disk faults and provides a crash recovery routine which restores a disk storage system to a consistent state after a processor crash. The use of such a crash recovery routine can be regarded as the use of a limited form of backward error recovery. The design and implementation of a stable storage system will be discussed in detail in chapter 3.

### 2.4.3. Maintaining Mutual Consistency of Replicas

The last two sections discussed measures and mechanisms for maintaining the consistency of data in the presence of concurrent access to shared data and component faults. This section considers the use of mutual consistency algorithms to maintain the consistency of stored data. Mutual consistency algorithms aim at providing tolerance for faults which arise due to the use of data replication. Algorithms for detecting mutual inconsistency among replicas in the presence of network partitioning are described in [Parker83], [Brereton83]. These papers discuss resolution of inconsistency only for simple files whose operation semantics are known, such as directories and mail-boxes. Assuming that inconsistencies have been detected, Wright[Wright83] presents an algorithm for merging a partitioned database for the general object type. However this algorithm requires that the readset and writeset of an application be known beforehand so that the activities of an application can be grouped into transaction classes. It could be difficult to use such a scheme for applications which do not know a priori all their resource requirements. A more detailed discussion of these issues is given in chapter 5. That chapter also presents an algorithm which handles the detection and resolution of inconsistencies among replicates in a partitioned data storage system.

## 2.5. CONCLUSION

This chapter has discussed the major causes of data inconsistency in distributed systems and summarised the approaches which are used to maintain consistency in the presence of faults. It classified methods for maintaining the consistency of data as consisting of three basic approaches, namely, methods for maintaining consistency in the presence of

(i) Concurrent accesses to shared data

(ii) Faults and errors

(iii) Replicated resources.

It discussed various strategies, abstractions and mechanisms which are used to keep stored data consistent.

# 3. A RELIABLE STABLE STORAGE SYSTEM FOR UNIX

## 3.1. INTRODUCTION

This chapter describes the first of the three stages in our data storage construction. It 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 UNIX environment in spite of various types of hardware faults. 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. Internally the implementation 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.

The problem 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 [Lampson79] is now accepted as one of the ways of maintaining the internal consistency of data which is stored on disk storage devices 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. Our implementation uses the stable storage mechanism to provide a reliable repository for data which tolerates disk faults through the provision of stable disks. It also provides crash recovery facilities for the data which is stored on such disks. Stable disks have the same actions as ordinary disk storage but with the property that, to a very high degree of probability, no anticipated adverse 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) by implementing such a system as a collection of user processes. (An alternative approach is described in section 3.7.) The implementation which is described here 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" in such a way 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 the way its internal design is based on a scheme for 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 [Lampson79], [Svoboda81], [Sturgis80]. 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 supports atomic reading and writing 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[Brownbridge82] which consists of a number of PDP/11 computer systems connected by a Cambridge Ring local area network. The facilities described in this chapter are part of a prototype reliability subsystem associated with the Newcastle Connection software of our UNIX United system. The implementation technique used has been inspired by the system design methodology for fault tolerant systems developed by Cristian[Cristian83]. An exception handling software package described by Lee[Lee83] 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 3.3 we describe the reliability issues which the stable storage system addresses. Section 3.4 presents the fault tolerance design notation which is used to structure 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 3.5 and section 3.6 contains some performance measurements. In section 3.7 we discuss an alternative approach for implementing stable files in UNIX. Section 3.8 provides some concluding remarks. (An example of how the fault tolerance design notation was implemented is given in an appendix to this thesis.)

## 3.2. OVERVIEW OF THE SYSTEM

This section gives a brief overview of the UNIX United distributed system[Brownbridge82] 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. Such a system 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.) Figure 6 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     \
   U2     / \     U5(system providing
  /  \        \    / \  stable storage)
 /    \           /   \
/                sf   sf
             (stable files)
```

Figure 6. 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 of 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 figure 6 it follows that a user process logged on to 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[Shrivastava82].

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, for convenience, 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 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 checked to distinguish between ordinary and stable files. This check would have to be incorporated in the "interception code" within each Newcastle Connection layer rather than in the UNIX servers. 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 aim was to investigate the design of stable storage systems themselves and to provide a stable storage server facility for computers in a UNIX United system.

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 figure 7.

```
 ----------------------                     ----------------------
|user programs, non    |                   |user programs, non    |
|resident UNIX software|                   |resident UNIX software|
|                      |                   |                      |
|----------------------|remote             |----------------------|
|Newcastle Connection  |<----------------->|Newcastle Connection  |
|        .-------------|procedure call     |        .-------------|
|        |stable storage|                  |        |stable storage|
|----------------------|                   |----------------------|
|                      |                   |                      |
|      UNIX Kernel     |                   |      UNIX Kernel     |
|                      |                   |                      |
 ----------------------                     ----------------------

          UNIX1                                      UNIX2
```

Figure 7: Software Subsystem Relationship

## 3.3. RELIABILITY ISSUES

Various faults can prevent a disk storage system from providing reliable service. A physical disk storage system 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 data from a disk, and also 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, read/write operations 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. The processor is also expected to stop its processing activities.

(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 of disk storage devices.

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

## 3.4. CONCEPTS AND NOTATION

The need to provide well-structured and effective fault tolerance facilities[Anderson81] 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 chapter including the fault tolerance notation and its implementation.

Our concept of an exception occurrence is based on those of Melliar-Smith and Randell[Melliar-Smith77] and Cristian[Cristian82]. The service which a procedure or component is intended to provide is implemented by a sequence of internal state transitions. This intended service can be specified by a binary relation 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

$$(s', s) \in INV$$

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

$$(s', s) \notin INV.$$

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 has been provided. If no handler was provided, the exception is propagated to the the enclosing exception context, and then 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.

### 3.4.1 Notation

The notation which we use to describe the stable storage system is an adaptation of the notation used by Cristian[Cristian83]. Suppose c is a command or procedure which may signal a set E of exceptions. Then one can give the declaration:

```
proc c SIGNALS E    (PD)
```

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 $(s',s) \in INV$ 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

```
c[E::H]             (EX)
```

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 (repetition) construct

```
(N)c[OTHERS::OH; E::H]          (R)
```

will be used as an abbreviation of the n-depth repetition

```
c[OTHERS::OH; E::c[...c[OTHERS::OH; E::H]]]
```

The semantics of the R construct are as follows: Suppose there is a special exception called OTHERS which is an element of E, and a handler 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 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 (that is not OTHERS), 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.

## 3.4.2. Implementation of Notation

The fault tolerance notation was implemented using the exception software package described by Lee[Lee83]. 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 Figure 8.

```
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. 8.

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[Lee83], which describes the package fully.

## 3.5. STABLE STORAGE IMPLEMENTATION

Several 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 the effects of 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.

### 3.5.1. Stable Files

A stable file looks to the user just 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 stored on the same disk storage device if there were any means of ensuring that the files are not "decay related"). A stable file in our environment is read and written using standard UNIX system calls. A stable read operation reads from the first file and if that read operation is not successful, it reads from the second file. A stable write operation writes to each of the pair of files. The details of these operations are given in section 3.5.2.

The problem we addressed was to provide the user with variable length atomic 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, unless 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 however 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 and the other is accessible and contains valid data.

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 assumed to be in one of the following states.

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

(ii) One of the objects has valid data and the other is detectably bad.

(iii) One of the objects is valid and the other is invalid but it is not known which object contains valid data.

In case (ii) a crash occurred while the disk write heads were turned on thereby corrupting the data which is stored on the disk. The data is detectably 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 consistent or is detectably bad. 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 suitable in environments where variable length objects need to be written atomically. 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 write operation. A timestamp would indicate to us the write operations that belong together but will not necessarily enable us to determine which write operation was completed and which was interrupted. One approach to timestamping every update operation would mean having timestamps scattered in the file. We decided that this was undesirable since we regard a file to be a data entity which has a meaning to the user. Alternatively, these time stamps could be kept transparent to the user by storing them in another file which in turn has to be made stable. This would increase the number of accesses to a disk storage device and hence the time spent in carrying out each stable operation. The overhead was considered to be excessive so this approach 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 unsuitable for our purposes because sometimes the two files representing a stable file are both consistent and the file which does not hold the token may be falsely considered inconsistent.

We used instead what we called a "moving tag" to solve this problem. A "moving tag" is a concatenation of any small set of characters such that the resulting string is assumed (as is the case with "end of file characters") 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 write operation would overwrite the tag written by the previous write operation while writing its data. The tag is therefore in effect removed by each logical write operation and always reinstated at the end of file. It was the removal and reinstatement of the tag that led to the name "the moving 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 processor 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 the stable file.

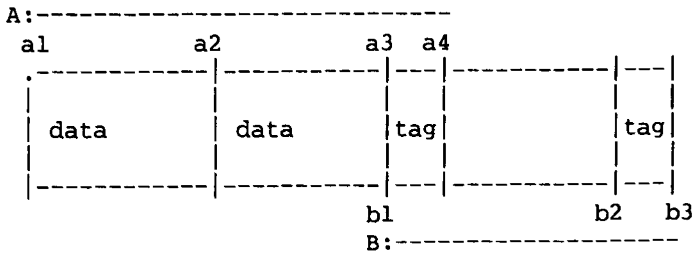

Figure 9: Two logical write requests.

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. Figure 9 shows two write system calls issued by a user. The first request, which we call the A-write, is to write 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 write request (called the B-write) is to write 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-write overwrites the tag written by the A-write by starting its write operation at a3 instead of a4.

Let us consider the effects on figure 9 of a fault such as a processor crash which could interrupt a write operation. The problem is to ensure that the A-write and the B-write, which are variable length stable write operations, are atomic. The following are the various scenarios when the operations in figure 9 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, the stable file must 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 confirm its completion. The inconsistency will be clearly indicated by the absence of the tag. In such a case, the crash recovery routine (which is invoked by the system manager after a crash) will restore the consistency of the stable file. If a crash occurs before a1 or after a4 or b3, the stable file will be considered consistent since each logical write operation has either not been started or 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 section 3.5.2.2.

This scheme is most efficient when the stable file is an append-only data structure. Supporting random access write operations would require writing the data at a specified address and then writing the tag at the end of the file. This extension is trivial but would increase the time which is spent in carrying out a write operation. Another observation is that most disk hardware systems 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 write operation.

### 3.5.2. Stable Operations

We shall now describe the implementation of stable operations using the notation of section 3.4.1. An example of how this notation was implemented using the exception handling package described in section 3.4.2 is given in an appendix to this thesis. The stable operations are organised as two levels of abstraction called the Transient and Stable layers.

#### 3.5.2.1. 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.

```
Proc Tread(fd:int; buf:array[...] of char; nbytes:int) SIGNALS
                                                         RD-FAIL;
RD-FAIL, DISKERR, OTHERS:exception;
var fd: int;
begin
    (N)read(fd,buf,nbytes)[OTHERS::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 an OTHERS 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.

```
Proc Twrite(fd:int; buf:array[...] of char; nbytes:int)SIGNALS
                                                       WRT-FAIL;
OTHERS,DISKERR,WRT-FAIL:exception;
var fd: int;
begin
    (N)write read(fd,buf,nbytes)[OTHERS::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.

