# The Design of a Network Filing System

Paul Michael McLellan

Ph. D.

University of Edinburgh

1981

Contents

| | |
|---|---|
| Abstract | 2 |
| Chapter 1: Introduction | 3 |
| Chapter 2: Criteria for a Network Filing System | 18 |
| Chapter 3: Some Existing Network Filing Systems | 30 |
| Chapter 4: Data Transfer Primitives | 50 |
| Chapter 5: Naming | 59 |
| Chapter 6: Security | 88 |
| Chapter 7: Consistency | 103 |
| Chapter 8: Local Discs | 155 |
| Chapter 9: Implementation | 161 |
| Chapter 10: Conclusions | 200 |
| References | 205 |

## Abstract

This thesis considers the issues involved in the design and implementation of a filing system for a large open inhomogeneous computer network. The filing system proposed is implemented on a number of file servers, that is, computers attached to the network specifically for this purpose. The view as seen by other client computers on the network is of a single coherent filing system, even though for efficiency and reliability the individual file servers are largely autonomous. The facilities and guarantees available do not vary with the location of files, so that sharing of remote files is easy. In particular, there is a single scheme for textnaming files which is almost completely free of restrictions. There is also a network wide security scheme for permitting and preventing access to files by other users and programs.

The most important guarantee given by the filing system is that of consistency. Irrespective of concurrent access by other clients, and of crashes of either servers or clients, consistency of files is preserved. Old versions of all files, not just those which are completely rewritten, are retained for a time so that consistent views of the past are available. This permits a consistent backup snapshot to be taken of part of the filing system whilst it is active. Consistency is preserved by atomic transactions, and a commitment scheme is developed which is faster and more suited than most others to an ordinary filing system.

## Chapter 1: Introduction

Before considering in detail the issues in the design and implementation of a network filing system, it is necessary to have a working definition of the term.

* A network filing system is a filing system implemented by a number (perhaps only one) of computers which are accessed by other computers over a communication network.

Almost any preconceptions about what is meant by a filing system and what is meant by a network will suffice until the definitions are refined. Perhaps the most ubiquitous examples of each are the Unix filing system [RIT74] and the Ethernet [MET76]. An important word in the definition is the "a"; a network filing system is a single coherent filing system and not a group of filing systems all accessible over the same network.

Many of the loose terms used in describing systems have been seized by manufacturers who have used them as precise technical terms in the description of their own products. In case these terms produce confusion, some of them are defined below.

The computers implementing the network filing system will be called "file servers". The computers accessing the network filing system will be called "clients". These client machines may have their own filing systems or just a small disc used as a cache (or, perhaps, bubble memory or something similar). Such discs are not necessarily part of the network filing system and will be called "local" discs. Note that a client is a computer, or a program running on a computer. When the distinction is important, people using client machines (and programs) will be called "users". The term "site" will be used to mean either a server or a client.

A filing system which runs in the same computer as its host operating system will be called a "coresident" filing system. It is not out of the question that a coresident filing system should also be accessible over a network as part of a network filing system.

There are a number of reasons why client computers or server computers may stop running. Obvious reasons are power failure, hardware failure or a software detected fatal error. Such stoppages will be called "crashes".

The filing system stores files somewhere. This storage is called "stable" since it is non-volatile and survives even when the file server processor is turned off. At present, this is almost certainly magnetic disc but write-once laser optical discs or bubble memory may become attractive in the future. Independent, and perhaps removable, units of stable storage will be called "volumes". A stable storage volume is thus a magnetic disc pack, although it could be other things in the future.

Most filing systems name files in some way. If these names are character strings then they will be called "textnames". Normally these textnames will be bound to some internal file identification, probably a number; this will be called the "internal name" (occasionally, "internal filename") for the file. If this internal name is meaningful outside of the network filing system, perhaps as a sort of capability for the file, then it may also be called a "token" for the file. The unqualified term filename will be avoided as being confusing. However, the words like "naming" and "nameable" will be used in a general sense covering both types of name.

Many filing systems allow users to permit and bar other users access to their files. This will be called a scheme of "security". Note that security has nothing to do with preserving files in the face of crashes but rather with protecting the data they contain

against unauthorised access.

Guaranteeing that a crash will not wreck the filing system, often called file integrity, will be called "consistency". File integrity, in the sense of no overlapping files and so on, is but a part of a wider notion which encompasses the integrity of data in files and agreement between the data in files and other data outside of the filing system.

## Why have a network filing system?

The motivations for having a network filing system are closely bound up with the motivations for having a network in the first place. In fact, if the existence of the network is taken for granted then it is difficult to find convincing reasons for not having some form of network filing system.

The motivations for having a network are twofold. Firstly, adding communications equipment to existing systems gives easier transfer of data from system to system. Secondly, constructing decentralised systems gives a new, and perhaps better, way of performing computation. Furthermore, using a network of small computers significantly reduces the cost of entry to having a computer system, and the cost of upgrading an existing computer system. It short, the approach is generally more flexible.

In the past, computing has been done on large central computers. If the sites where computational power was required were remote from the central computer then communication lines were used to connect terminals to the central computer. If the terminals were within the boundary of the institution owning the computer then these communication lines were normally dedicated wires run around the site. If the terminals were more remote then a combination of

economics and, especially, the monopoly of most countries' telecommunication carriers forced the use of the telephone network. This mode of computing had a number of underlying assumptions.

Firstly, since most terminals (for example, teletypes) were slow then it followed that high bandwidth communication was unnecessary. Secondly, it assumed that computers were very expensive and had to be shared. Thirdly, it assumed that appropriate bandwidth communication equipment was relatively cheap (compared to computers). All three of these assumptions are much less valid today.

Visual display units have replaced most printing terminals. The bandwidth provided by the telephone network is insufficient to drive such displays at acceptable speeds for the kind of tasks now needed and even that provided by dedicated lines is not as great as the bandwidth at which users can extract information. It is at least an order of magnitude faster to skim through a (paper) book to find something, than to look at pages of the book on a display drawing at 1000 characters per second. When any sort of graphics is considered, for preparation of text for a photo-typesetter or for design of VLSI circuits, it is apparent that existing bandwidth between a central computer and a terminal is woefully inadequate. Even developing programs benefits from high bandwidth graphics [TEI77]. The first solution was to put limited computational power into the terminal. For example, editing text could be done without much recourse to the central computer; it thus had better response and, often, better bandwidth to the display. Graphics was done by running a suitable graphics protocol in a small satellite computer with the computational work being done on a much larger central computer.

The tenet that computers are very expensive is found wanting. Moderately sized computers have become very cheap and are still becoming cheaper (state-of-the-art machines are expensive and will always be so; what is commonplace and cheap today was

state-of-the-art and expensive yesterday). Computational power is now cheaper bought in the form of a lot of small machines than one big one. Unfortunately, connecting hundreds of small computers to get the power of a very large one has proved elusive. What has happened though, is that a single cheap computer can provide most of the needs of a single user. This leads to a connection topology which is all too easy to understand; the cheap computers are not connected at all. Since there is no longer a central filing system, each cheap computer is provided with a cheap disc. Eventually, some point to point communications equipment may be added. However, this is not an integral part of the system, but rather a route for two computers to transfer data.

The third tenet, that communications were relatively cheap has changed. The first reason is that processors have become cheap making communications relatively more expensive even without any other considerations. The second reason that communications have become more expensive is that higher bandwidth is now expected to be delivered to end-users and bandwidth is not obtained cheaply. However, over a small area, high-bandwidth (1-50 megabits per second) networks are acceptably cheap. Such networks use a very cheap transmission medium, wire or a fibre-optic light-pipe, rather than interposing a lot of expensive equipment between sites [MET76,WIL79]. The cost of the network is just the cost of the interfaces at each site.

Communications have also been added to existing large systems to connect them up. The most well known of these is Arpanet [MCQ77] but most manufacturers of large systems have a proprietary network offering similar facilities [WEC80,CYP78]. Connecting existing multi-user systems in this way perpetutates a mainly centralised mode of computing. Most work is done in the traditional way on a time-sharing system accessed through fairly unintelligent terminals. Files (and usually other resources) are available over the network, usually by specifying a host computer name. The filing systems of all the hosts taken as a whole form a sort of

network filing system, but it turns out to lack many desirable facilities. Only in the area of file textnaming is there any kind of coherence at all.

The mode of computing in which all these factors have really come together is the local area model of computing. Each user has a small personal computer. Sometimes these personal computers are allocated from a pool as required but they are not otherwise shared. This personal computer is of sufficient power to fulfil most of the computational needs of the user. Due to cost or inconvenience, other services are provided centrally. Line-printers are expensive, used only occasionally and most people would not want one in their office anyway. Such services are accessed over a local network. One other service which is both expensive and inconvenient is a filing system. Large capacity magnetic disc drives are both costly and noisy. However, this is not the main reason for providing such a service centrally.

Users preserve the state of their work in filing systems. Usually, when a user turns on a computer or logs into an ordinary time-sharing system, the only preserved state from the last session is held in files. If a user is to be able to work on any machine then these files must be accessible from anywhere. This is particularly important for high level language source files which may be runnable (after suitable compilation) on a range of machines of different characteristics which users may desire.

Users cooperate and share data. Usually, the only uniform way in which this can be done is to share files containing the data. This again means that any file must be accessible from any personal computer.

This can be achieved either by having local discs on each personal machine and making them accessible from anywhere, or else by having a central filing system accessible from anywhere (or possibly a mixture of both). The first approach is very

unattractive. As has already been mentioned, large discs are too costly to give each user a sufficiently large one, and too noisy to go in offices beside the personal computer. However, the biggest problem is that a user is free to run any program in a personal computer, to crash it, to write experimental operating systems and so on. Only in the case that all personal computers run a reliable protected unstoppable operating system is it possible to guarantee reasonable access to any file from anywhere. This probably only happens with large time-sharing systems leading to the centrally oriented model of computing already discussed.

The second approach, where the filing system is provided independently, is much better. The fact that it is provided centrally does not mean that it is provided by a single central file server, merely that it does not comprise the personal machines of users. As has already been seen, local high bandwidth communication is acceptably cheap. Long distance communication at the same bandwidth is prohibitively expensive, but if a much lower bandwidth is used the cost becomes acceptable. Rooftop satellite communication should offer high bandwidth cheap communication, but the transit time from sending a message to receiving its reply remains long. A large institution thus ends up with a number of high bandwidth local networks connected by a much lower bandwidth intermediate network, probably rented from the telephone company or some other third party.

Clearly it makes sense to have at least one file server on each local network since the communication overhead of accessing a remote network is considerable. In order to share processor cycles, memory bandwidth, channel bandwidth and so on, there are even good reasons for having multiple file servers on the same network. These servers may cooperate to varying degrees. The filing system which they present to a user at a personal computer is the network filing system.