#### 3.5.2.2. 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 to read from and write 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. 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 appropriate Sread and Swrite operation of the stable storage system.

```
Proc Sread(fdp:int; buf:array[...] of char; 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 ensure that failures of the stable read/write operations rarely occur.

```
Proc Swrite(fdp:int;buf:array[] of char;nbytes:int)SIGNALS
                                                    SWRT-FAIL;
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 exception, SWRT-FAIL. This helps ensure that not more than one disk can be damaged following a crash which occurs during a write operation. However, care must be taken to ensure that when a stable write operation returns, the data has 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"[Ritchie79] or by forcing the system write buffers to be flushed after each write operation. Flushing the system write buffers was found to be expensive and therefore 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*/
ctag1, ctag2: boolean initially false;
              /*consistency status indicator*/
buf1,buf2: array[...] of char;
begin
  /*check for consistency tag */
  if file1 contains a consistency tag, set ctag1 = True;
  if file2 contains a consistency tag, set ctag2 = True;
  if (ctag1 and ctag2 ) 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 stable files. 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 periodically 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 a disk. To perform crash recovery, Crec 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 does not have a tag) 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 on the two files are readable. If a block of data from the first file is not readable (i.e. is 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 can not complete its operations due to the existence of permanent 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 time Tu 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.

#### 3.5.2.3. Other Stable Operations

We also found it necessary to implement a stable version of some other 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[Ritchie79] for more information on these system calls. The stable storage system requires that pairs of disks should be mounted for stable operations. If this requirement is not met stable operations would fail.

## 3.6. PERFORMANCE MEASUREMENTS

Some initial tests were carried out to assess the performance of the stable storage system. These tests were performed on a PDP-11/45 running V7 UNIX and using RK05 and RL01 disks. The aim was to compare the disk access times for ordinary disk storage and stable storage. The time spent in reading and 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 "time" facility was used to obtain time measurements to the nearest millisecond. Table I contains the results from these performance tests. The figures in table I are averages calculated from the results of several experiments.

| UNIX read (sec) | stable read (sec) | UNIX write (sec) | stable write (sec) | block size in bytes | total no. of bytes |
|---|---|---|---|---|---|
| 20.40 | 20.60 | 20.40 | 84.880 | 64 | 50k |
| 10.60 | 10.80 | 10.80 | 44.740 | 128 | 50k |
| 5.60 | 5.80 | 5.80 | 24.600 | 256 | 50k |
| 2.580 | 2.60 | 2.600 | 13.620 | 512 | 50k |
| 1.340 | 1.360 | 1.360 | 13.120 | 1024 | 50k |
| 0.720 | 0.740 | 0.720 | 12.680 | 2048 | 50k |

Table I

Data Access Times for UNIX and Stable read/write Operations

We observed that the time required by the stable read operation is approximately equal to the time required by the UNIX read operation when exceptions are not encountered (the average overhead was roughly 2%). For the write operation, the access time ratio of the stable write operation to the UNIX write operation is approximately 1:4 for block sizes up to 512 bytes. The 1:4 ratio is due largely 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 also 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 generally maintained if the transient layer operations read and write larger blocks of data. In our environment, a weak atomicity of operations which write 512 byte blocks is provided by the operating system therefore blocks which were larger than 512 bytes were fragmented so as to use this atomicity facility.

From these figures it can be seen that applications which perform mostly read operations pay a very small 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 a substantial increase in overheads. Accurate figures on the ratio of read to write system calls are not presently available, so it is difficult to estimate how large an overhead the general use of the stable storage system would impose. Wyeth[Wyeth73], in his simulation of a recursive cache mechanism, analysed the references to shared resources in a set of sequential programs and his figures showed that read operations occurred about three times as frequently as write operations. Applying his results to disk storage would suggest that the proportion of read accesses will generally be fairly high. Figures from table I indicate that there would be a minimal performance degradation of about 80% when the stable storage system is used. This applies to tasks which perform only read and write operations. However, systems differ greatly in the extent to which disk operations dominate the average workload. For most applications where disk operations do not dominate the workload the use of the stable storage system will rarely be noticed on the user level.

## 3.7. AN ALTERNATIVE 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 a stable storage server as an independent facility which is not embedded in the operating system. An alternative approach would be to provide robustness 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 use of the UNIX buffering mechanism which forms the block-device interface. Such an implementation would have the advantages of utilising fully the facilities provided by the UNIX file system. The next chapter addresses these issues and discusses the implementation of a crash resistant file system.

## 3.8. 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 application level. The provision of the abstraction of stable files extends the domain of reliability facilities available to the UNIX user.

# 4. A CRASH RESISTANT UNIX FILE SYSTEM

## 4.1. INTRODUCTION

This chapter describes the second stage in our approach to the construction of a reliable data storage system. It describes the modifications which were made to the UNIX kernel to support a crash resistant file system and the associated recovery facilities for maintaining the internal consistency of files despite the failure of the host processor, transient hardware failures and decay of the storage medium.

Chapter 3 described a stable storage system which provided stable disks as an independent facility which was not embedded in an operating system. The stable storage system was implemented as a set of user processes which meant that there were no modifications to the UNIX kernel. The aim of that implementation was to provide reliable disks for storing user data. The client of such an implementation is the user. However, there is also a need to make the operating system a client to a reliable disk storage system. By this we mean that, as with the user, the operating system has a need to store system information on reliable storage. There is the need, for instance, to make file management information (such as i-nodes in UNIX) crash resistant. In environments where there is the need to provide reliable disk storage facilities which can be used by both the user and the operating system, a crash resistant UNIX file system will have to be implemented.

Various faults could harm the integrity of the data which is stored by a file system. A processor crash is an example of a system failure which could affect the consistency of stored data. Such a failure could lead to the loss of information which was available before the failure occurred. This loss of information would make the data which is stored by the file system inconsistent. Consider the problem of an operating system whose I/O subsystem employs a buffering mechanism so as to reduce the amount of physical I/O operations on disk storage devices. The use of such a mechanism may give rise to delayed write operations. If a processor crash should occur, there could be logically complete but physically incomplete I/O requests in the buffers. These buffered I/O requests will be destroyed. The inconsistency in the file system arises due to the fact that the user's view of the disk is different from the actual disk state. Section 3.7 of the previous chapter commented on the need for a crash resistant file system which is capable of providing reliable file service despite the use of a buffering mechanism by a file system. This chapter describes the implementation of such a crash resistant file system on stable disks. By a crash resistant file system we mean a system whose operations are such that

(i) their effects would survive a crash and

(ii) the effects of an uncompleted crash resistant operation will be undone by a recovery routine.

In order to provide a crash resistant file system it is necessary to provide stable disks as well as to ensure that the activities which are involved in managing files do not introduce data inconsistency. The consistency of a stable storage system does not imply the consistency of a file system which is implemented on it. As has already been observed, the use of a buffering mechanism by a file system could introduce data inconsistency if buffered I/O requests are lost during a processor crash. Some system file management information is not accessible to user processes. Kernel modifications are therefore necessary in order to make such information crash resistant and to ensure that the kernel invokes crash resistant operations for both system and user functions. The implementation described here addresses these issues and makes the necessary kernel modifications needed to make the UNIX file system crash resistant. It also uses a numbering scheme to avoid replication of disks which store short lived data for which crash resistance is not desired, such as swapping devices.

Although the idea of using crash resistant files is attractive, there have not been many accounts describing how to make an already existing file system crash resistant. The proper placement of reliability facilities in an existing system so as to provide effective fault tolerance is not usually clear. The UNIX operating system is widely used for research in operating systems and the use of stable storage is advocated by many workers [Lampson79], [Svoboda81], [Cristian83]. We therefore believe that the implementation of a crash resistant UNIX file system which is based on the use of the abstraction of stable storage will be of interest to people in the area of reliable computing.

Section 4.2 gives a general description of the UNIX file system and its associated I/O subsystem. It examines UNIX file operations and their reliability problems. Section 4.3 describes the implementation of the crash resistant system. Performance measurements and analysis are given in section 4.4 and section 4.5 presents some desired extensions. Concluding remarks are given in section 4.6.

## 4.2. GENERAL DESCRIPTION OF THE UNIX FILE SYSTEM

Our intention is to implement the UNIX file system on stable disks and to make it crash resistant. This section therefore gives a brief description of that file system and its associated I/O subsystem. A reader who is interested in a deeper understanding of the UNIX file system would find the presentation in[Thompson78] on the "UNIX File System Implementation" helpful.

The basic function of a file system is to divide disk storage into units which we call files. Unlike many operating systems which provide flat file systems, UNIX provides a tree-structured hierarchical file system. The nodes in the tree are directories and the leaves are files. A file in UNIX is an unstructured finite sequence of characters. A directory is a file which contains a list of file names and a set of numbers used to access system information on files. Directories are used to impose a hierarchical structure on the file name space since every file or directory (except root) appears in some directory. The base of this tree structure is the root directory.

Files are named in the form of a path name which is a sequence of directory names separated by "/" and ending in a file name. If a name begins with a "/" a search for the file begins in the root directory otherwise the search begins in the user's current working directory. For example, the name "/jo/doc/myfile" requires that the root directory be searched for a directory named "jo" and "jo" is to be searched for the directory "doc". The directory "doc" is then searched for a file which is called "myfile".

A UNIX user sees the file system as a hierarchical structure. However, below this hierarchy of directories is a flat file system implemented by the list of file definitions (i-nodes). These file definitions contain most of the information about files such as the location of a file, length of a file, access mode, owner and creation date. Below the flat file system is the block I/O system which carries out physical I/O operations on devices. After this comes the physical hardware. Fig. 10 shows the relationship between the various subsystems.

```
         Hierarchical File system
        ---------------------------
            Flat File System
        ----------------------------
    Block and Character I/O System
        ----------------------------
            Physical Hardware
```

Figure 10: I/O Subsystems

Most of the reliability problems which we will be discussing occur at the interface between the block I/O system and the flat file system.

### 4.2.1. The Block I/O System

The block I/O system refers to devices that can be addressed in blocks of 512 bytes (or 1024 bytes in some versions of UNIX). This applies mostly to disk and tape devices. Devices that do not fall into the block I/O category such as communication lines and line printers belong to the character I/O system.

The block devices use a buffering mechanism which enables the kernel to reduce physical I/O traffic to these devices. The buffers act as a data cache and are searched whenever a read request is issued. If the desired block is available in the buffers, the data is made available to the user without physical I/O being performed. Blocks which are frequently accessed are retained in the buffers to reduce I/O traffic. A write operation involves acquiring a buffer, filling it with data and carrying out the physical transfer of data between the buffer and the device. The transfer of data between the buffer and the device need not be done in the same order as that in which the requests were issued by the user. System efficiency considerations may also require that this transfer be deferred. The physical transfer of data is then carried out at a later time by the file system. A return from a write system call may therefore take place before the physical I/O operation is carried out.