Note that the main motivations for having a network filing system are not economic but because files are objects which it is desirable to share between users and between client machines. The economic arguments merely make this sensible.

## Hardware

Before going on to consider the design of a network filing system, it is sensible to examine the hardware to be used to implement it in more detail. We have already seen how the change in the ratio of the costs of computers and high-bandwidth communications has changed the ground rules for doing computing. Similarly, details of the performance of the computers comprising the system, of the network and of the storage media all impact on the design to some extent.

We assume that the file servers are computers of ordinary power. In the future, what constitutes ordinary power will change, but at present the fastest single chip (hence suitably cheap) processors run at about one million instructions per second, with reasonable memory space, say one quarter megabyte. These figures are vague since it is only their order of magnitude which is important. It would not impact much on the design, merely on the performance, to use a 10 mip machine with four megabytes of memory. It is actually very difficult to make use of large amounts of memory in a file server since, for robustness, it cannot be assumed to be preserved over a crash, and hence can only hold transient data en route for the stable storage.

A machine of this speed can cope with the filing system demands of many users, but not necessarily of all users on a particular network. This fact is likely to remain true as the cost of high performance computers continues to drop. Although this means that

the performance of the server will improve, the services demanded are likely to increase since the performance of the clients will likewise have improved.

We make very few assumptions about client machines. They may be single user microcomputers, single user high-performance personal computers with virtual memory, ordinary time-sharing systems and so on. In particular, we do not assume that they are all the same, nor that they all run the same operating system. Particularly importantly, we do not assume that client machines have local discs. In general, we do not assume that the operating system which they run is to be trusted although we may relax this for systems where the operating system provides as safe and controlled an environment as the computers comprising the filing system. In this case, which probably only occurs for a large traditional time-sharing system, we might implement all or part of a file server as a process in the system.

We assume that the network is inhomogeneous, consisting of a number of local networks connected by long haul transport services, perhaps using satellite circuits. We will describe a file server on the same local network as the client as a local server, and a file server on a different local network as a remote server. We assume a bandwidth on the local network to be between one and fifty megabits per second, with little more than transit delay between sending a message and its being received, perhaps 50us. On the other hand, access to servers on remote networks may vary from about 100us (where the two networks are intimately connected) to 10 seconds (where the two networks are in different continents) and the bandwidth may vary from a hundred kilobits to a few megabits per second.

The network is used by transmitting datagrams, messages of a few bytes up to a kilobyte or so which either arrive correctly at their destination or are detectably incorrect on arrival. There is no restriction on the data part of the datagrams that the client may

inject into the network; this non-restriction is very important. Sites are distinguished by network addresses, one address for each site. We assume that the network is capable of delivering a datagram to any accessible network address without the sender providing routing information. We assume that the network addresses in a message are reliable and that it is not possible, for example, for a malicious client to masquerade as a file server. Thus, although we make no assumptions about the data part of datagrams, the control and routing part is assumed to be reliably handled by the network hardware. We will not be particularly concerned with how errors are handled, but merely note that one way which is particuarly convenient when the error rate is as low as in most local networks is to attempt the operation again if no reply is received within a suitable time limit. If the reply was lost rather than the initiating command message, then this will result in the operation being repeated. We will thus take care to make as many operations as possible idempotent; they have the same effect applied many times as they do applied once.

The storage media are magnetic discs or something similar. In particular, they could be write-once video discs or large bubble memories, or a mixture of all of these. The characteristics that are important are that access is not as fast as to main store, say 10-100ms. We assume that the storage is divided up into fragments which we will call pages. Attached to each page are sufficient error detection bits that the possibility of reading a bad page without the error being detected can be disregarded. We assume that the pages are small enough that they need not be shared between files, say 128 bytes to 4Kbytes. The characteristics so far described are just those of modern magnetic disc drives. In the foreseeable future things are not likely to change all that much. Current very large volume media all have a latency associated with the cycling of the storage, either mechanically in the case of magnetic and optical discs, or more directly in the case of bubble memories. Improvements in implementation have tended to lead to increased capacity rather than faster access, and

this is likely to continue. One change which will probably take place is that large page sizes will become more attractive since there is an overhead associated with each page.

A more contentious assumption is that the cost of the storage medium is low enough that disc pages are not an incredibly scarce resource to be recycled among users at every opportunity. This is the exact opposite of the corresponding assumption underlying the design of most filing systems today, which seek to reuse pages as early as possible. If the storage medium is write-once optical disc then this assumption is true in two ways. Firstly, a disc may hold about 4000 megabytes [LAU80]. Secondly, the pages cannot be reused anyway, so it does not matter if they are treated as scarce or not. In any case, there are currently 600 megabyte discs available (and 1200 megabytes on the way). A 600 megabyte disc has over one million pages (of 512 bytes) so even a one percent working margin is 10000 pages.

The current cost per bit is minimised for magnetic discs with a 475 megabyte Winchester drive. This costs just under 4,750 pounds, giving a cost of ten pounds per megabyte or a penny per kilobyte. Writeable optical drives are currently about 80,000 pounds and so they are still unacceptably expensive. Even at this price, though, the cost per megabyte is only twice that of magnetic discs. Very thin film head technology arising out of integrated circuit fabrication techniques may push the density of magnetic discs up until a 10 platter magnetic disc volume holds the same as a single platter laser optical disc. Irrespective of which techniques become very cheap, it seems likely that the cost per megabyte of stable storage is likely to continue to fall.

The way in which laser optical discs may be used to construct a filing system is an open question which will have to await their arrival at acceptable cost. One important feature is that although disc transports which can write the disc are expensive, read-only transports are very cheap due to their development for the domestic

entertainment market. At about 400 pounds, a read-only transport for thousands of megabytes costs less than the disc pack for a 100Mb disc drive. An enormous filestore could well be constructed with a number of read-only transports and one writing transport.

Problems

The main problems associated with the implementation of a network filing system stem from two distinct causes.

Firstly, client machines run operating systems about which nothing can be assumed. They may range from well-behaved, through unreliable, to deliberately malicious. Further, since the filing system has no control of the messages which such a system can send, the only reliable identifiers in messages are identifiers issued and controlled by the filing system itself, unforgeable identifiers such as passwords and capabilities, and identifiers which do not, in themselves, convey any privileges, such as file textnames.

For example, it is not sufficient for a client system to check a user's password and then merely tell the filing system that each command is on behalf of user "John". A malicious user could adapt the operating system so that it always says that commands are on behalf of "John", or even on behalf of the system manager. Either the user must log on to the filing system and receive back a token controlled entirely by the filing system, or else the password (or a similar token) must be included in each command to the filing system.

However, the designers of client operating systems want to present a view of the filing system which best fits the philosophy of the rest of the client system. This needs to be done without compromising the uniformity of the network filing system. Any

parts of the implementation which can sensibly be devolved to untrusted clients without compromising the safety or the uniformity of the network filing system should be so devolved. Furthermore, the network filing system should be designed in such a way as to maximise the number of such parts.

The second major problem in the design of a network filing system is that we are attempting to create global entities out of loosely coupled local implementations. This problem seems to be a fundamental one in the design of decentralised systems. Efficiency depends on locality but the construction of global abstractions conflicts directly with this. For example, we will want a network-wide file security scheme. Yet we want to achieve this in a way that enables each file server to autonomously decide whether or not to allow a file access to proceed. This rules out centralised user validation, but we would also like to avoid the need for every file server to know about every user.

This thesis refines the statement of these problems and proposes solutions to them in various areas. The problems are not considered directly since they are not soluble directly. Rather they will crop up repeatedly when considering different areas of the design of a network filing system, each different area requiring different compromises and so a different solution.

Solutions require an environment in which their goodness or badness can be assessed. This is provided by chapter 2 in which criteria are developed to guide and assess the design of a network filing system.

Chapter 3 examines other existing network file servers and filing systems in the light of these criteria. The early server which we built [DEW77] is examined in detail to elucidate the lessons learnt from its design and construction.

Chapter 4 considers what data transfer primitives are suitable for a network filing system. That is, what is meant by the concept of a file and how it is transferred between client machine and the servers which implement the network filing system.

Chapter 5 considers the problem of naming of files. It considers what kind of naming scheme is appropriate for a network filing system, and what compromises are necessary to keep the implementation tractable. These compromises are mainly concerned with increasing the locality of the file servers. Location of files amongst all the file servers of the filing system is also considered in detail.

Chapter 6 examines the problem of file security. This is mainly concerned with the implementation of a security scheme which does not depend on central validation, nor on the trustworthiness of client systems.

Chapter 7 develops the notion of consistency and a scheme to control it. It also considers the problem of how to handle multiple versions of files. The problem exists in coresident filing systems and in single server network filing systems, but to an extent problems can be minimised by judicious implementation tricks. In a multiple server filing system, consistency has to be controlled properly. Happily, this turns out to be acceptably cheap.

Chapter 8 looks at how local discs on client machines can be used to improve performance. Unfortunately, since client systems cannot be trusted, these discs cannot be used to hold the current copy of any file (the client may choose not to give it back when it is required elsewhere). The nature of the files which can sensibly be cached at clients is considered in detail.

Chapter 9 describes in detail an implementation of the more contentious ideas of the preceding chapters. This centres mainly on the control of consistency, which is the most difficult area to implement. Given this control, the implementation of the other areas is relatively straightforward.

Lastly, chapter 10 reviews all the earlier chapters. It assesses how well our original design criteria have been satisfied by the solutions proposed. Finally, it considers where the solutions are unsatisfactory and where further research is required.

## Chapter 2: Criteria for a Network Filing System

In any design it is necessary to have criteria by which to judge the goodness or badness of the design, a sort of shopping list of desirable features and, perhaps, a list of horrors to be avoided. Inevitably, some of these features will conflict and a final design will be a compromise.

The first, and most important, criterion is that the network filing system is a filing system. Different programming languages and their run-time systems, different operating system and different users have various views about what consititutes a filing system. Nevertheless, these views have much in common. A filing system implements an abstraction called a file which is a repository for data. Files may be read and written (perhaps by mapping them into virtual memory). The filing system is responsible for the administration of the storage space required to preserve files, and for the order in which competing reads and writes are scheduled to the storage medium. The physical characteristics of the medium are hidden from the user, except in terms of performance. The filing system provides a means of textnaming files with character strings. We will use the filing system of Unix [RIT74,THO78,RIT78] as an example filing system since it is widely known and information about it is readily accessible. Other filing systems would do equally well.