The asynchronous nature of the algorithms in the block I/O interface make error reporting and error handling difficult.

Secondly, altering the physical I/O sequence from that of the logical I/O sequence requested by the user could sometimes result in data being written in the wrong order. The adverse effects of processor crashes on the write operation which is implemented by this interface has already been mentioned. These problems led to a desire to implement crash resistant file operations which use redundant disk hardware (i.e. stable disks) to provide a more reliable file service.

## 4.3. THE CRASH RESISTANT SYSTEM

A number of changes were made to the UNIX kernel to support crash resistant file operations.

(i) The device driver software was modified to incorporate management of replicated disks in the form of stable disks.

(ii) The UNIX buffering mechanism was modified so as to provide a more reliable write operation and

(iii) A crash recovery routine is provided on the user level

The aim was to implement file operations such that the effects of a completed operation would always survive a crash. In a reliable file system, the acknowledgement that a user would expect for a write request is that the data has been received and has been safely stored (in stable storage). This is in contrast to an ordinary file system which could simply acknowledge the receipt of the data. Such a response might not be adequate for a user of a reliable file system. Rather, there is a need to know that the request has been acted upon. To achieve these objectives, each operation once started must wait for the physical I/O to be completed before returning to the caller. Replication of disks is used to ensure that if an operation is interrupted by a crash, an earlier consistent disk state will be available for crash recovery purposes. The crash recovery routine is invoked to restore the consistency of disk data structures before the system restarts normal operation after a processor crash had occurred.

The operations of the crash resistant file system are implemented on three levels of abstraction: the disk level, the file level and the user level. The disk and file levels are embedded in the UNIX kernel. The disk level implements the abstraction of stable disks. The second level, namely the file level, implements the UNIX file system concepts (with its associated i-nodes and directory system) on the stable disks provided by the disk level. A crash recovery operation is provided on the user level. The following subsections discuss the details of each of the three levels.

### 4.3.1. Disk Level: Implementation of Stable Disks

The description of a stable storage system has been given in chapter 3. That chapter described an implementation of the stable storage system as a set of user processes. This section describes how the stable storage mechanism was reimplemented in the UNIX kernel. It was necessary to first provide stable operations in the kernel and then to interface the UNIX file system with stable disks so as to provide a crash resistant file system. Unlike the implementation in chapter 3 the provision of these facilities in the kernel ensures that these operations will be invoked by the kernel for both user data and system management information thereby providing stability for both types of data. Provision of these facilities required kernel modifications. The operations implemented by a user-level stable storage system such as those of chapter 3 are not expected to be invoked by the kernel when writing system data. The aim there was to provide stability for user data only and no kernel modification was necessary. Also, the provision of a stable storage system is only a part of what is needed in order to support a crash resistant file system.

Each stable disk which is implemented by the disk level consists of a pair of conventional disks which are numbered by a sequential pair of even and odd integers. Operations on this level are atomic. This means that they are either successfully completed or the effects of such operations are undone by a crash recovery routine. A write operation to a stable disk first writes the information to the even numbered disk and then to the odd numbered disk. The following program implements read and write operations for the disk level.

```
Writed (  )
begin
  acquire a free buffer;
  if buffer pointer is pointing to even device then
  write to even device and to the next odd device
  else write to odd device only;
end.
```

```
Readd (  )
begin
   acquire a free buffer;
   read from the even device;
   if the even device is unreadable
   read from the next odd device;
end.
```

We have omitted error reporting aspects of the read and write routines for the purposes of readability. Both of these routines write error reports to say which disks can not be read or written. Also, if the write operation to the even numbered device was not successful, the write operation to the odd numbered device would not be initiated. In such circumstances, error reports would be written and crash recovery would be invoked. From the program it can be seen that once an even numbered device is written the associated odd numbered device will also be written with the same information. If the pointer is already pointing to an odd numbered device (this is the case with non stable disks, example, swapping devices), only that device is written. All disks which do not require crash resistant capability are treated as odd numbered devices. Once such a device is written, there is no write operation to a second disk. The modifications to the read operation were to enable a read action from a second disk when an attempt to read from the first disk fails. It also incorporates a more comprehensive error reporting capability.

The actual modification to existing kernel code and the addition of new statements constituted less than a page of C programming language statements. Basically, we used the routines which were already available. We substituted existing routines whose behaviour had characteristics which we found desirable for routines which we did not consider suitable for our needs. For example, a write routine which waited for physical I/O to be completed before returning was substituted for write routines which carried out asynchronous I/O and which did not wait for the completion of physical I/O. In order to incorporate the management of replicated disks, modification to existing statements was required. However, there is no substantial difference in size between the modified file system and the standard UNIX file system.

### 4.3.2. File Level: Implementation of Crash Resistant File Operations

The previous level implements stable disks as uninterpreted streams of bytes. The concept of a file is not known on that level. The file level implements the UNIX file concepts, utilising stable disks. Thus, a crash resistant file system would be constructed from a set of ordered pairs of "even, odd" disk storage devices numbered (<even1, odd1>, <even2, odd2>, ...). The aim of this level is to help ensure that the activities involved in managing a file system do not introduce data inconsistency. The UNIX file location and file accessing mechanisms were not modified. The main modifications made on this level were to ensure that physical I/O operations on behalf of a file are carried out before a return from a write system call is made. This is to ensure that a processor crash would not distort a users' view of the disk and that such a view would be consistent with the actual disk state. This level also interfaces file operations to those of a stable disk so that all files are automatically replicated on the two disks that make up a stable disk.

### 4.3.3. Crash Recovery

It is assumed that both disks that make up a stable disk will not be damaged at the same time. The file system therefore ensures that the crash recovery routine is invoked more frequently than the estimated mean time between failure (MTBF) of an individual disk. Unlike chapter 3, the crash resistant file system has error reporting schemes inside the kernel so as to enable the crash recovery routine to know which disk was being written when a crash occurred. Crash recovery here consists of copying one of the disks to the other. The preference is to copy the first disk to the second disk so that the most recent update can be reflected. If the first disk is not readable or was being written when a crash occurred the second disk is copied to the first.

## 4.4. PERFORMANCE

Initial tests were carried out so as to gain some insight into the performance characteristics of the crash resistant file system. Our goal was to measure and compare its performance with that of the conventional UNIX file system. Such measurements would enable us to know the kind of overhead a user incurs when using crash resistant data. Our discussion will concentrate on the write operation since that operation is most affected by our modifications. Four kinds of experiments were carried out. The measurements were again performed on a PDP-11/45 machine running V7 UNIX and RL02 disks were used. The "time" and "times" facility of UNIX were used to obtain timing measurements. The first two sets of experiments were designed to measure the effects on a file system when crash resistance facilities are incorporated. They were therefore carried out in single-user environments so as to avoid multi-user interference. The last two sets of experiments attempt to measure the same effects but in the face of multi-user interaction.

## 4.4.1. Single-User Environments

The first set of experiments measured the rate at which each file system wrote 6k bytes of data from and to user files. Several measurements were performed using different block sizes. The results of the experiments are shown in figure 11.

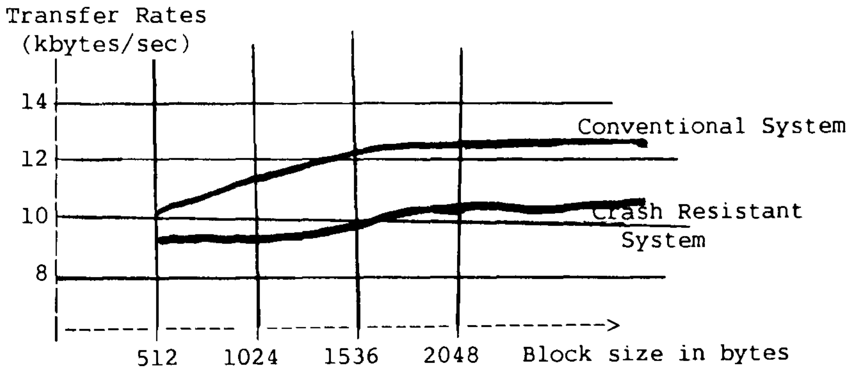

Figure 11: Data Transfer Rates for Write Operation

Measurements indicate a minimum transfer rate of about eight kilobytes for each of the two file systems. The transfer rate increases as the block size increases. This is because less time is spent in making system calls when larger block sizes are used than when smaller block sizes are used to write the same amount of data.

The crash resistant file system writes to two disks so a 50% degradation might be expected. However, effective placement of fault tolerance facilities has enabled this to be kept to between 10% to 25%. Suppose that we represent the time which is spent during a write operation by Wop = Wr + Ov. ("Wr" is the time which is spent on actual physical write operation and "Ov" is the time which is spent on calculating disk space allocations, carrying out required swapping activities, transferring data from user buffer to system buffer and carrying out any other file management activities that are necessary). The "Wr" factor is usually small compared with the "Ov" factor. Since a crash resistant operation writes to two disks, it could be expected that the time spent in carrying out a write operation would be Wres = 2Wr + 2Ov. However, our implementation avoided a recalculation of disk space allocation and other file management activities so that Wres = 2Wr + Ov. A repetition of the "Ov" factor which is responsible for most of the overhead in a write operation was avoided. Instead the same calculated parameters are used to write to both disks thereby resulting in a significant reduction in the expected performance degradation. Some performance degradation should in our opinion, be acceptable if high reliability is required.

A second set of experiments used a task which represents an important class of typical UNIX file system activity (such as a task with more read than write requests). The speed of the file systems was measured by executing this typical UNIX task on the benchmarked file systems and measuring the time which was used to complete the task for different file sizes. Again based on the results of the analysis which were made by Wyeth[Wyeth73] we chose a task with a 3:1 ratio of read to write operations. The results of the measurements were used to calculate the slowdown measure of the modified file system by

$$\text{Slowdown Measure} = \frac{\text{Time spent by Our System} - \text{Time spent by ordinary UNIX}}{\text{Time spent by ordinary UNIX}}$$

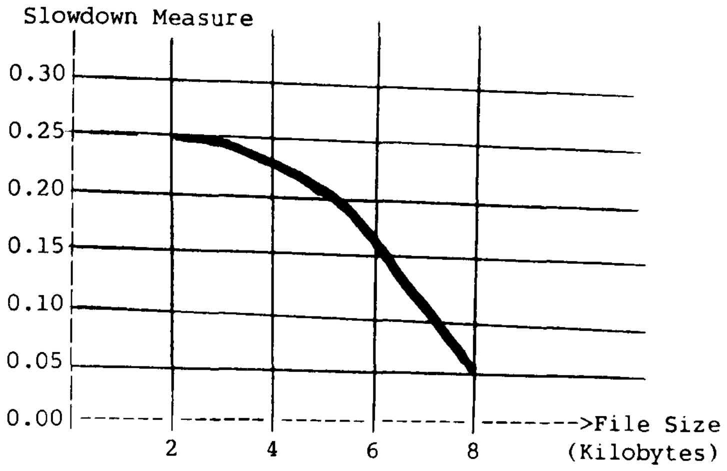

Figure 12: Slowdown Measure for Typical UNIX Task

As the file size increases, the number of I/O operations required to access the entire file increases. In such circumstances, the read operation would usually dominate the I/O accesses and the slowdown measure will decrease as can be seen in the graph of figure 12. A typical application could thus be expected to run with not more than 25% performance degradation on the average when the crash resistant UNIX file system is used. As would be expected, this implementation has a better performance characteristic than the stable storage implementation of chapter 3. This is because unlike that implementation, the management of replicated disks was controlled from within the kernel by this implementation. As has already been observed, this resulted in the reduction of the "Ov" factor.

## 4.4.2. Multi-User Environments

The tests described so far were carried out for single-user environments. The results indicate that performance degradation is acceptable. It is desirable, however, to know the performance of this file system in the presence of multi-user interactions. The third and fourth set of tests were designed for this purpose.

When users interact, there is the possibility that the operating system would carry out asynchronous I/O on behalf of the users. The "elapsed" time for user A say, might include the time which is spent in carrying out part of user B's request. The third set of tests were designed to measure the actual time (exclusive of asynchronous I/O times for other processes) spent by each file system in writing a specified number of bytes to a user file. The "user" and "sys" times, which represent the time spent outside and inside the kernel, respectively, on behalf of a user process, were measured. Three users were allowed to write 5k bytes of data to user files and time measurements were recorded. Figure 13 shows the result of six test runs.

| User1 | | User2 | | User3 | | Total Time | |
|---|---|---|---|---|---|---|---|
| UNIX | Crash Resistnt | UNIX | Crash Resistnt | UNIX | Crash Resistnt | UNIX ** | Crash Resistnt ** |
| 0.08 | 0.06 | 0.04 | 0.04 | 0.06 | 0.06 | 0.18 | 0.16 |
| 0.04 | 0.04 | 0.06 | 0.10 | 0.04 | 0.06 | 0.14 | 0.20 |
| 0.06 | 0.06 | 0.10 | 0.08 | 0.08 | 0.04 | 0.24 | 0.18 |
| 0.08 | 0.06 | 0.12 | 0.08 | 0.06 | 0.06 | 0.26 | 0.20 |
| 0.06 | 0.04 | 0.04 | 0.06 | 0.06 | 0.10 | 0.16 | 0.20 |
| 0.04 | 0.06 | 0.04 | 0.04 | 0.02 | 0.04 | 0.10 | 0.14 |

Figure 13: Multiple Users Processing Time: Write Operation

Time measurements given for each user in figure 13 are the sum of the "user" and "sys" times. The figures in the starred columns are not measured values. They are calculated by summing the appropriate values in each row. Taking the average of the six runs in these figures, it can be seen that the operating system spends a total of about 0.18 seconds for the three users in each of the two file systems. (It is relevant to point out that measurements for "user" and "sys" times in UNIX are highly approximate.) However, overheads were sufficiently small as not to affect these measurements.

Though the operating systems spent approximately the same time for a task in both file systems, the user process in the crash resistant file system is blocked until acknowledgement from a write operation is received. This is to ensure that the effects of a completed operation can not be abrogated by a processor crash. It would be useful to know the kind of delays which a user of a crash resistant file system would expect in a multi-user environment. However, measurements to determine process "blocked times" are very unreliable since they depend heavily on the load on the computer system at any particular time. Elapsed time measurements for file access rates in such environments are sometimes difficult to account for. Unlike the previous measurements, such measurements would include time spent for other processes in an asynchronous I/O environment. Memory contention experienced by the processor and swapping activities are some of the determining factors.

Some experiments were carried out for a three user system. Since the results which are obtained from such measurements are highly load dependent, figures are not very meaningful. However, the results show between a 3:1 to 5:1 ratio in UNIX: crash resistant file system performance as measured by the "elapsed time". This is in contrast to a single user system which showed a negligible difference in performance. The use of a crash resistant file system in a multi-user environment could therefore contribute to a substantial overhead depending on the load on a computer system. Some ways of reducing this overhead are discussed in the next section.

## 4.5. EXTENSIONS

Some optimisation to the existing algorithms can be made. A desirable refinement would be to help reduce process "blocked times" when a crash resistant file system is used in a multi-user environment. The major cause of the "apparent" (not real with respect to overall processor and disk utilisation) performance degradation is the requirement that an acknowledgement for a write operation must be delayed until physical I/O is completed. In transaction-based systems it might not be necessary to ensure the crash resistance of operations until commit time. If a crash occurs, only uncommitted updates will be lost and this would be acceptable. File operations in such environments should support fault tolerance facilities for storage decays, disk crashes and transient I/O faults but not against processor crashes. A "make-cr" (make crash resistant) primitive could be provided to be invoked at transaction commit time. This primitive would ensure that all outstanding write operations on behalf of a user process are written to stable storage before file operations are committed. There has been a lot of work in making transaction based systems reliable. Some approaches and strategies for such systems can be found in [Eliot83], [Moss81], [Paxton79].

The above approach would only apply in transaction-based environments. In other environments it would be necessary for file operations to be resistant to processor crashes since there is no transaction or file commit operation. One approach which could be appropriate in some circumstances (in both transaction and non-transaction systems) involves the use of two processors, say p1 and p2, in an implementation of stable storage. It is reasonable to assume that these two processors are not crash-related. A write operation will write to the two processors and asynchronous I/O in each processor will be performed as in an ordinary file system. If p1 crashes, only buffered data in that machine will be lost, for we assume that p2 will complete its write operation. A return from either p1 or p2 is sufficient to ensure crash resistance of the stored data. This would eliminate the blocking of user processes which use a crash resistant file system in a multi-user environment. This approach would be similar to the Tandem[Borr81] approach to the provision of reliable storage except that in a Tandem system not only disk drives and processors would be replicated but also I/O channels, interprocessor buses, power supply units and so on in order to provide a non-stop service.

Another extension concerns the method of crash recovery. It would be desirable to implement a more selective crash recovery scheme which would avoid recovering the entire disk after a crash has occurred. It might be sufficient in many environments to recover only the files that were being written when a crash occurred. One approach would require recording the identity of the file that is being written so that only that file is recovered by the crash recovery routine. However, in environments such as ours, where the crash recovery routine is invoked at regular intervals for maintenance purposes, it is desirable to recover the entire disk during each crash recovery operation. This would enable the crash recovery routine to check the status of the entire disk as a maintenance action.

## 4.6. CONCLUSION

The UNIX file system has been implemented on stable disks so as to provide a crash resistant file system. Unlike the implementation of a stable storage system in chapter 3 which provided reliable disks for storing user data, this chapter has provided reliable disks for storing both user and system information. Modifications were made to the kernel to incorporate the management of replicated disks. It was also necessary to ensure that file management activities (such as the use of buffering mechanisms) do not introduce data inconsistency into the file system. This necessitated limited modifications to the UNIX file system code. Our approach has made it possible to make both user data and system file management information (such as i-nodes in UNIX) crash resistant. This has resulted in a UNIX file system which contains sufficient redundancy to enable easy recovery from transient hardware malfunctions, decay of storage media and processor crashes.

# 5. CRASH-RESISTANT, REPLICATED and STABLE STORAGE

## 5.1. INTRODUCTION

A reliable data storage system is expected both to safeguard the data that is entrusted to it and also to ensure that data objects are accessible in spite of the occurrence of failures. In a centralised system, it might be sufficient (with respect to accessibility of data) to record the value of an entity only once. In a distributed system, data entities can be made more readily accessible by replicating them on different machines so as to keep them nearer to where they are being used. Replication also enables many nodes to service requests for the same information in parallel. However, one of the main reasons for replicating data in a distributed system is to provide resiliency with respect to node and network failures. Failure of the machine storing the only copy (or copies) of some data will prevent the completion of any task which needs that data. In a distributed environment this could prevent the completion of a distributed task even though many of its sub-tasks running on other machines have been successfully completed. The provision of replicas on different machines means that any task (whether or not distributed) which needs the replicated data has a chance to proceed as long as there is a copy on a working node.

The previous chapters used the duplication of data within the same processor to help provide fault tolerance capabilities in the face of various disk and processor failures. This chapter uses the replication of data on different machines in a distributed system to increase the availability of stored data despite the failure of the communication network. Here we are concerned only with faults which occur in a distributed system due to the replication of data on different machines. In particular, we aim at ensuring that such replication of data does not result in the mutual inconsistency of data when the network partitions.

This chapter in fact describes the final stage in our data storage construction. It describes the construction of what we call CRES (Crash-resistant, Replicated and Stable) storage. This combines the use of data replication with the fault tolerance capabilities provided by the stable storage mechanism and the crash resistant file system of chapter 4 to produce a data storage system which is resilient to many of the faults which affect disk storage devices as well as to processor crashes and network failures. Such storage provides a higher degree of data reliability and availability than conventional data storage systems.

CRES storage replicates the crash resistant files which are implemented by the previous chapters on different machines in a distributed system. Our model of CRES storage is a collection of logical files. Each logical file F, is made up of multiple crash resistant files located on different computers such that F = {f1, f2,..., fm} where each fi is a crash resistant file on computer i. Each crash resistant file fi is in turn represented by a set of stable objects which reside on stable disks associated with computer i. (Recall that a stable object is an atomic entity that reliably stores data and with the attribute that any update operation on it either occurs completely or not at all.)

The use of multiple copies of data requires that mutual consistency of the copies be maintained. A user should not be allowed to access a resource which is in an inconsistent state. However, network partitioning could interfere with the maintenance of such consistency. Network partitioning occurs when a distributed system is divided into subsystems such that the sites in the different subsystems cannot communicate. This is usually brought about by a failure of the communications network. The non communicating subsystems are referred to as partitions. Many solutions have been proposed for maintaining the mutual consistency of resources [Alsberg76], [Ellis77], [Thomas78], [Stonebraker79], [Gifford79], [Eager,83]. However, these have not directly addressed the problem of resolving inconsistencies that can arise among replicated resources after a network had been partitioned. One drastic "solution" to the inconsistency problem resulting from network partitioning is to cease all operations in the distributed system until the network is fully reconnected. Another conventional solution is to allow processing to continue only in one partition. It is often desirable though, to keep individual sites or partitions operational when a distributed data storage system partitions. Such autonomous operation by various partitions could clearly result in mutual inconsistency among multiple copies of resources. It is therefore necessary that when the network reconnects, individual partitions should compare their update operations, detect conflicts and resolve them before partitions are allowed to merge.

This chapter presents an algorithm which addresses some of these issues. We address the problem of detecting and resolving mutual inconsistency among replicated data entities before the merge of a partitioned data storage system. It is usually accepted that automatic resolution of mutual inconsistency among replicated entities is not generally possible except when the semantics of operations on the entities are known a priori

[Walker83], [Brereton83], [Wright83]. These algorithms require a declaration of the readset and writeset of a task. (Recall that the readset and writeset of a task refer to those data objects which a task reads from or writes to.)

In contrast our algorithm combines conflict detection and resolution in such a way as to allow automatic conflict resolution without requiring that the readset and writeset of an application be declared beforehand. Another desirable property is that the update operations which have already been applied to a majority of the replicates are not undone when a conflict is detected.