The important factor is that the details of the implementation of the abstraction of a file are hidden from users of the filing system. This is in contrast, for example, to a database system where the division of a magnetic disc into cylinders might be central to the placement strategy of the database manager; there the abstraction is not hidden. It is also in contrast to very early filing systems in which the user had to specify the physical layout of the file, perhaps even to the level of the actual disc addresses. Further, the abstraction of a file textname hides the

location of a file from users.

We certainly want to support all notions of files visible in high-level programming languages such as Pascal, Fortran or Algol68. This gives us our first criterion:

* The existence of a network filing system has no implications for application programs. A program written in a high-level language will run on an operating system using the network filing system in the same way that it runs on a traditional operating system with a coresident filing system.

This does not mean that there are no implications for operating systems or for programs on the borderline such as command line interpreters. It merely means that it is possible for operating systems and, perhaps, the command line interpreter to provide an environment in which application programs can run. The implementation of such operating systems may require additional primitives to those of a coresident operating system communicating with its filing system. Some of these additional primitives may be desirable but more easily avoidable when the filing system is coresident. For example, a coresident filing system is restarted when its host operating system crashes; a server in a network file system does not (and nor does a client operating system if a server should crash). This means that extra commands will be required to allow clients to communicate to the network filing system the fact that they have recovered from a crash.

The second criterion for the design is that the filing system should be efficient. The efficiency of the filing system dominates many activities which users carry out. A computationally heavy operation such as compiling a program is dominated by the time taken to load the code of the compiler, read the source file and write the output file. On most systems, a source program has to be longer than about 100 lines before the compilation time is not dominated by the time taken to load the code of the compiler. Much

file activity is of this nature, so it is no good providing elegant features at the cost of the efficiency of basic operations such as reading a file.

One particular consideration of this sort is that we do not want the efficiency of using a local server to be impaired by the fact that we may access files on a remote file server. We naturally expect a performance degradation when we do access remote files. This is analogous to our expectations with telephones. We do not expect the fact that we can direct-dial international calls to affect the speed with which local calls are connected; correspondingly, if we do make an international call we do not expect it to be connected at the same speed as a local call.

Summarising these efficiency considerations gives us our second design criterion:

* The network filing system should provide similar performance to a coresident filing system, at least when accessing a local server. The existence of other servers in the network filing system should not degrade local performance.

So far our criteria have insisted on an efficient filing system without any implications for how the different servers should cooperate together. We want the different servers to work together to implement a single filing system. It is not sufficient for each server to implement an independent filing system, with some escape from the naming convention to nominate a file on another server. Our servers are more tightly coupled than this.

One key implication of this is that there should only be one textnaming scheme for files. The name of a file should not depend on the location of either the file or of the client attempting to access it. It should not normally be necessary to specify (or even know) the location of a file. Occasionally this specification may be necessary to create files at specific sites - on a removable

storage volume, for example.

The other major implication is that there should be a uniform scheme for permitting and restricting access to files by other users and programs. This is bound up, to some extent, with how access is controlled in the network to other objects which may be protected; network addresses or vector processors, for example.

Another implication is that the guarantees and facilities available should not depend on the location of files (again, the performance may). Since it is not normally necessary to know these locations, it would be an anomaly for their treatment to vary from server to server.

Furthermore, we do not want to restrict the facilities which we provide in order to make it easy to satisfy this criterion. The network filing system should offer similar facilities to those obtainable from other good filing systems. It should be state-of-the-art.

This requirement for divorcing the abstraction of a file from its location leads to our next requirement:

* The servers of the network filing system cooperate together to give the illusion of a single coherent filing system. The guarantees and facilities provided do not vary with the location of files. It is not normally necessary to know the location of a file in order to access it. The facilities provided by the filing system are not restricted to make this criterion easy to satisfy.

Our next criterion works against the previous one in a direct way. Although we want the illusion of a single coherent network filing system, we want to achieve it without losing too much locality. In particular, we should avoid any sort of centralised manager. Apart from obvious reliability issues, a centralised

manager is inevitably remote from most of the servers in the network and so it will significantly degrade performance. In fact, we want the different servers to be largely autonomous, only communicating when they need to synchronise operations.

The network filing system design should not be predicated on there ever being a time when all the hardware and software comprising it is working simultaneously. For example, crash recovery should not depend on all other servers being accessible. It will often depend on some servers being accessible to pick up the pieces of incomplete operations, but the crashed server must be able to decide which they are without reference to them. Crash recovery of a server in London should only depend on a server in New York being accessible if they were cooperating near the time of the crash.

Correspondingly, it should not be assumed that there is ever a time when all the servers can simultaneously be taken out of service. This might be desirable in order to take a backup snapshot of the entire filing system, or to release a new version of the file server code. However, all these operations must be performed on a local basis. In particular:

* The servers should be largely autonomous. There should be no dependence on all the servers ever being simultaneously accessible, nor on all the servers ever being simultaneously available to be taken out of service.

Underlying all the criteria for a global filing system with local efficiency is the assumption that most access to files will, in fact, be local, normally accessing only one server. This is an assumption rather than something which is inevitably true, but it seems reasonable. It is partly justified by network transit times which make efficiency of access to remote servers relatively unimportant. The time taken by the server to perform its part of a request is a smaller proportion of the time taken to access a

remote server than of the time to access a local server; the bulk of the time for a remote access will be network delay.

So far we have assumed nothing about the client machines. That we do not is important if the network filing system is not to be unusable by small machines or too restrictive for large ones. We cannot assume that client machines all have their own local discs but it is unreasonable to assume that they definitely will not; they may or may not have local copies of some files. We cannot assume that clients have any form of virtual memory but it is unreasonable to assume that they do not; they may or may not map files as virtual memory segments. We cannot assume that they are or are not very powerful computers and so on. Clearly we need some assumptions, that they have a network connection for example, but these have to be a sort of lowest common denominator of the computers which may be used as clients.

Further, we do not want to make assumptions about the messages which a client may send. Local area networks tend to be open, in the sense that any computer on the network can inject any message into the network. Whether the receiving site chooses to respond to the message or ignore it depends on agreement between sender and receiver. Whether the two parties trust each other requires more than agreement. A client can trust a file server since it is assumed to be impossible to run other than the ordinary file server program in the file server machine. Sending a password to a file server is thus safe, since it is assumed to be impossible to substitute a password collecting program for the file server program. However, the server cannot trust the client since no such dual assumption is reasonable. This is very similar to the relationship between an ordinary operating system and a programming language run-time system. The operating system cannot trust the run-time system since, although it is probably reliable, it is under the direct control of the user who could choose to amend it.

The requirement that we do not assume that the clients have vast resources, nor that we assume that they do not have such resources leads to our next criterion:

* The design of the network filing system should make no assumptions about client machines. In particular, no assumption is made about whether or not they possess such facilities as local discs, virtual memory, large amounts of real memory and so on. No assumptions are made about the security or reliability of operating systems running in client machines. No assumptions are made about the validity of data contained in messages from clients.

Finally, we want to avoid the servers of the network filing system doing all the work whilst the client machines wait idly. The servers are shared resources and there are few of them, compared to the clients which are unshared and comparatively numerous. However, we do not want to do this distribution at the cost of having to trust untrustworthy client systems. This means that sometimes work has to be done in the server rather than in the client. Deleting a file has to be done at the server; providing nicely formatted file directory listings does not. When there are no compelling reasons to do a task in the server then it should be done in client machines (or, perhaps, in a separate server provided for the purpose). Summarising this criterion for maximising functional distribution:

* As much function as possible should be devolved from file server to client machines; the file servers should not provide services which could be provided elsewhere without compromising security or consistency.

The criteria which have been developed can be briefly summarised as follows:

No implications for application programs;
Access to a local server is efficient;
A single coherent state-of-the-art filing system;
High autonomy of each server;
No assumptions about client machines or systems;
High functional distribution to untrusted clients.

All the criteria seem to be necessary, since removal of any one of them permits implementations with undesirable features. They are certainly not sufficient on their own (which would imply that only one filing system could satisfy them all). Other minor criteria will be discussed when choices between alternatives need to be made.

With suitable alteration of the third criterion, the criteria form a basis for assessment of the design of a wide range of services which might be provided by a network based operating system. They are not tuned to present a particular type of network filing system favourably, but rather they outline how servers should be constructed in the environment of a large open network.

These criteria taken together outline the network filing system. Locally efficient servers implement a single filing system with global facilities. This is used by client systems which are not trusted by the file servers but which are, nevertheless, expected to play their part in implementing the view of the network filing system seen by application programs and users at terminals.

## The problem areas

These design criteria lead to a number of problems in different areas of the design. Quite a few of these arise directly from the fact that we are dealing with a distributed solution, both because the filing system is devolved from the client and because the filing system is itself distributed amongst a number of servers. These problems will be elucidated briefly before considering each area in detail in the following chapters.

Part of the last criterion, that clients are not trustworthy, means that a network filing system is different from a coresident filing system. Most coresident filing systems trust their operating system. For example, a filing system may provide the capability to mark a file of code as being executable but not otherwise readable; it then trusts the operating system to tell it whether the code is to be read by some form of loader or by another program (perhaps an unassembler or a copy program). A network filing system cannot trust client operating systems in this way since it would be relatively easy to write a program for a personal machine which masqueraded as an operating system loading the code.

The requirement for efficiency means that the primitives used to transfer the data in files are important. After all, reading some data from a file is likely to be the most common operation which the filing system will be called upon to perform. The requirement for devolving functionality conflicts with attempting to provide a large number of sophisticated primitives. Clients can themselves find the next newline character or unblock logical records. However, the desire for a single coherent filing system means that we want the data to represent the same information on all systems. A file of text is something every system supports, yet its representation in terms of bytes within a file may vary. This problem is considered in detail in chapter 4.

The criterion for a single filing system leads to a single textname space for naming files. If no restriction is placed on naming then it is very difficult to decide when a file is no longer named and so no longer required. If too much restriction is placed on naming then the demands of some operating systems may be unsatisfiable. Further, we have to be able to locate a file given its textname. We do not want to force the textname to contain the site at which the file is to be found so this is not trivial. Normally it will be found locally, but a strategy is needed to cope when this is not the case. This problem is considered in detail in chapter 5.

Security is a problem. Identifying users in a distributed environment (as opposed to identifying the client machines they may happen to be using) is difficult. Identifying any sort of privileged program is even more difficult since it has to carry its identity around within it. This means that a security scheme based on permitting file access to certain classes of users or programs is tricky. If we use a scheme in which users do not need to be identified directly, by using some form of unforgeable file token, then we either need multiple tokens for different access to the same file, or else client operating systems have to trust each other to restrict file access in an agreed uniform way. We want as much as possible of the filing system to be devolvable to client machines but this conflicts directly with the requirement for security. If there is no security then the filing system only has to check the legality of requested operations. If there is no distribution of function then security is implemented entirely within the network filing system and so it is a much simpler issue. This problem is considered in detail in chapter 6.

Preservation of consistency when operations are split amongst several servers is difficult. A coresident filing system takes some care to order writes to discs so that a crash normally has minimal effect. When only a single server is involved it can do the same. However, we require that the guarantees are not

different when files are split amongst several servers, so consistency has to be controlled more explicitly. As an example, consider creating a file on one server and putting its textname in a directory on another. If a crash occurs we either want both to happen or neither to happen. This problem is considered in detail in chapter 7, and a detailed implementation is described in chapter 9.

File accounting is difficult in a filing system with a generalised textnaming scheme and it is even more difficult in a network filing system. Accounting may be desirable, either for charging for filespace or to encourage frugality. However, as the cost of stable storage continues to fall it will make less sense to rent filespace from a third party, and wastage of filespace will be less of a problem. Indeed, for a write-once laser optical disc the space charge would need to be the total number of pages ever written. It would be difficult to reduce the charge by deleting files; since this would probably rewrite a directory it would increase the charge! While acknowledging the problem, we will not consider accounting further.

Making sensible use of local discs on client machines without trusting their operating systems is difficult. If the local discs are used for holding copies of writeable files then there are some problems. Firstly, when does the written version drift out to the network filing system and how does the network filing system keep track of the current version of a file, or keep all copies up to date? Secondly, how can the client system and the network filing system cooperate to provide the same facilities and guarantees as when the files are not cached locally? These problems are considered in detail in chapter 8.

The five main problem areas then are:

*data transfer primitives;
*naming and location of files;
*security of files against unauthorised access;
*preservation of consistency in the face of crashes;
*increasing performance by using local discs at clients.

## Chapter 3: Some Existing Network Filing Systems

The term file server covers a huge range of facilities. Most people would not consider either a multiply ported disc controller a file server, nor an ordinary time-sharing system accessed over a terminal network a file server. In between these examples are servers which provide a virtual disc, through to servers which provide a complete filing system together with some commonly used file transformation programs such as editors. These are all considered to be file servers but the distinction near the extremes is fuzzy.

This section examines some existing network filing systems, and particularly how they cope with the problems of transfer primitives, naming, security and consistency.

### Early file servers

The earliest file servers were single servers providing one fixed filing system [DEW77,COL72,NPL77]. The filing systems implemented made little attempt to be general. Rather they provided a set of commands to manipulate files at about the same level of complexity and with about the same generality as a filing system coresident with its operating system. They provided a textnaming scheme for files and implemented some form of security control. They all have the concept of users; that is of people who have confirmed their identity by logging-on to the file server and quoting a password.

Providing a single filing system in this way has a number of advantages. Firstly, it is simple to implement and to understand. There is a large body of knowledge about implementing filing systems of this type. Secondly, it imposes the minimum load on the

client systems that support it, both in the amount of processor bandwidth used to access the file server, and in the amount of code used to implement this. Until recently, most clients had either restricted processor bandwith or restrictive memory limits (or both) and so moving the hard work across the network to the file server machine made sense. The present generation of microprocessors already offer much larger address spaces than previous generations, and are much faster. The next generation will be more so. Limitations of the client hardware are no longer a good reason to centralise everything for it may be the processor bandwidth at the server which will be under pressure.

The main disadvantage of the early servers is that no attempt was made to be general. If there is only one way to view the filing system then it may constrain client operating systems too much. This is especially so when an operating system already exists with a coresident filing system.

## A filestore system

One filing system of this early type is the filestore constructed in the computer science department at Edinburgh University [DEW77]. Other early file servers were very similar in most respects. We provided a complete simple filing system including textnaming of files and a security scheme. It was lightly based on the filing system for Titan [BAR67].

The network consists of a star of point to point connections using 2 megabaud byte serial links, with the filestore at the centre of the star. There was no provision for clients to communicate with each other through the filestore but neither was it assumed that clients did not have other channels of communication.

There are two completely different data transfer primitives corresponding to the two different types of file organisation. Both organisations impose no structure on files other than that they should be a sequence of bytes, although these bytes are divided into 512 byte pages. Both data transfer primitives have the concept of an open file. A file is opened by a command which provides the file textname; the filestore returns a small integer channel number used in further commands. The file is locked under the usual multiple reader single writer discipline. After it is opened, a contiguous file is accessed by two commands, read and write, which fetch a specific page from the file or inject a specific page into the file. Most files are only accessed sequentially and in this case the commands give no page number; the filestore records the current page. Two files of the same name may be read and written simultaneously. When the file being written is finally closed it replaces the file being read which is then deleted. If the file has other readers at this point, or at any other point when it is to be deleted, then the file contents (but not the file textname) are preserved until the last reader has departed. The two read and write commands for these sequential files fetch the next page from the file and add another page onto the end of the file. The last page only of a file may be shorter than 512 bytes.

Originally there were no commands to alter the current page pointer, but two commands have been added. For a file open for reading the pointer may be reset to any page in the file. For a file open for writing, the last page may be read and removed from the file. This last command is the only way that a sequential file may be read and written at the same time, without first closing and reopening it. It is motivated by the need for simple text editors in clients with very limited memory space.

Textnaming is a two level scheme, the first component normally being the initials of the owner of the file. There is a scheme for defaulting the owner of files where it is not specified. Each

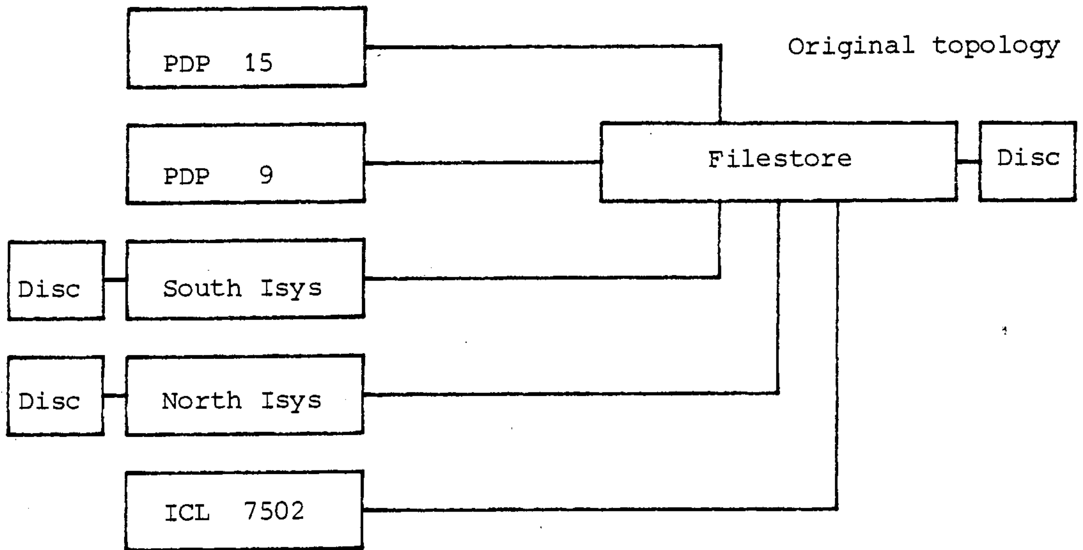
owner is limited to 64 files.

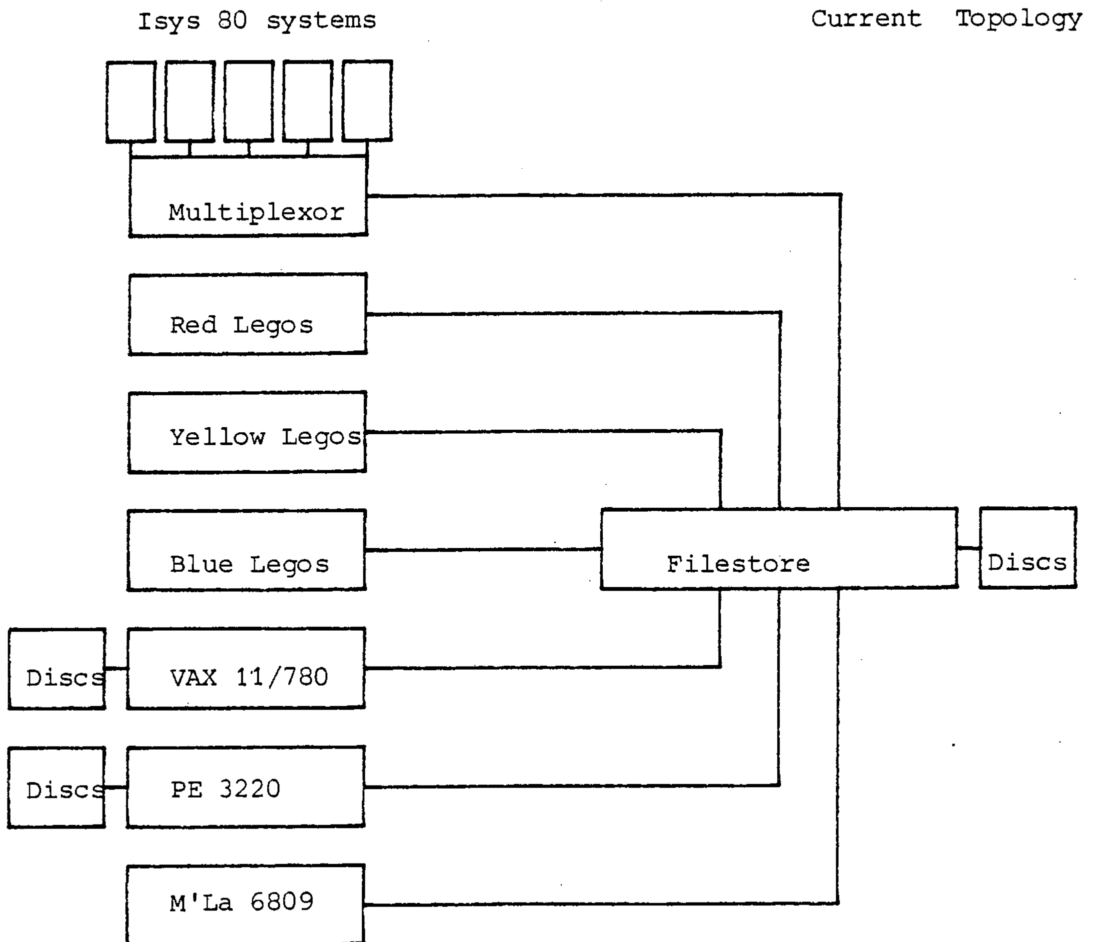
Security is controlled by means of passwords. The filestore has the concept of a logged-on user. Logging-on is achieved by presenting the initials of the owner and a password. However, anything which an owner can do can be done by another owner if this password is quoted. Associated with each owner is another password which allows more restricted access to other users if they first quote it. This is roughly equivalent to a user group in many systems (for example, Unix). In common with experience with these systems, it is a rarely used facility.