Before going into the details of our proposal for a partitioned update and merge algorithm, it may be helpful to present in the next section a brief survey of previous work on replication and network partitioning. Section 5.3 presents our proposed algorithm for the update and merge of a partitioned data storage system. Section 5.4 considers the implementation of the proposed algorithm in a UNIX United system. It discusses a method for achieving replication transparency as well as the creation, reading and writing of replicated files in such an environment. Section 5.5 considers the performance characteristics of the proposed algorithm. In section 5.6 we present an extension to the merge algorithm. Section 5.7 gives some concluding remarks.

## 5.2. PREVIOUS WORK

This section gives a brief survey of replication and network partitioning algorithms. These algorithms can be classified as those requiring (1) weak consistency and (2) strong consistency.

### 5.2.1. Weak Consistency Algorithms

Weak consistency algorithms do not require that all copies of a replicated data item contain the same information at all times. Instead, the requirement is that all replicas contain the same values when update operations on a data item cease. In these algorithms update operations on replicas are performed in isolation (without consulting other replicas). The update requests are sent to other nodes with the hope that eventually all nodes will receive and carry out these update operations. Weak consistency algorithms depend on a highly reliable network which is expected to deliver all messages.

Arguments in support of maintaining only weak consistency are based on the need for increased availability. These algorithms are usually designed for such applications as mail systems and name servers which do not require strong consistency. (For example, take a simplified mail service which allows clients to add mail messages only at the end of the mail-box. Various copies of the mail-box may at times contain different messages as long as all copies of the mail-box contain the same set of messages when update operations on the mail-box ceases.) The degree of inconsistency allowed would depend on the application requirements. Weak consistency algorithms allow processing while a network is partitioned, but these algorithms are only suitable for a limited set of applications.

## 5.2.2. Strong Consistency Algorithms

Many algorithms which insist on strong consistency require that all copies or a majority of the copies be accessible before update operations can be performed on a data item [Ellis77], [Thomas78], [Mullery75].

When a network partitions, there are three processing possibilities (i) no part of the system can continue processing, (ii) only one part of the system can continue processing or (iii) two or more parts of the system can continue processing. Voting schemes are often used to determine whether or not processing (particularly update operations) can be carried out. There are however other algorithms such as control token algorithms [Alsberg76], [LeLann78], [Minoura82] which do not use voting solutions but instead insist that a primary site must be contacted before update operations can be performed.

## Voting Schemes and Primary Site Algorithms

In voting solutions, update operations are carried out only if all sites that store replicas have agreed to carry out the update operations. In this approach, mutual consistency of replicas is maintained by having all sites perform the same update operations in parallel. Usually, an initiating site broadcasts update requests to all sites. A "reject" or "accept" acknowledgement is returned from each site. Having received votes from all site, the initiating site broadcasts an "update" or "abort" message to all sites depending on the outcome of the voting. Sites have some criteria for accepting or rejecting an update request. A two phase commit [Gray78], [Lampson79] or similar protocol is used to obtain agreement among sites which store replicas.

The majority consensus algorithm[Thomas78] requires only that a majority of the sites return "accept" acknowledgements. The weighted voting scheme described by Gifford[Gifford79] provides a biased method of election in favour of application requirements. In that scheme, the democratic "one man, one vote" election criterion is not strictly adhered to. Replicas are assigned votes based on reliability, performance and availability requirements. If reliability is required for instance, replicas on reliable nodes are assigned more votes so that a consensus (permission to write to a replicated file) is obtained when such nodes vote.

Voting schemes which require that all replicas be accessible before an update operation can be carried out will of course be unable to access all replicas when a network partitions. Consequently, no replica will be updated. Majority consensus and weighted voting algorithms seem better suited for network partitioning since they do not demand that all replicas be accessible. It could therefore be possible at least for one partition to continue processing. However, a network could still partition in such a way that obtaining a majority consensus will not be possible.

Primary site algorithms do not seek consensus among replicas but require that a primary site be contacted before update operations can be performed. Various algorithms are used to determine a primary site. Control token algorithms allocate a control privilege temporarily to one site which then acts as the primary site. Other algorithms induce a total ordering of sites and then take the site with the highest sequence number as the primary site. In these algorithms, network partitioning might make the primary site inaccessible. However, the partition containing the primary site (possibly consisting of the primary site alone) will always be able to continue processing.

The major pitfall of the primary site approach is that it unduly restricts processing during both normal and partitioned processing since the accessibility of replicated data depends on a particular node being accessible.

### 5.2.3. Conflict Detection and Resolution Algorithms

The strong consistency algorithms are not resilient to network partitioning. By this we mean that they do not allow separated parts of a system to carry out autonomous processing on replicates when a network partitions. They do not, therefore, need to provide schemes for resolving the mutual inconsistency among replicates which could result from such partitioned operations.

In many circumstances a desirable solution is to allow autonomous operations in the various partitions and then detect and resolve conflicts when two or more partitions are merged. Algorithms for detecting mutual inconsistency among replicated objects are presented in [Parker83], [Brereton83]. However, these discuss resolution of inconsistency only for special file types such as directories and mail-boxes for which the semantics of operations are simple and, more importantly, known. Conflict resolution algorithms are similar to optimistic concurrency strategies[Kung82] since they do not try to mask or prevent faults from occurring but instead aim at resolving the conflicts which may arise.

A merge algorithm is presented in[Wright83a....]. This algorithm is based on obtaining a global serial schedule of the combined set of transactions from all partitions. Each partition maintains a record of the order in which transactions in that partition are committed. A history of the readsets and writesets of transactions in each partition are also kept using some appropriate scheme. A survey of techniques for storing update histories and recovery data can be found in[Verhofstad78]. When a network reconnects, each partition derives a serial schedule of all transactions which have committed in that partition since the partition was formed. These individual serial schedules are used to obtain a global serial schedule by backing out a subset of transactions. This algorithm, which uses a graph-theoretic approach, guarantees that it can be determined in polynomial time which transactions must be backed out so as to keep the combined set of transactions serializable. One approach to determining the order of transactions in a network would require a reasonable amount of synchronisation among clocks in the individual sites. (Clock synchronisation[Lamport78] in a distributed system is in itself a difficult problem.)

Another merge algorithm is described by Wright[Wright83].

This algorithm does not consider the problem of detecting inconsistencies. However assuming that inconsistencies can be detected, the method requires that the activities of an application program be grouped a priori into transaction classes. It might not always be possible to determine beforehand all the resource requirements of an application so as to group application programs into classes. Consider database applications where records to be read or written might depend on the value of fields in other database records. It would be difficult to know beforehand the complete set of resources which such an application program requires. In such circumstances, it might be necessary to declare the entire database as the readset and writeset of the application program.

A more desirable solution is to allow application programs to acquire necessary resources as the need arises. Our proposed algorithm (which is discussed in the next section) permits autonomous operations in the various partitions of a network and dynamically determines the writeset of an application program. It also provides a merge scheme for the general object type as opposed to a scheme for special file types whose operation semantics are known.

## 5.3. PARTITIONED PROCESSING

The following discussion applies only to replicated files.

For non-replicated files, the update and merge protocols are not invoked since partitioned processing is invoked only for replicated files. Our system differentiates between replicated and non-replicated files. A technique for detecting replication and keeping it transparent to the user is discussed later in this chapter.

A set of sites which are in communication with each other constitute a partition. It might be helpful to think of a partition as a single logical node consisting of a set of communicating nodes. The universal node unode(f) for a file f will be used to refer to the set of all sites in a distributed system which store copies of the file f. The majority partition Majp(f) for a file f will refer to the sites which store copies of f and which are in communication with a majority (absolute majority) of the sites in unode(f). Any partition which is not the majority partition will be referred to as a minority partition Minp(f). (The concept of majority partition is file dependent. A partition is a majority partition with respect to a specific file or a set of files.)

### 5.3.1. A Partitioned Update Algorithm

The algorithm presented here allows update activities in all partitions when a network partitions. However, only the update operations of the partition which has a majority of file replicates (majority partition) are reflected in the actual file copy. The update operations of a minority partition are delayed so as to be incorporated into the actual file copy after that partition has rejoined the majority partition. (The actual file copy refers to the file copy which is stored by each site in the majority partition.) We assume the existence of a mechanism for obtaining majority consensus. The use of a majority updating scheme ensures mutual consistency between the majority of the copies of a file (namely, those copies which are stored by the majority partition). It also ensures that the update operations which have been applied to the majority of the file copies are not undone during a merge operation since only the operations of the minority partition are undone. The merge algorithm is discussed in the next section. In order to allow minority updating and to be able to resolve conflicts which might arise, we use the following data structures:

An entity list is an ordered set of integers (E1, E2, ..., Em). It contains the identification numbers of all objects which are modified by a partition since the partition was formed. This data structure is maintained by every node in a partition.

A stable update set is a collection of records where each record contains the following:

N: node identifier
f: file identifier
ob: object number
u: update request = (read, write)
d: data to be written

An update set is similar to an intentions list in[Lampson79] except that instead of identifying a transaction, it identifies a node which is involved in an update operation. We shall make the following assumptions about the activities of a minority partition:

AS1: The update operations of a minority partition are assumed to be initiated at the time when that partition rejoins the majority.

AS2: Objects which are updated by a minority partition are not accessible to users until after that partition has merged with the majority partition. All access requests for such objects fail giving appropriate error reports.

Each site can determine whether it is part of a majority or minority partition by executing a majority consensus algorithm. The details of how this is carried out will be discussed later when considering implementation issues. The algorithm for updating objects is as follows: Each site in a minority partition maintains a stable update set for each file which it desires to update and stores its update requests in that set. A stable update set serves as an audit trail which is used to record requested operations on objects by nodes in a minority partition. This allows a file to be left in the state it was in when the partitioning occurred, and enabling subsequent update requests to be carried out later during a merge operation as long as no conflict is detected. It is possible to allow local processes to make use of (locally) updated values by making the contents of the stable update set accessible to local processes. It would then be necessary to address the problem of results being allowed to leave the computer system from a minority partition. Our present algorithm does not address this issue.

Sites in the majority partition apply their update operations to their local file copies after an agreement to perform the update operations has been obtained from every site in the majority partition. This is not difficult since sites in the majority partition are in communication with each other. If agreement is not obtained, the update request is rejected. Each partition maintains an entity list and records in it the identification numbers of objects that are modified in that partition. The use of an entity list enables the update and merge protocol to determine dynamically the update class of each partition and thus enable easy conflict detection. It is not necessary that a task declares its update class a priori (as in[Wright83] for example). Conflict detection is carried out when a site in the minority partition requests a merge. The next section discusses how a merge operation is initiated and carried out.