At any time a user has two passwords quoted. One of these is the password used to log on to the filestore and the other is the current password. This means that it is not easy to run a program which accesses two files belonging to different owners. This would be much more of a problem if the filestore was not in an open environment where most people leave their files readable by anyone. Programs requiring access to private files are in an even worse situation since it is difficult for a program to safely quote a password in case it is interrupted. In any case, the filestore has no mechanism to return the current password (in fact passwords are encrypted by a hard-to-invert procedure), so a program quoting a password could not reset it and so it would be an inconvenience to users running it, who would find their current password altered. Any security scheme which relies on a program presenting its credentials must ensure that they have a strictly limited life. This is most easily achieved by allowing the credentials to be specifed with the command for which they are necessary. Otherwise it is very difficult for a program to run with the authority of its creator rather than of the person running it. Being able to do this solves many of the authorisation problems which arise in practice.

The filestore makes very little effort to preserve consistency. Because two files of the same name may simultaneously be read and written, there is no problem about loss of data since both the old and the new versions are in existence until the new version is complete. However, the transient file being written can be left around if client or server crash, requiring users to examine their directories to see what has happened. There is also a security breach here if the filestore should crash. At any time a file will, in general, be overallocated space and this extra space is removed when the file is closed. However, if the filestore should crash while the file is being written, then this extra space remains allocated to the file. Since disc pages are not initialised before being provisionally allocated to the file, this extra space will contain old contents of pages from deleted files of other users, preliminary copies of exam questions, for example.

Figure 1 : Topology of Edinburgh Filestore Network



Initially, the filestore supported two machines (a PDP9 and PDP15) at a virtual disc level (actually, a virtual dectape), provided an archive store for two machines with their own small discs, and provided the filing system for a small editing station which had no other connections (see figure 1).

At this level of use, few problems were found since the connected client systems did not exercise the filing system enough to reach its limitations. All its clients had either no pre-existing filing system or an even more rudimentary one. In any case, none of the clients with existing filing systems made any attempt to integrate the filing system of the filestore.

Used in this way, the filestore had no real impact on the way people chose to use the machines. The functionality available on the connected systems had only increased marginally.

Currently, the filestore supports about ten single user clients which have no other disc storage, and it is accessible by two time-sharing systems (see diagram).

The connection of further clients is limited by the lack of space in the processor chassis for the communication link interface boards; a broadcast network will solve this particular problem. At present it is partly solved by using another machine as a multiplexor.

This second generation of connected systems has had a major impact on the way in which people work. Processors which had been sitting idle in cupboards became the most popular machines on which to work, eventually superseding the existing local disc system. Noone would now construct a processor or operating system without considering how to connect it to the filestore. The filestore has thus been very successful; nevertheless experience with it has indicated several weaknesses in the design.

The major change in the use of the filestore came in its use when we wrote Legos [MCL78], a personal operating system which ran on clients connected only to the filestore, to a terminal and to an extra uncommitted communication link. This provided an environment which made all of the facilities of the filestore directly available to users. Several client machines ran Legos and it was used by research groups for the development of suites of application programs. Since Legos was conceived after the filestore, there was no conflict between its view of the filing system and that provided by the filestore; it merely inherited the filing system of the filestore and propagated its flavour through to users. Device names on Legos are local to each system; it is not possible to name the keyboard of another system, for example. The one area in which Legos needed its own textnaming scheme was for naming commands. Apart from a very few built-in commands, a command was just a file textname.

The biggest problem for program development was the poor textnaming scheme. The population of potential users has never seriously approached the limit on the number of ownernames but the restriction to 64 filenames per owner has been far too severe. Further, the lack of a properly structured naming system made equating file textnames and command names difficult when the user was working in a rich environment drawing commands from a number of places. Searching under several usernames is cumbersome and has often led to finding an incorrect file of the same name.

The lack of any client-definable file attributes caused a problem too. A command on Legos can either be executable code or a further file of commands, known as an obeyed file. The decision as to which was made by examining the first bytes of the file. The differences in the way in which executable code and obeyed files are handled are so substantial that it is a problem discovering so deep in the system which has been delivered. A system derived from Legos, Isys80 [DEW80], insists that the textnames of such files are extended to indicate whether they are to be executed or obeyed.

While this solves the identification problem, the number of file textnames to be considered when searching for a command is doubled.

As has already been mentioned, most files can only be read and written serially whereas a second class of files can be directly accessed on a page basis. However, the lack of any consistency guarantees ruled out use of such randomly accessible files for many applications which could have made great use of them. The problems about handling the files in a crash resistant way outweighed the advantages of using them. Consequently, large files were serially scanned in their entirety to extract small portions, and the whole file was rewritten to update small parts of it. File directories are not files and are updated in a manner not available to clients. Nevertheless, only very weak guarantees can be given about directories across failures and the filestore has once crashed whilst writing a directory, leaving a four page directory with one new page, one unreadable page and two pages remaining from the old directory.

Consideration of how to develop Legos further has exposed more weaknesses in the filestore design. All tokens generated by the filestore for logged-on users and for open files are associated with the client which requested the operation. Transmitting these tokens to another client, a compiling machine for example, is not sufficient to transfer the authority. The compiling machine will itself have to log on to the filestore. That these tokens of authority are not transferable makes devolving further function from client operating systems difficult.

From our experience of constructing the filestore, there are four areas of weakness that any network filing system should address.

Firstly, the textnaming scheme should be as completely without restriction as possible, and special types of files should be distinguishable.

Secondly, the behaviour of the filing system should be predictable over crashes of both clients and file servers.

Thirdly, tokens which the filing system generates for use in further commands should be transferable to other clients.

Fourthly, the authority and security mechanisms should make it possible for a program to run with the authority of its creator rather than the authority of the person running the program.

Finally, there should be clear consistency guarantees; that is, statements about what is and is not guaranteed if the server or a client should crash.

## Universal file servers

This generation of file servers [BIR79,DIO80,SWI79] attempts to solve the basic problems in a different manner. Rather than implement a complete filing system with textnaming, security and so on, they implement a universal filing system. This is a sort of kernel filing system which, hopefully, provides suitable primitives out of which it is possible to implement any filing system.

The first such system was WFS [SWI79]. The only abstraction which WFS supports is that of a file, an indexed set of 492 byte pages. On creating a file, a large integer is generated by the filing system and returned as a token with which to access the file. Any further access to the file required this token to be produced; producing the token is sufficient authority to access the file. The token is a 32 bit integer and so it is hard to guess. Unfortunately, losing a token precludes deleting the file it identifies. As one would expect, pages in a file may be read and written, new files created, old files deleted and so on.

The data transfer primitives are to read and write a page from a file. There is no concept of an open file although there is a scheme to lock a file for exclusive access.

There is no textnaming scheme; this has to be built on top of WFS. There is no real security scheme. Possessing the token for a file is sufficient authorisation to do anything to it.

Consistency is handled by giving sufficient limited guarantees to make it possible for a client to handle the consistency [PAX79]. Each command to WFS is limited in its effect on the disc. This makes it easy to make each command atomic, so that it either succeeds completely or fails completely (even over a crash). For example, each page must be removed from a file by the client, one command for each page, before the file may be deleted. The guarantee that each action is itself atomic makes it possible (but tricky) to construct larger atomic actions.

The main way in which WFS differs from its predecessors is that it uses a connectionless protocol. The commands are self-defining and require no context of logged-on users or opened files. This simplifies the amount of state information which needs to be maintained at client and server, and makes further distribution of applications amongst multiple clients easy. The one exception to this stateless protocol is the file lock. A file may be locked and a key is returned which gives exclusive access to the file (there are no shareable locks although there is nothing to stop a key being shared). If the file is not accessed for a minute or so then the lock is broken by the server on the assumption that the client has crashed or otherwise lost the key.

The Cambridge file server [BIR79,DIO80] also has a notion of a file, this time an unstructured sequence of bytes. Files are identified by 64 bit integers, 32 bits of which are random making the token probabilistically unforgeable. Unlike WFS though, there is a concept of a file index. Clients cannot put anything other

than tokens in file indices and are expected to pair off an ordinary file with each index for their own administrative data. If a file has no token preserved in an index then it will be deleted. Apart from this restriction, tokens may be preserved in any way.

As far as data transfer primitives, naming and security go, the Cambridge file server is similar to WFS. The protocol is about as connectionless as that of WFS. There is a concept of a locked file but a file can be accessed without explicitly locking it. Shared and exclusive locks are distinguished under the usual multiple reader single writer discipline.

As far as consistency goes the Cambridge file server is much better than WFS or the earlier servers. A file may be marked as important and in this case it is updated atomically; any writes made while the file is locked are either all performed or not performed at all when the file is unlocked. This is true even over crashes of client or server. However, no guarantees are made about atomically updating several associated files (other than that each file will be independently updated atomically). Naturally, all the files maintained by the file server itself, such as file indices, are marked as important (and hence updated atomically).

The vestigial naming graph maintained in the indices is not restricted in form and so cycles can be formed necessitating occasional garbage collection. This is done in a separate client machine without interruption of service [GAR80].

The concept of a universal file server as the kernel of a full filing system has been successful. To some extent, however, the lack of prescribed higher levels has resulted in a tower of Babel of higher level filing systems erected on the same universal filing system, none of which can communicate with any other. Since files on one filing system are not nameable on other filing systems they cannot be accessed. The only way to access a file on another

filing system is to drop down a level and use the token for the file. Unfortunately, doing this interacts unfavourably with the file security scheme.

## Distributed filing systems

There have been a number of distributed network filing systems and distributed databases built. Note that a collection of separate file servers all accessible from a client does not constitute a single network filing system. It is necessary that the servers conspire together to give the illusion of a single filing system. It should be possible to run a program on a client which takes a filename as a parameter and then nominate any file on any server.

It is worth noting that a distributed database differs in character from a distributed filing system. The purpose of a distributed filing system is to implement the abstraction of a file and hide as much as possible of the implementation; in particular, the implications of distribution. Filing systems are general purpose. On the other hand, databases are usually special purpose, to the extent that they can achieve considerably better performance by being completely aware of the implications of distribution. Further, a distributed database is not usually accessed directly by client application programs. Client application programs access a large database access program, possibly running in the client but almost certainly written by those responsible for maintenance of the database. The point at which the decomposition to the two sides of the network occurs is thus not really visible to clients and has sophisticated programs maintained by the same people on both sides. For example, central databases usually have some query processing which attempts to select the best way to answer a query,

often by deciding in which order the sub-queries should be dealt with. In a distributed database, the best way to answer a query depends on the distribution of the data around the various sites holding the database, especially when parallel processing of the subqueries is taken into consideration. This decomposition will be done by a complex program in the client or perhaps by one of a number of query servers. Nevertheless, some of the techniques used for implementation of distributed databases are applicable to the design of a network filing system.

One of the earliest recorded distributed filing systems was RSEXEC [THO73]. This did not run on special servers but on a number of Tenex systems [BOB72] connected by the Arpanet. The textnaming scheme spanned host boundaries and allowed a user to nominate files on any of the systems participating in the scheme. This is an example of a scheme where only the textnaming of files is distributed. The underlying filing systems have no communication with each other below the level of the distributed textnaming scheme. Data transfer, security and consistency are just those of the underlying Tenex filing system.

Most networks supplied by mainframe manufacturers, such as Decnet [WEC80] or IBM system network architecture [CYP78], have made some attempt to make nameable files at other sites accessed over a network. This is done by extending the file textname with a machine site name. The names are even less distributed than RSEXEC; for example, it is not possible to have a directory spanning several machines. The filing systems are actually completely independent, the extended naming being accomplished by a remote third party process with knowledge of both the remote filing system and the network. It is almost impossible to interface a different sort of filing system to the scheme since too many restrictive assumptions have been made about file formats, textnames and so on.

Another distributed filing system is DFS [STU80]. This is actually implemented on top of WFS described earlier and at the same level whereby files are accessed by 32 bit tokens. There is no textnaming or security scheme; these must be built upon DFS.

DFS is oriented towards distributed databases. The motivation for its construction is the variation in processor power to disc storage space available by splitting the storage space among separate processors. Thus a DFS system is expected to consist of a number of servers on the same network. It is a little dubious whether the gains in processor bandwidth achieved by having multiple processors are not outweighed by the communication required to coordinate them all. It certainly requires a very sophisticated client application program to cope with the complexities of the locking and unlocking of data, and with the unsolicited messages which the servers may deliver at any time.

The main difficulty of using DFS is the complexity of the file locking scheme. DFS allows locks to be broken under many circumstances and it sends unsolicited messages to clients telling them that this has happened. The client is then expected to reread the data to see if it impacts on the results being written. There is no way that this can avoid being pushed up for the application program to handle. Normally, this application program will be a database access program handling queries on behalf of the real application program and so it will be specially designed to cope with these complexities.

The primary facility which DFS provides is good control of consistency. This is achieved by providing the concept of an atomic transaction [LAM79,GRA77,BER79,ESW76], a series of reads and writes which are performed indivisibly and without interference even in the face of crashes and competitive access by multiple clients. The concept of an atomic transaction is becoming accepted as a good general way of controlling both crash and deadlock control, and of administrating competing access among multiple

clients.

There are two parts to implementing atomic transactions, one part corresponding to each word. Firstly, there is the problem of ensuring the atomicity whereby the transaction succeeds in its entirety or leaves no trace of its passing. These two cases are described as the transaction committing or aborting respectively. Secondly, there is the implementation of a transaction so that two concurrent transactions competing to access the same data do not interfere.

In a distributed system there are two stages to building the atomicity. Firstly, each site has to be able to perform actions atomically, and secondly we have to ensure that all sites make the same decision about commitment or abortion of the transaction.

Single site atomicity is concerned with taking a non-atomic storage medium and making it atomic. The problem is to do this without incurring unacceptable inefficiency. One suggestion [LAM79] is to write all critical pages to two connected places, and to choose either of them. If one page is unreadable then the other is used. The algorithms have to be sufficiently incremental that using either of the two pages is not incorrect when the pages are not identical due to a crash interrupting the writing of the second page. Writing everything twice seems unattractive in a filing system. In a database an update is usually very small and localised so this approach is more reasonable whereas in a filing system it will often be a whole file or a large part of a file [WIL72]. This will be true even in the common case where the change is small and local - deleting a single line from a 1000 line source program, for example.

Writing everything twice is only necessary since a page cannot be reliably updated in place. If pages are always written to new sites then an unreadable page is not a problem and the number of writes to achieve atomicity is halved. However, more storage is

used although suitable reclamation schemes can be devised easily. This technique will be expanded on later.

One motivation for writing everything to a new site comes from the prospect of write-once video discs where this is the only option. After all, reusing the same pages is an economic consideration based on the fact that pages are considered a scarce resoure. Removing this assumption leads to a policy of non-deletion [SCH77,COP80,REE79] in which everything is preserved forever (or as near to forever as can be managed).

Keeping everything forever leads to a naming problem which is most easily solved by using timestamping [REE79,THO76]. This leads to the problem of distributing time to perform these timestamps. A complete analysis [LAM78] leads to the conclusion that synchronisation is more difficult the more erratically the message transit times vary. The variation in transit times will normally correlate with the distance between the communicating sites. This will turn out to be just what is required, since the size of the error in synchrony that can be tolerated is closely related to how frequently the sites can interact.

The second part of implementing atomic transactions is the concurrency control. The non-interference of two competing transactions is defined in terms of serialisability, whether or not there is some serial ordering of the transactions which has the same effect [GRA77]. We would like a scheme which only holds up transactions when they would otherwise violate this serialisability condition but unfortunately no such scheme is known [BER79]. There are a number of schemes which guarantee that competing transactions will not interfere, although they all sometimes interfere when serialisability is not at risk. Some of these involve a transaction announcing in advance all the data which it will access, allowing the scheme to decide whether this conflicts with other transactions already started. Such a scheme is not useful for a network filing system since, in general, a program only knows

in advance what accesses will (or at least could) be made if opening a file from within a program is not possible. For example, it would be impossible for a compiler to implement a scheme for including a declaration file nominated from within the source text of the program.

Other schemes abort one of the transactions when a conflict occurs, on the assumption that such conflicts will be rare. This is not an attractive solution for a filing system either. If two clients attempt to edit the same file then we would like to see one of two things. Either the second client is told to wait until the first has finished, or the second client merely receives no reply from the file server until the first has finished. Which we want to happen will probably depend on whether there is a person waiting or a program waiting. What we do not want to happen is that the second user be permitted to edit the file and then have the edits thrown away on attempting to leave the editor.

The most widely used scheme, since it permits the locking of data to take place as it is needed, is the two-phase locking scheme [ESW76]. In this scheme, data is locked whenever it is accessed by a transaction and has not yet been locked by that transaction. These locks are held until the transaction commits. The way in which this interacts with other aspects of the filing system will be discussed in more detail later.

## Summary

We have looked at the state of the art in network filing systems. These have either been single server filing systems or else oriented towards the construction of distributed databases. The construction of a distributed network filing system inherits problems and some solutions from both of these areas. We reiterate our five main problems.

The first problem is that of the primitives provided for the transfer of data. This is bound up with precisely what is meant by a file. Network filing systems have tended towards the view of a file as an unstructured vector of bytes, but we shall see that this does not solve all the problems when a file is accessed by more than one client operating system.

The second problem is that of textnaming files. Little work has been done on distribution of textnaming and even many traditional filing systems coresident with their operating system make it necessary to know on which disc volume a particular file resides; some even make it necessary to know on which drive the volume is mounted.

The third problem is security in a network filing system. This is not particularly affected by the number of servers comprising the filing system but by the desire to functionally distribute as much as possible to the client machines without their having to be trusted.

Next, there is the problem of maintaining consistency. This is a problem in any filing system but it has rarely been properly addressed except in database systems. Crashes are more likely in a distributed system than in a centralised one, although any crash should have a less disastrous affect. Problems with the preservation of consistency are thus more frequent than in a centralised filing system.

The fifth problem is how to make good use of local discs at clients. This problem is meaningless in the context of a coresident filing system, although a similar problem can arise there when considering how best to use very fast fixed head discs or drums. The problem lies in resolving how to increase performance and autonomy of client systems without requiring them to be trusted.

There are, of course, other problems common to both network filing systems and to coresident filing systems. For example, what unit of allocation should be used for disc space and how should it be chosen from all those available? That these points are not considered is not to imply that there is an acknowledged best solution, but rather that any solution is unaffected by the problems of distribution.

# Chapter 4: Data Transfer Primitives

The prime purpose of a network filing system is to store data from client machines and redeliver it on request. How the data is presented to the file servers, and how it is retrieved will have a major impact on the performance of the network filing system since these are far and away the most common operations. Our design criteria include a requirement for good performance.

There are two main problem areas. Firstly, the file server is a shared resource and so it may well not be sensible to use it for computationally intensive operations such as unblocking records, which could equally be done in the client machine. Secondly, different operating systems use different data representations for the same information. Since one of the purposes of a network filing system is to make sharing of file amongst different users and machines easy, it is a pity if the contents of these files are not so easily shared.

## Where to do the work

A file consists of some pages in the stable storage of a file server. These pages contain bits of data which are a representation of some information structure. Turning pages of bits into information may involve transforming it in some way, and preserving a certain amount of state information about what has already been transformed.

For example, Unix [RIT72] allows an arbitrary sequence of bytes to be read from the current position of a file. Assume someone wants to access a few bytes from the middle of a file which has not previously been accessed. First the file is opened, a seek is

performed to the required position in the file, the bytes are read and the file closed. In more detail the following has to take place. First, the file textname is resolved to get an internal file number. This internal file number is checked to see whether the requestor should have access to the file. The in-store data structure of active files is examined for a conflicting concurrent access. The housekeeping data for the file is set up in an in-store data structure, and a current position variable set to the start of the file. The seek is received, which merely alters this current position variable. Next, the read is received. The current position is examined to see which page or pages will be needed to satisfy the request. A check is made to see if these pages are beyond the end of the file. The pages are fetched from the disc. A check is made to see if the last bytes requested are beyond the last byte of the file should the last page be amongst those fetched. The actual bytes of data are extracted from the page. The current position is updated to the byte following that delivered. Finally, the file is closed by updating the in-store data structure.

This immediately raises a number of points, some of which are due to problems with naming and security which will be covered later. Should the file server have the concept of an open file or should the internal file number be provided in each request? Should the client or the file server maintain the current position pointer? Should the client or the file server extract the bytes required from the pages which contain them? Should the file server have a concept of the length of a file, and if so how is it determined when a file is created or updated? These questions are much simpler if we only have to deal with Unix-like systems, but this is not necessarily the case since we are not making such restrictive assumptions about client operating systems.