## 5.3.2. A Network Merge Algorithm

When a site in a minority partition determines (by executing a majority consensus algorithm) that it has resumed communication with the majority partition, it initiates a merge operation by sending a merge request to all sites. The majority consensus scheme which is applied to update requests by the majority partition is used to process a merge request. A merge request is accepted either conditionally or unconditionally. Unconditional acceptance is given when the update classes of the majority and minority partitions do not conflict; otherwise a conditional acceptance is given. In order that the operations of the two partitions may be considered conflict-free, it is usually required that the invariant

$$\text{INV: } Wi \cap Rj = 0 \wedge Ri \cap Wj = 0 \wedge Wi \cap Wj = 0$$

hold (Here i and j are the majority and minority partitions respectively; Ri and Wi are the readset and writeset of partition i). However, because of assumptions AS1 and AS2, the activities of the minority partition cannot influence the data which is read by the majority partition. We can therefore remove the first two requirements of the invariant INV and require only that $Wi \cap Wj = 0$ holds. The activities of two partitions can therefore be considered to conflict if their associated entity lists have one or more entries in common. If no conflict is detected, and the merge request is unconditionally accepted, the stable update set of the minority partition would be applied to the majority partition file copy and the entity list of the minority partition would be added to that of the majority partition. Conditional acceptance of a merge request requires that the affected minority partition discards its update set and entity list before rejoining the majority partition. After a successful merge operation, a minority partition initialises its file copy and entity list from the (merged) file copy and entity list of the majority partition. This algorithm is applied independently for each file which is modified by a minority partition.

Figure 14 gives an example of what happens during a merge operation in the case of a file f which is replicated on five machines U1, U2, U3, U4 and U5.

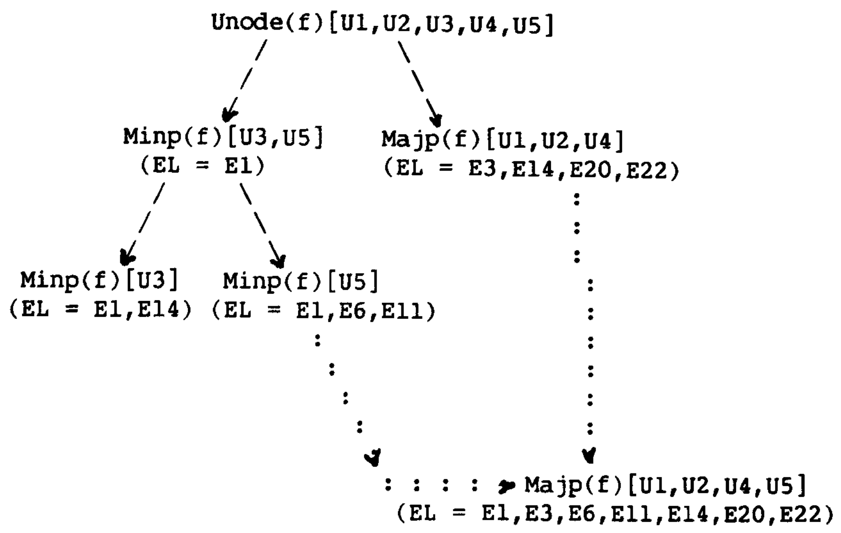

Figure 14: Merge Graph

In figure 14, the universal node Unode(f) consists of the five nodes which store copies of the file f. The entity list "EL" is used to record the identification numbers of updated objects in a partition. Stable update sets are used to store the update requests and data which is associated with such requests. Sub-partitions inherit the entity lists and stable update sets of their parent partition. An entity list for a partition would be empty only if no update operations were carried out in that partition. Each partition can determine whether or not it is part of a majority partition by executing a majority consensus algorithm. From figure 14, it can be seen that the minority partitions Minp(f)[U3, U5], Minp(f)[U3] and Minp(f)[U5] each carried out update operations since their entity lists are not empty. It can also be seen that the update operation carried out by Minp(f)[U3, U5] on object 1 as reflected in "E1" was inherited by the sub-partitions Minp(f)[U3] and Minp(f)[U5].

Suppose now that the minority partition Minp(f)[U5] requests a merge with the majority partition Majp(f)[U1, U2, U4]. The entity lists of the two partitions will be compared. From figure 14 it can be seen that these two entity lists have no entry in common which means that there is no conflict between the update operations of those two partitions. The merge request will therefore be accepted unconditionally. The contents of the stable update set of the minority partition Minp(f)[U5] will be applied to the majority file copy and sites which were in Minp(f)[U5] will reinitialise their entity lists from that of the majority partition's (merged) entity list.

Now consider a merge request by the minority partition Minp(f)[U3] with the Majp(f)[U1, U2, U4]. The entity lists of this partition and that of the majority partition have an entry in common namely, "E14". This means that a conflict exists. A conditional acceptance to the merge request will be given. Minp(f)[U3] will be required to discard its update set before rejoining the majority partition. It will also reinitialise its entity list from the entity list of the majority partition.

Notice that the majority partition does not maintain a stable update set, since its operations are applied directly on the majority file copy and such operations are not undone during a merge operation. An extension to the merge algorithm presented here which allows the merging of minority partitions is given in section 5.6

## 5.4. IMPLEMENTATION ISSUES

This section considers the implementation of the proposed algorithm in a UNIX United distributed system. The architecture of a UNIX United system has been described in chapter 3. The replication layer can be conveniently placed between the user and the Newcastle Connection software as portrayed in Figure 15.

```
.----------------------------.
|     User                   |
|-------------------------   |
|     Replication Layer      |
|--------------------------  |
|     Newcastle Connection   |
|----------------------------|
|     Kernel                 |
------------------------------
```

Figure 15: Position of the Replication Layer

The aim is to support file replication facilities while maintaining the standard UNIX system interface. This is achieved by making replication transparent to the user so that a replicated file can be accessed and updated with the same system calls as an ordinary UNIX file. However, the creation of a replicated file is not transparent to the user since we believe that a user should be aware of the fact that he is creating a replicated file. A new system call is provided by this algorithm for the creation of replicated files. (An alternative would be, as with stable files, to provide directories within which all files are automatically replicated.)

## 5.4.1. File Creation and Placement

To create replicated files a user would invoke a "**mkrep**" primitive. These files are created as special objects. The "mkrep" primitive marks these files to ensure that they are detectable as replicated objects. Each site has an ordered set of sites (related sites) on which it stores its file copies. The "mkrep" primitive takes an argument which indicates the number of replicas required. It creates a "first" file copy on its node and replicas on related nodes until the required number of replicas have been created. Entries are made in an RN (related node) table stored at each node to record the identity of nodes where a site stores replicas. Each entry in the RN table consists of an ordered set of node names which indicate the order in which related nodes should be contacted. The first entry in each table is referred to as the first related node, the second as the second related node and so on.

```
U1: (U2)
U2: (U1, U3, U4)
U3: (U2, U4)
```

Figure 16: RN (related node) Table

Figure 16 shows that computer U1 stores its file copies in computer U2. Computer U2 stores file copies in computers U1, U3 and U4. Computer U3 stores its file copies on computers U2 and U4. Each computer maintains a separate replication directory "REP" on behalf of each node for which it stores file copies. These file copies are stored in the respective "REP" directory. Looking at figure 16, since computer U2 stores file replicates on behalf of computer U1 and U3, computer U2 will maintain directories "REP1" and "REP3" which will contain file replicates from U1 and U3 respectively.

### 5.4.2. File Access

File access is the same for replicated files as for ordinary UNIX files. A user calls the operations of the replication layer in order to open, read and write files. Replication is made transparent to the user. The replication layer intercepts all system calls so as to detect access requests to replicated files. This layer is also responsible for obtaining majority consensus. In accessing a file which is replicated on

N machines, this layer would map the user's single access request into N requests (to all copies) in the case of a write request and one request (to any accessible copy belonging to the majority partition) in the case of a read request. We assume the existence of a facility to enable a site to execute a majority consensus algorithm by sending messages to all nodes in order that it can determine whether or not it is part of a majority partition. The number of acknowledgements received in response to such messages would enable a site to determine whether it is in communication with the majority of the sites in the network. When this has been determined, the replication layer then issues requests to all sites in the partition to write to their local file copy or stable update set depending on whether or not a site belongs to the majority or minority partition respectively. A read request is first directed to the local file copy, if a copy is available at the user's node otherwise it is directed to the "first" file copy. If this is not successful, the related nodes are searched for replicas starting with the first related node. Whether a site seeks majority consensus for each separate access or at periodic intervals would depend on the consistency requirements of the application. If an application seeks majority consensus periodically, it has to assume that it is part of a minority partition until such consensus is sought and obtained.

Our file access strategy requires that if for any reason (network partitioning or node failure) the "first" file copy is not accessible, the replication layer is to search the RN table to determine where the file is replicated. The replication layer then tries to access the file replica in the first related node of the RN table. This is continued until a replica is accessed or all related nodes have been tried. If the file is not a replicated file, all file access requests to related nodes would fail, otherwise a replica will be accessed. The replication layer makes file copies belonging to minority partitions inaccessible to remote nodes. All requests for such files fail giving appropriate error reports. The replication layer passes all accesses to remote nodes to the Newcastle Connection software which handles remote accesses in a UNIX United system.

### 5.4.3. Replication Transparency

There are various methods which can be used to obtain replication transparency. One approach would be to use a centralised name resolution scheme for replicated files. In such a scheme, a particular site would be consulted for resolution of name references for the entire network or for a subset of the nodes in the network. It would then be possible to determine that a file is replicated by consulting such a node and to make such name resolution transparent to the user. The use of a current synchronization site[Walker83] and the total resource directory[Lunn82] are examples of such schemes. This approach introduces a centralised source of unreliability because of the potential failure of such a vital node.

A preferable solution would be the implementation of a distributed name resolution scheme. Such a scheme will be capable of using information which is locally available to determine that a file is replicated and will be able to give information on its related nodes. One approach would be to tag pathnames to files so that replication can always be detected by inspecting the pathname to a file. Implementing such a scheme in a hierarchical system requires storing not only file names but also the contexts in which those names appear. The existence of a hierarchical structure imposes a context relative naming scheme. Consequently, names do not make sense except when interpreted within a context. In a hierarchical filestore implemented by a network of computer systems, there are numerous pathnames to a file depending on where a name reference starts. It would therefore be prohibitive in terms of storage space requirements to store all possible pathnames to a file.

However, these problems can be avoided by tagging files at creation time. Since the "mkrep" primitive creates a replicated file as a special object, the replication layer sees such a file as one requiring special processing. Further investigation into the "specialness" of such a file can reveal that the file is a replicated file and not a "special file" in UNIX terms. (Special files in UNIX are usually associated with devices.) When it has been determined that a file is replicated, the file access strategy which was described in the last section would then be invoked.

The replication layer intercepts all system calls and can therefore control the contents of the information which a user sees. This layer ensures that a user sees a replicated file as an ordinary UNIX file. Replication transparency could thus be achieved without modifying the UNIX kernel.

### 5.4.4. Data Structure Implementation

The two main data structures which are used by this algorithm have a straightforward implementation in UNIX as crash resistant files. A stable update set can be implemented as a sequence of 512 byte blocks within a file, where each block constitutes an object. Each disk block would contain the "N", " f", "ob" and "u" parameters which identify the node and file to which a block of data belongs. A sufficient number of disk blocks within a file would be used to store the data which is contained in the "d" parameter. The entity matrix can be implemented as a file of integers.