However, all systems are not like Unix. Multics [BEN72] and, more recently, Pilot [RED80] access files by casting them as virtual memory segments. A file is actually accessed when an

address translation fault occurs and the operating system steps in. The operating system decides whether the excepted address is within the file, selects a suitable part of the file including this address, fetches it from the filing system, updates the address translation tables and restarts the interrupted process. There is no concept of the current position in a file (at the operating system level anyway). There is no requirement to extract bytes from the pages since each page is either completely resident in store or completely at the file server, assuming that the granularity of virtual memory mapping is a multiple of the network file server page size. The length of a file probably needs to be known when the segment is first mapped rather than upon the first attempt to access a page not in the file.

Local area networks have a very low error rate, low enough that a good strategy for handling errors is to retry the operation if a suitable reply is not received in a reasonable time. For this to be a workable recovery strategy it is necessary that operations are idempotent. If the current file position is held at the file server then this is not the case. If a request to read some data is received but the data is lost in transit, then a subsequent repeated request will cause new data to be sent and the first lot to be lost. It is therefore much more sensible to keep any file pointers resident in clients. Besides, as we have seen with mapped files, the concept of a current position within the file is not always useful and would mean that handling a virtual memory exception would incur an extra message exchange to set the position.

With no concept of an open file, as in WFS [SWI80], there is no file locking. Unintended conflicts of file access are rare but not unheard of and so this seems unacceptable. On the other hand, the requirement to open and close a file in order to extract all or part of it in one go seems unnecessary. A good compromise seems to be that adopted by the Cambridge file server [DIO80] where a file may be opened (and locked) or it may be accessed without being

opened (in which case the file server does an automatic implicit open and close).

Even if we have a connectionless protocol and files are not opened, some form of file creation primitive will be required. This may well interact with the file textnaming scheme since most operating systems interpret opening a file for writing as implying creating the name (and the file) should it not already exist. If there are different types of files then creating a file is significantly different from writing an existing file, even if all the old data is updated. Creating a file requires extra information about the type of the file being created.

The length of a file is clearly one of its most important attributes. A file has other attributes too, but these only exist to be examined and do not interact with the data transfer primitives; for example, the date of creation of a file does not affect the response to requests to read parts of it. The length of a file is a rather odd attribute because of its interaction with data transfer. Most files are written and read sequentially and so we would certainly like a file written to, to automatically inherit a length such that the same bytes will be delivered when it is reread.

There are two different ways in which the file length can be decided. Firstly, it can be a function of how the file was written - the high water mark of bytes written to the file for example. If files are only written in pages then an extra mechanism will be needed to cope with partly filled final pages. Secondly, the length can be set explicitly by the program writing the file. The best solution seems to be a combination of the two. The length of a file is the high water mark of all bytes written although it can be set by a command to truncate the file (discarding unwanted data) or to elongate the file (to create a large file without the bother of writing it all).

The most contentious issue in the selection of data transfer primitives is the actual unit of transfer. There are two very compelling, but conflicting, arguments.

General programming practice argues that if a file is an unstructured array of bytes, then the unit of transfer should be an arbitrary block of these bytes. A file is an abstraction, and one of the purposes of such abstractions is to hide its implementation. If we let implementation specific details through into the implementation, such as pages or cylinders, then we may force an unsuitable unit on clients or restrict ourselves should a later server wish to use hardware with different characteristics. This approach is taken by the Cambridge file server [DIO80].

On the other hand, our two criteria for efficiency and for maximum functional distribution press for the disc page, or perhaps sequence of disc pages, as the unit of transfer. If clients blindly choose an incompatible unit for transfer, then the transfer becomes inefficient. This argues for the page size to be visible to clients, but then it is only a short step to arguing that other details, such as disc addresses, could usefully be made visible. Furthermore, accessing arbitrary blocks of bytes forces extra work on the shared processor of the server in extracting them (when reading) or in reading the old page and injecting them (when writing). This last case is particularly important since many files will be written sequentially. We certainly want to avoid the server rereading the last page of a file on each request, to add some more bytes to the end. This approach is the one adopted by WFS [SWI79], although its bizarre page size of 492 bytes is unlikely to be convenient for client systems. If we opt for this scheme, we may need a way of writing a partial page to finish off a file. This is most easily dealt with by allowing a partial page to be written at any time, the rest of the page being filled with some standard byte (probably zero).

One further consideration is that the designers of client systems are likely to tune them to the network file server. Even if the page size is not pushed through into the command interface, client systems are likely to access files at page boundaries by choosing suitable in-store buffers.

The arguments for and against making the page size visible are sufficiently strong to make a choice very difficult. To some extent they depend on the ratio of processor power to communications bandwidth, since there is little point in shifting stuff across the network for the client to handle if the network is too slow. In the future, bandwidths of networks are likely to increase and the number of clients being serviced is likely to increase. The processor of the server is likely, therefore, to become a bottleneck and the second approach, where clients can only access single pages, or perhaps multiples of a page, seems more attractive. Nevertheless, it is a slightly uneasy decision.

## Standards and translation

One important purpose of a network filing system is to permit files to be shared. In particular, it must permit high level language source files to be shared amongst different systems on which they may be compiled and run. This requires either that all systems adhere to a common standard for such files, or else that the network filing system massages the file data for each different system or group of systems.

In fact, these are just two cases of the same thing. If the view of a file by the time it reaches the application program (or its run-time system) is different from the representation of the data at the file server then it has to be transformed somewhere. If this is performed at the client, then the interface at the

server is standard. Alternatively the data could be transformed at the server. This second option is very unattractive since such operations are likely to be processor intensive, and our criterion for distribution makes it desirable that this is done in the unshared client.

If the transformation takes place within the application program, within a database management program for example, then there is not really any problem. If the program itself is portable between different client systems, then the transformation should impose the same interpretation on the same raw data on both systems. The problems arise with files where the transformation is done either in the operating system or within a very pervasive run-time system.

A common standard is attractive but there is no certainty that pre-existing systems will be able to live with it. For example, systems which regard a file as a sequence of bytes usually have some conventional line separator character such as a linefeed. Other systems have the concept of a record more deeply defined and no explicit separator character but rather record lengths preceding each record. Under this regime it is possible for any character to occur in a record, in particular, whatever character might normally be regarded as a separator. It is thus an information losing operation to take a file of such records and replace all the record lengths with record separators; it is not necessarily possible to regenerate the original records accurately. This is most obviously a problem with files such as compiler object code, which are not required to be shared anyway. However, the problem frequently occurs with files destined for high-quality printers or plotters, which may be required to be shared between the system generating the file and the system controlling the appropriate peripheral. In any case, application programs do not necessarily distinguish such near-binary files from more universal ones.

A further consideration with a common standard is that a client system with a very different internal standard will have a very expensive interface. File formats will be translated both on being written to the filing system and on being read back, even though most files will not be accessed from elsewhere before they are altered again.

So it seems that the best that can be achieved is to have a standard for certain types of files, most importantly text files and files destined for peripherals, and accept that some client systems may be unable to live with it and that others may find it rather expensive. This is rather negative, but results from the greater weight placed on the uniformity of the network filing system (hence, how well files can be shared) rather than on efficiency of use by any particular client system.

## Conclusions

There are two separate areas where the data transfer primitives impact on the design and use of the network filing system. The first area concerns the primitives actually provided by the file servers for the transfer of data. The second area is the need for client systems using the network filing system to stick to standard data representations for data which is potentially shareable with other systems.

A file is just an vector of pages of vectors of bytes. The imposition of further structure on this raw data is the concern of client systems and programs. The considerations about whether or not to make the page divisions visible to clients seem to be fairly evenly balanced. Even if they are not made visible by the primitives used to access files, it is likely that client systems will be tuned by taking the actual page size into account when

relevant, such as when deciding on the size of buffers. This factor seems to tip the balance in favour of only making files accessible a page at a time, although very restricted client machines may find this problematic if the page size is large.

Text files should be shareable between the different systems attached to the network filing system. This requires the cooperation of client systems since the network filing system imposes no restriction on what raw data can be stored in files. This cooperation means that there has to be a canonical form of a text file. Only for the strongest of reasons should client systems store text files in internal formats unacceptable to other systems. When at all possible such files should be converted by client systems.

# Chapter 5: Naming

Files have to be named. That is, there has to be a scheme agreed between the clients and the network filing system whereby certain symbols appearing in messages are interpreted as files (or, strictly, references to files). If a file is not nameable by a client then it is not accessible by the client, so if files were not named then they would all be inaccessible to all clients. Note that we are not (yet) talking about conventional file textnames, but about a more general concept which covers textnames, file capabilites and internal file identifiers (such as the i-numbers of Unix).

The file server implements files in terms of names agreed with its disc controllers, physical disc addresses. One function of the filing system is to administrate the conversion from a filename and page within file pair, into a physical disc address. It does this by resolving the page within file number in a context, a different context for each file, giving the disc address as the resolution. Depending on what strategy the filing system has chosen for laying out files on the disc, the exact form of this context will vary.

Similarly, the name of a file is resolved in some context to get the administrative data for the file and so, ultimately, the data that it contains. However, unlike the disc addresses, this context is distributed amongst all the servers of the network filing system, which considerably complicates the issue.

The file server's requirements

Files have some internal name, such as the number of its file header or a directory and slot number pair; this is what gets returned by the lookup filename procedure of the file server. These internal names are used as the ultimate resolution of textnames in any textnaming scheme which may be implemented.

Note that these internal names may only have an agreed meaning between the textnaming scheme and the underlying file access scheme. Whether client systems are allowed to meaningfully use the internal names in commands is a different question. For example, internally both Unix and DEC files-11 name files by a header number (i-number or fid). On Unix, this number is only used between the textnaming scheme and the underlying file scheme - it is not possible to open a file by header number, but only by textname. On files-11, this number has meaning outside too; although resolving a textname is the usual way to discover the number, any other route is acceptable since ultimately files are only accessed by header number.

Names in any particular context have to be unique; for example, there is only one page corresponding to a given physical disc address on a particular disc. Our criteria for autonomy and distribution mean that file servers need to be able to generate names unique in the context of the whole filing system, without having the whole context available (to check the uniqueness) and without using a central agency issuing unique names. Instead of generating these names, they may be supplied by client systems, but the problems of checking uniqueness are essentially the same. We cannot have two files with the same name. There are two fundamentally different ways of achieving this.

The first way is to make the name depend on the server to which it is presented. Essentially, the global context is partitioned into a number of local contexts. Since names only have to be

unique within this local context, each file server can ensure uniqueness easily. Under this scheme, each server can have a file with the same name and, provided that there is never any need to communicate with one file server about a file which is at another, there is no conflict. Unfortunately, at some stage the client system has to decide which file server provides an appropriate context for resolution of the name, and a wrong choice may not provoke an error message but merely access the wrong file.