## 5.5. PERFORMANCE

In order to estimate the overhead incurred when this algorithm is used, let us define a few parameters:

(i) There are n file replicates for each file in the data storage system.

(ii) The time required to write a local copy is w

(iii) The time required to read a local copy is r.

(iv) Let e represent the delay in searching for a replica in one node. The delay in searching for an accessible replica when a copy is not available locally is $d \leq ne$.

(v) The time required for obtaining majority consensus is c.

We assume that computing times and times for making remote procedure calls are negligible. Storage overheads are also negligible since update sets are discarded after successful merge operations and therefore do not become very large. In order to avoid having large entity lists, all entity lists can be initialised to empty when a merge operation results in Majp(f) = Unode(f). (That is when the network is fully reconnected and all nodes are in communication with each other. In such a situation, it would not be necessary to maintain entity lists.) The overhead in reading and writing replicated files would be:

Read-overhead = $d + c$

Write-overhead = $(n - 1)w + c$

The "d" parameter may be negligible if a copy is available locally or if the first related node is accessible. The "w" parameter may be small since all accessible nodes will carry out their update operations on an object in parallel, each on its local copy. The major overhead might be in getting majority consensus or the use of agreement protocol in order to obtain consensus from replicas on the course of action to be taken during an update operation. If a particular implementation decides to seek majority consensus periodically rather than for each update operation, the "c" overhead would be reduced.

However, we believe that in appropriate circumstances the enhancement in reliability obtained by replicating resources could outweigh these performance costs. Suppose that each of the replicates has a probability p of being accessible. A system consisting of n replicates would have

$$s = 1 - (1 - p)^{n}$$

probability of being accessible. As n increases, the probability of such a system being accessible approaches 1 thereby providing a very high degree of data availability. However, if network partitioning and update conflicts are frequent the probability "p" of accessing a replica will be small.

## 5.6. AN EXTENSION

The only merge operation allowed by the merge algorithm presented in this chapter is a merge between a majority and a minority partition. However, a network could partition in such a way that there would be no majority partition though one could come into existence again through the merging of minority partitions. An extension to this algorithm which would allow the merge of minority partitions is generally not difficult.

One approach would be to merge minority partitions by combining their stable update sets and entity lists so that each site in the merged partitions contains the same information in these two data structures. This will be done after the entity lists of the two minority partitions have been compared so as to ensure that there is no conflict between the operations of the two partitions. If a conflict is detected, the operations of the partition which carried out the least number of update operations will be backed out by discarding its update set. The partition which was backed out would then initialise its entity list and stable update set from those of the partition with which it has been merged. Then a majority consensus algorithm would be executed to determine if the newly merged partition has formed a majority partition. Each site knows the number of sites in Unode(f) so it is easy to determine if a partition constitutes the majority partition. If the newly merged partition contains a majority of the nodes in Unode(f) then that partition constitutes the majority partition, in which case its update set would be applied to the local file copy of each site in that partition. The update set would then be discarded since majority partitions do not maintain stable update sets. On the other hand if the newly merged partition does not form a majority partition, it would remain as a minority partition and would continue to follow minority processing procedures. There can not be more than one majority partition in a network at the same time since any site which is in communication with a site in the majority partition is itself part of that partition.

## 5.7. CONCLUSION

The problem of maintaining the consistency of data in a distributed data storage system which supports replicated resources has been considered. An algorithm for detecting and resolving mutual inconsistencies among replicates in the presence of network partitioning was presented. The algorithm provides a merge scheme for the general object type instead of for special types of objects whose operation semantics are known. Implementation of this algorithm in a UNIX based distributed system was discussed (although unlike the techniques of chapters 3 and 4 this algorithm has not been implemented). The algorithm also discusses a simple method of implementing replication transparency in a distributed hierarchical file store.

# 6. CONCLUSION

## 6.1. SUMMARY

We have considered the general problem of constructing robust data storage in a distributed computer system. Our approach to the construction of such a system involves providing reliable storage of data and also ensuring that data objects are made accessible in spite of the existence of node and network failures. In particular we investigated the integration of the stable storage mechanism with data replication on different computers as a basis for the construction of reliable distributed file storage systems.

The investigation was carried out in the environment provided by UNIX United which is a UNIX based distributed system implementing a global hierarchical file store. Such a system was considered highly appropriate for the construction of reliable file systems because of the convenient features which it provides such as expandability and distribution transparency.

A survey of the causes of data inconsistency and techniques for maintaining data consistency in distributed systems has been discussed. The design and implementation of a stable storage system was presented. This system implemented stable disks which provide fault tolerance facilities for faults which affect disk storage devices, such as decay of the storage medium, transient I/O faults and some effects of processor crashes.

An implementation of a crash resistant UNIX file system which is based on the use of the stable storage mechanism has been described. Performance evaluation of the stable storage system and the crash resistant Unix file system has also been carried out. Replication of resources can be an aid to both performance and reliability in distributed file store design. An algorithm for detecting conflicts among replicated objects and a merge protocol for a partitioned data storage system was presented. Implementation of these algorithms in a UNIX United system was discussed.

## 6.2. AREAS FOR FUTURE WORK

Some extensions to the presented algorithms have already been mentioned in the individual chapters. This section highlights some of these extensions and makes further suggestions as to areas where more work could be useful.

Obviously it would be desirable to carry the ideas presented in chapter 5 on network partitioning through to implementation so that the actual overheads and efficiency of the scheme, could be assessed. This will however require access to quite a large distributed system.

Another interesting area for investigation would be the provision of user-level (application dependent) fault tolerance facilities on top of the facilities which we have provided. The main aim would be to ensure that, if all automatic fault tolerance facilities fail, an end-user who knows the semantics of the application should be given control before extensive "UNDO" operations are initiated. Such a user would be in a position to

(i) accept all the results of processing as produced,

(ii) or accept only part of the results,

(iii) or reject all results and call on the computer system to initiate "UNDO" processing.

It would of course be even more desirable to have a full and preferably rigorous specification of an application's requirements, so that the computing system can make decisions (i), (ii) and (iii) above automatically.

Another area which requires further investigation is the use of forward error recovery techniques for the resolution of inconsistency between replicated objects in a distributed system. Most existing schemes use backout strategies for conflict resolution. Using forward error recovery would again require a knowledge of the semantics of the application so that conflicts could be resolved appropriately. A mechanism which is sufficiently general to support both forward and backward error recovery strategies would be highly desirable. Interesting discussions on the general use of forward error recovery in computer systems can be found in [Cristian82], [Campbell83].

We have already mentioned the need to provide robustness of file systems in the presence of buffering. To solve this problem we had suggested as an extension to our algorithms the use of two processors instead of one in the implementation of stable storage systems. We could generalise this extension further so as to be able to tolerate a bounded number of processor crashes by the use of a specified number of processors. An efficient solution which uses this technique and whose cost can be justified would also be of interest. However such an extension would be taking us toward the situation where one could be using N-modular redundancy techniques, based on redundancy at the level of complete computer systems. Thus it would be necessary to investigate the relative merits of the two approaches.

The provision of robust processes can be considered to be a possible step forward after the provision of stable storage facilities. Such robust processes would be resilient to processor crashes. These processes would store their states in stable storage so as to be restarted from the saved state after a processor crash or service interruption. This idea has already been suggested in[Lampson79]. A specific way of looking at the implementation of such a scheme is as follows:

An operating system creates processes which carry out work on behalf of the computer system and the user. We shall consider these processes to be part of the processor state. The processes which are part of (owned by) the current processor state we will refer to as "own" processes. Processes whose states have been saved in stable storage, and which need to be restarted, would have to be considered "adopted" processes since they do not belong to the current processor state but instead belong to a previous (crashed) processor state. The problem then is to integrate these adopted processes into the current processor state.

In summary, we have investigated various related approaches to achieving high reliability and availability based mainly on the extensive use of storage replication. It would be very useful to have some means of estimating the relative costs and benefits of these and other approaches for a given type of environment and use, so as to be able to choose an optimum system design in each case. This however would be a major research project in its own right.

## 6.3. ACHIEVEMENTS AND CONCLUSION

Previous work on stable storage systems provided atomic fixed size pages for storing data. This thesis investigated the design and construction of a more flexible stable storage system which provides atomicity for more complex data structures instead of the usual fixed size pages. An algorithm for providing such a facility has been presented and implemented and its performance reported on.

The thesis has also demonstrated how the stable storage mechanism can be interfaced with an existing file system and how such a file system can be made crash resistant. This required constrained modifications to an existing operating system kernel. We consider the success of that investigation and the subsequent implementation of our ideas rewarding, since it is usually not easy to incorporate fault tolerance facilities in an already existing system. The proper placement of such facilities so as to provide effective fault tolerance is not clear.

Another contribution of this thesis is in the area of network partitioning. We have presented a novel algorithm for the update and merge of a partitioned data storage system. This algorithm has some highly desirable properties. It allows autonomous processing in all partitions of a data storage system whilst it is partitioned. Its protocol for the merge of a partioned data storage system ensures that conflicting update operations are detected and resolved before a merge operation is allowed to proceed. It uses the contents of its stable update set to carry out operations of a minority partition for which no conflict is detected. In the event of the detection of an error, it ensures that update operations which have already been applied to the majority of replicas of an object are not undone.

Our approach combined the stable storage concept with facilities for providing crash resistance in file systems and replication of data objects on different machines to provide a data storage system which is highly reliable and highly available. In conclusion, we would like to believe that we have explored a particular subject area and thrown some light on some concepts and aspects of that area through survey, design and implementation. On a personal level, it has provided me with a deeper knowledge of computing systems and their reliability problems, and yielded the satisfaction of participating in their solution (even in a small way).

**References**

Alsberg76.

P. A. Alsberg and J. D. Day, "A Principle for Resilient Sharing of Distributed Resources," IEEE Proc. Second Int. Conf. on Softw. Eng., pp. 562 - 570, San Francisco, Oct.1976.

Anderson81.

T. Anderson and P. A. Lee, Fault Tolerance: Principles and Practice, Prentice-Hall, 1981.

Avizienis75.

Avizienis, "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing," Proc. Int. Conf. on Reliable Software, pp. 458 - 464, Los Angeles, California, April 1975.

Avizienis77.

A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance During Execution," COMPSAC 77, pp. 149 - 155, Chicago(IL), November 1977.

Bernstein80.

P. A. Bernstein, D. W. Shipman, and J. B. Rothnie, "Concurrency Control in a System for Distributed Databases (SDD-1)," ACM Trans. on Database Systems, vol. 5, no. 1, pp. 18 - 25, March 1980.

Bernstein81.

P. A. Bernstein, "Concurrency Control in Distributed Database Systems," Computing Surveys, vol. 13, no. 2, pp. 185 - 221, June 1981.

Bhargava82.

B. Bhargava, "Resiliency Features of the Optimistic Concurrency Control Approach for Distributed Database Systems," Second Symp. on Reliability in Distr. Softw. and Database Systems, pp. 19 - 32, July 1982.

Borr81.

A . J. Borr, "Transaction Monitoring in Encompass[TM]: Reliable Distributed Transaction Processing," Proc. Seventh Int. Conf. On Very Large Databases, pp. 155 - 165, Cannes, France, Sept. 1981.

Brereton83.

P. Brereton, "Detection and Resolution of Inconsistencies among Distributed Replicates of Files," ACM Operating Systems Review, vol. 17, no. 1, pp. 10 - 15, Jan. 1983.

Brownbridge82.

D. R. Brownbridge, L. F. Marshall, and B. Randell, "The Newcastle Connection," Software Practice and Experience, vol. 12, no. 12, pp. 1147 - 1162, Dec. 1982.

Campbell83.

R. H. Campbell and B. Randell, "Error Recovery in Asynchronous systems," Technical Report 186, Computing Lab. University of Newcastle Upon Tyne, July 1983.

Cristian82.

F. Cristian, "Exception Handling and Software Fault Tolerance," IEEE Trans. On Computers, vol. C-31, no. 6, pp. 531 - 540, June 1982.

Cristian83.

F. Cristian, "A Rigorous Approach To Fault-Tolerant System Development," Report RJ3754, IBM San Jose, California, January 1983.

Defence80.

U. S. A. Department of Defence, Reference Manual for the ADA Programming Language, 1980.

Eager,83.

D. L. Eager, and K. C. Sevcik, "Achieving Robustness in Distributed Database Systems," ACM Trans. on Database Systems, vol. 8, no. 3, pp. 354 - 381, Sept. 1983.

Eliot83.

J. E. Eliot, "Checkpoint and Restart in Distributed Transaction Systems," Proc. Third Symp. on Reliability in Distr. Softw. and Database Systems, pp. 85 - 89, October 1983.

Ellis77.

C. A. Ellis, "A Robust Algorithm for Updating Duplicate Databases," Proc. Second Berkley Workshop on Distr. Data Management and Computer Networks, 1977.

Eswaran76.

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," CACM, vol. 19, no. 11, pp. 624 - 633, Nov. 1976.

Gifford79.

D. K. Gifford, "Weighted Voting for Replicated Data," Proc. ACM Seventh Symp. on Operating System Principles, pp. 150 - 162, Pacific Grove, California, Dec. 1979.

Goodenough75.

J. B. Goodenough, "Exception Handling: Issues and a Proposed Notation," CACM, vol. 18, no. 12, pp. 683 - 696, Dec. 1975.

Gray78.

J. N. Gray, "Notes on Database Operating Systems," Lecture Notes In Computer Science: An Advanced Course, pp. 393 - 481, Springer Verlag, New York, 1978.

Gray81.

J. N. Gray, "The Recovery Manager of System R Database Manager," <u>Computing Surveys</u>, vol. 13, no. 2, pp. 223 - 242, June 1981.

Gray81a

J. N. Gray, "The Transaction Concept: Virtues and Limitations," <u>Proc</u>. <u>Seventh Int</u>. <u>Conf</u>. <u>on Very Large Databases</u>, pp. 144 - 153, Cannes, France, Sept. 1981.

Haerder83.

T. Haerder, "Principles of Transaction Oriented Database Recovery," <u>Computing Surveys</u>, vol. 15, no. 4, pp. 287 - 317, Dec. 1983.

Hoare75.

C. A. R. Hoare, "Data Reliability," <u>Int</u>. <u>Conf</u>. <u>on Reliable Software</u>, pp. 528 - 533, Los Angeles, California, April 1975.

Horning74.

J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B.

Randell, "A Program Structure for Error Detection and Recovery," Proc. Conf. Operating Systems: Theoretical and Practical Aspects: IRIA, pp. 177 - 193, Rocquencourt, April 1974.

Kohler81.

W. H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralised Computer Systems," Computing Surveys, vol. 13, no. 2, pp. 149 - 182, June 1981.

Kung82.

K. T. Kung and J. T. Robinson, "Optimistic Methods for Concurrency Control," ACM Trans. on Database Systems, vol. 6, no. 2, pp. 213 - 226, June 1982.

Lamport78.

L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," CACM, vol. 21, no. 7, pp. 558 - 565, July 1978.

Lampson79.

B. W. Lampson and H. E. Sturgis, "Crash Recovery in a Distributed Data Storage System," Xerox PARC Report, Palo Alto, California, April 1979.

LeLann78.

G. LeLann, "Algorithms for Distributed Data-Sharing

Systems which use Tickets," Proc. Third Berkley Workshop on Distr. Data Management and Computer Networks, pp. 259 - 272, Aug. 1978.

Lee83.P. A. Lee, "Exception Handling in C Programs," Software Practice and Experience, vol. 13, no. 5, pp. 389 - 405, May 1983.

Liskov83.

B. Liskov, "Preliminary Argus Reference Manual," M. I. T. Programming Methodology Group Memo 39, October 1983.

Liskov79.

B. H. Liskov and A. Snyder, "Exception Handling in CLU," IEEE Trans. on Software Engineering, vol. SE-5, no. 6, pp. 546 - 558, November 1979.

Lunn82.

K. Lunn, "Reliable File Storage in a Distributed Computing System," Ph. D Thesis, Keele University, March 1982.

Lyons62.

R. E. Lyons and W. Vanderkulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability," I. B. M. Journal of Research and Development, vol. 6, no. 2, pp. 200 - 209, April 1962.

Melliar-Smith77.

P. M. Melliar-Smith and B. Randell, "Software Reliability: The Role of Programmed Exception Handling," SIGPLAN Notices, vol. 12, no. 3, pp. 95 - 100, March 1977.

Minoura82.

Toshimi Minoura, "Resilient Extented True-Copy Token Scheme for a Distributed Database System," IEEE Trans. on Softw. Eng., vol. SE-8, no. 3, pp. 173 - 188, May 1982.

Moss81.

J. E. B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing," Ph.d Dissertation, Laboratory for Computer Science, M. I. T., Mass., April 1981.

Mullery75.

A. P. Mullery, "The Distributed Control of Multiple Copies of Data," Report RC 5782, IBM T. J. Watson Research Center, Yorktown Heights, N. Y., August 1975.

Papadimitiou84.

C. H. Papadimitiou and P. C. Kanellakis, "On Concurrency Control by Multiple Versions," ACM Trans. on Database Systems, vol. 9, no. 1, pp. 89 - 99, March 1984.

Parker83.

D. Stott Parker, G. J. Popek, G. Rudisin, A. Stoughton, B.

J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of Mutual Inconsistency in Distributed Systems," IEEE Trans. on Softw. Eng., vol. SE-9, no. 3, pp. 240 - 247, May 1983.

Paxton79.

W. H. Paxton, "A Client-Based Transaction System to Maintain Data Integrity," Proc. Seventh Symp. on Operating System Principles, pp. 18 - 23, Asilomar, California, December 1979.

Randell75.

B. Randell, "System Structure for Software Fault Tolerance," IEEE Trans. on Softw. Eng., vol. SE-1, no. 2, pp. 220 - 232, June 1975.

Randell78.

B. Randell, "Reliable Computing Systems," Lecture Notes in Computer Science: Operating Systems, pp. 282 - 391, Springer Verlag, N. Y., 1978.

Ritchie79.

D. M. Ritchie, "The UNIX I/O System," UNIX Programmer's Manual, Seventh Edition, Jan. 1979.

Rosenkrantz78.

D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II, "System Level Concurrency Control for Distributed Database Systems," ACM Trans. on Database Systems, vol. 3, no. 2, pp. 178 - 198, June 1978.

Rothnie80.

J. B. Rothnie, "Introduction to a System for Distributed Databases (SDD-1)," ACM Trans. Database Systems, vol. 5, no. 1, pp. 1 - 17, March 1980.

Schlageter76.

G. Schlageter, "The Problem of Lock by Value in Large Databases," Comput. J., vol. 19, no. 1, pp. 17 - 20, Feb. 1976.

Schneider83.

F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," ACM Trans. on Computing Systems, vol. 1, no. 3, pp. 222 - 238, August 1983.

Shrivastava82.

S. K. Shrivastava and F. Panzieri, "The Design of a Reliable Remote Procedure Call Mechanism," IEEE Trans. On Computers, vol. C-31, no. 7, pp. 692 - 697, July 1982.

Stonebraker76.

M. Stonebraker, "The Design and Implementation of INGRES," Trans. On Database Systems, vol. 2, no. 3, Sept. 1976.

Stonebraker79.

M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," IEEE Trans. on Softw. Eng., vol. SE-5, no. 3, pp. 188 - 194, May 1979.

Sturgis80.

H. Sturgis, J. Mitchell, and J. Israel, "Issues in the Design and use of a Distributed File System," ACM Operating System Review, vol. 14, no. 3, July 1980.

Svoboda81.

L. Svoboda, "A Reliable Object-Oriented Repository for a Distributed Computer System," Proc. ACM Eigth Symp. on Operating Systems Principles, pp. 47 - 58, Pacific Grove, California, Dec. 1981.

Thomas78.

R. H. Thomas, "A Solution to the Concurrency Control Problem for Multiple Copy Databases," IEEE COMPCON, pp. 56 -62, San Francisco, Spring 1978.

Thompson78.

K. Thompson, "UNIX Implementation," Bell System Technical Journal, vol. 57, no. 6, pp. 1931 - 1946, July 1978.

Verhofstad78.

J. S. M. Verhofstad, "Recovery Techniques for Database Systems," Computing Surveys, vol. 10, no. 2, pp. 167 - 195, June 1978.

Walker83.

Bruce Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," Operating Systems Review, vol. 17, no. 5, pp. 49 - 70, Oct. 1983.

Wright83, a.

D. D. Wright and D. Skeen, "Merging Partitioned Databases," Report TR83-547, Dept. of Computer Science, Cornell University, Ithaca N. Y., April 1983.

Wright83.

D. D. Wright, "Managing Distributed Databases in Partitioned Networks," Report TR83-572, Dept. Computer Science, Cornell University, Ithaca, N. Y., Sept. 1983.

Wyeth73.

D. Wyeth, "Estimates for the Size of the Recursive Cache," Internal Memorandum SRM/71, Computing Laboratory, University of Newcastle Upon Tyne, Dec. 1973.

## APPENDIX

### Implementation of the Fault Tolerance Notation

This section shows how the fault tolerance notation of chapter 3 was implemented using the exception package described in section 3.4.2. 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. However, we have 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: int)
const bufsize = 512; /*size of buffer */
type answer = (goodread,diskerror,...);
var result: answer;
retryno, maxtry: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)
                retryno = retryno + 1;
            WHEN(OTHERS)
                write reports, "error not disk error";
                break;

      END     /*end of context */
     until retryno >= maxtry;
     if result = goodread then return(result)
     else exc-signal(OP-FAIL);
end.
```

```
Proc Twrite(fd: int; buf: array[...] of char; bufsize: int)
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
      else 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 is not amenable 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.