This is the situtation with very loosely connected (or even disconnected) systems. There is no confusion between files of the same name on all the Unix systems in existence, nor with the i-numbers of such files on all the different Unix disc volumes. This is because the contexts for resolution are carefully recognised and separated. As a result, it is not possible to name a file on one volume in a directory on another, let alone name a file on one system in a directory on another system.

The second way to ensure uniqueness of names is to generate them in such a way that they cannot be anything but unique. The easiest way is to arrange for each server to have a unique number and to use this number as part of the name. Allocating these numbers (and hence ensuring their uniqueness) requires a central agency, but since this only happens when a new file server is first added to the filing system, it is only a minor transgression of our design criteria. If the network addresses are eternally unique, as are those proposed for the commercial Ethernet [DEC80], then servers already have unique numbers. In this scheme, to resolve a name, a client merely presents the name to a file server. There is no confusion if the wrong server is selected, but there is no guarantee that any particular server will have the appropriate part of the global naming context accessible to perform the resolution.

These two schemes seem very similar, but they differ in a subtle way. The name of a file and its location are two different concepts. The first scheme couples them tightly since the name of

a file also carries its location. If a file should move then its name would change. The second scheme divorces the two concepts. True, a location has been used in the name, but only to ensure uniqueness. A file can move freely without changing its name. However, because the name of a file and its location are independent there is a problem about finding it quickly (it can always be found slowly by trying each file server in turn).

The second scheme is also more error tolerant. Because names are unique rather than unique within each server, presenting a name at the wrong server is not disastrous. It is certainly detectable, but, more importantly, the name is meaningful at that server since the context for its resolution (although not necessarily all the data needed to resolve it) exists there. The first scheme splits up the naming context itself; the second scheme merely partitions the data needed to implement the context, a more satisfactory solution.

It is clearly desirable to be able to have multiple copies of files, both to increase tolerance to file server failure and because file servers are not accessible uniformly fast from clients. This introduces further problems into the naming scheme.

Maintenance of multiple copies of writable files is an area of current research, especially for infrequently updated databases and for databases which are strongly partitioned with people in different places responsible for each partition. Problems arise when a file is updated and it becomes necessary to distribute the update. It is not really acceptable to lock all copies of the database from starting an update to its being distributed, since they may not all be accessible, and most updates can wait for a massed distribution without causing any problem. However, if this is not done, two updates done around the same time may clash and some technique will be needed to resolve them. Resolution techniques of this kind are clearly outside of the filing system used to hold the database. Whilst clearly important, we shall not

consider multiple copies of such files further. If multiple copies are needed, then each copy will need a separate name and the responsibility for consistency of the different copies rests with some other agency (such as the database management program).

At first sight, one class of writeable files for which we would like the ability to distribute multiple copies has less stringent consistency problems. That is those files about which we are concerned that the update gets done eventually, but not particularly worried about precisely when. For example, the executable code of a compiler is rarely updated. Furthermore, when it is updated, we are not especially concerned that all copies are simultaneously updated but rather that the change propagates through the system acceptably fast. However, more often, updating the executable code of a compiler implies a new release. Both the old and the new release are often available in tandem for a time. Thus updating a compiler is not a true update of the executable code file, but a change in the release (and hence the file) used by default.

Releases of compilers are thus never updated. They are one example of a large number of read-only files. These are unalterable (although there may need to be a scheme to get rid of them eventually). Since they are unalterable, there are no problems about consistency between the various copies. Since they are identical, it seems appropriate for them to have the same internal filename. They form a particularly important class of files, since they can be cached on local discs at clients without complex locking problems. This will be discussed in detail later.

## The user's requirements

Users like to choose their own textnames for files since names chosen by the network filing system are unlikely to prove memorable. These textnames have to be unique within the context in which they are resolved and so either users have to agree a scheme among themselves to keep the textnames unique, or else multiple contexts for resolution are necessary. We will call the contexts for the resolution of textnames directories. They will often be implemented as files (with security to prevent unrestricted scribbling on them).

The minimum workable scheme for having multiple contexts is to have one context per user [SAL77]. If users are to share files, then a user needs to be able to specify that a particular textname is to be resolved in another user's directory. Traditionally, this is done by giving each directory a textname (unique amongst all directories) and allowing file textnames to be suitably prefixed with directory textnames. Thus "john.abc" is the file "abc" in john's directory. This scheme is used in all the early file servers.

Many users amass so many files that the restriction that their files must have unique names amongst all their files is too severe. Further levels can be added to the naming so that each textname has three (or more) components. Now textnames have the form "john.pascal.abc" and it is only necessary that "abc" be a unique name amongst all the files with names starting "john.pascal"; there could be a "john.algol.abc", for example.

The restriction to a fixed number of components soon becomes restrictive. If it is large enough to cope with the worst requirements of users, then it is tedious for users with few files. So, the number of components is made variable, giving a full pathname. Now "john.pascal.abc" and "john.temp" can both coexist. Depending on whether directories are made visible as files,

"john.pascal" is either a file or else is inaccessible (except by special primitives). Since users often produce inventories of all the files that they possess, it simplifies things if "john.pascal" is a file which can be read, for otherwise special primitives will need to be implemented for production of inventories. In any case, "john.pascal" is the textname of a context suitable for the resolution of "abc".

The first component, "john" above, is resolved in a distinguished global context. In a coresident filing system this is not problematic since there can easily be a unique master directory holding this context. In the case of a network filing system we want to avoid centralised resources such as single master directories. There is already a master directory of sorts, implemented by outside agencies such as name servers, which resolves textnames for services into network addresses. These change infrequently enough that the consistency problems caused by duplicating it are far less than the reliability and performance problems caused by centralising it. So we use the server names as the first components of pathnames ("server1.john.pascal.abc"). Note that using the server name in this way does not constrain where the file itself resides, but merely defines a starting context for resolution of the textname. It is quite possible that "server1.john.pascal" and "server2.john.pascal" are the same file.

It is clearly tedious to have to type complete pathnames all the time, and so there is usually a way to omit some of the early part of the pathname and have it understood by some means. For example, most systems have a concept of a default working directory which is added to the start of incomplete pathnames. In effect, the working directory defines a context in which the textname can be resolved. We shall write incomplete pathnames by starting them with a period, ".abc". In practice, when pathnames are often typed at terminals, it is more convenient to use the opposite convention and only prefix full pathnames with a period.

Users do not normally work with files themselves, but use files (and parts of files) as representations for objects in which they are interested, such as programs, integrated circuit designs or telephone directories. This is because files are generally the only long-term storage available to them. These objects often have names themselves and it is convenient if the textnaming scheme of the filing system is versatile enough that it is not necessary to have separate schemes for naming the objects and the files in which they are represented. Otherwise, separate contexts will be needed, outside of the filing system textname scheme, for resolving the name of objects to get their containing file. For example, many command interpreters find it convenient to use file textname directories as dictionaries for command words, rather than constructing a separate scheme to turn a command word into the name of a file containing the program which implements it.

There are clear advantages to the user if the textnaming scheme is sufficiently versatile that additional textnaming schemes are unnecessary. This saves both the effort needed to understand another scheme, and the effort involved in writing the code to implement it. In a distributed environment this can be particularly important since the additional naming scheme may need to run on a number of different types of computer. It is clearly undesirable if the accessibility of objects such as telephone directories should depend on having an appropriate compiler for the language in which the naming scheme is written. This could be a particular problem when an application is itself distributed across a number of different types of client.

The problem is further complicated by the fact that objects can contain references to other objects. For example, a number of programs use the same external module, a number of integrated circuit designs use the same standard cells or a telephone number program uses a number of telephone directories. If the textnaming scheme is versatile enough that it can be used for these references, then each of these examples reduces to a file

containing the textnames of other files.

To see that problems can occur with such a scheme, consider a telephone number program which looks up telephone numbers in telephone directories. To avoid confusion with file directories, we will call these telephone lists. A sensible approach would be to represent a telephone list using a file. We would like the program to be sufficiently general that it can work with any suitably formatted telephone list. This means that the program cannot have embedded within it the full pathname of the telephone list, such as "eu.phones.numbers", since it will then always use a fixed directory. On the other hand, we do not want to have to type (nor even know) the textnames of the common telephone lists. On a system with a coresident filing system, we may be able to get away with a scheme where a default telephone list specified by complete pathname can be overridden by the user running the program. In a distributed environment this does not work. The default telephone list in London and Edinburgh are unlikely to be the same.

If the program does not contain a full pathname but a partial pathname, ".numbers", then there is confusion about what context should be used for resolution. There are two obvious contexts, either of which may be appropriate. The first is the working directory of the user running the program. In this case, the user's directory is expected to contain the file "numbers". It is clearly inconvenient for every user to have to set the working directory appropriately before running the program. The second context is that of the directory in which the telephone program was found. Now the program and the directory form a closure (as in the Lambda calculus [CHU41]), that is an object containing names together with a context in which the names are to be resolved to objects. The directory containing the program (strictly, the last component of the textname of the program) should also contain the component "numbers". The way in which the program is invoked thus determines which telephone list is used. If a user wishes to use a personal list, then the program and the list must be put into the

same directory. Note that both of these solutions use the filing system textnaming to perform the resolution, and both require multiple names for files if the filing system is not to fill up with a proliferation of identical telephone lists and programs with different textnames.

Another way to solve the problem is to construct a completely new naming scheme in addition to file textnaming. Programs are then written using these special names, such as "stream1" or "sysin". Before running a program these special names are initialised and bound to the real pathnames to be used. When a program attempts to access a file "sysin", "sysin" is resolved in the special scheme to get the pathname to be used. Unfortunately, if files can be accessed using either these special names or real pathnames there is a lot of confusion. Is "sysin" a pathname or a special name? If an application is split over more than one client machine do they share a global "sysin" or each have their own? Such a scheme looks unattractive in a network environment, unless it is actually implemented within the textnaming scheme of the filing system. For example, in addition to a current working directory, each user could have an additional directory holding these special names.

The problem is more complex with references between program modules. Here there are conflicts between the intentions of the author of the program and intentions of the person running it. The author of the program may split the program into modules, perhaps because some of the modules are used in several different programs. For example, Jack creates a program which uses another module of his, called "common". When Jill runs the program the textname "common" has to be resolved to find the code of the module. This has to be resolved in Jack's context, not Jill's, since Jill should not need to know about "common" and may even have a module of her own using the same name. One way of resolving the problem is to insist that Jack resolves all the references to code before letting Jill use the program. Jack now has to pass the program through a

special program, usually called a linker, resolving all the references and removing all the textnames. Apart from the undesirable implications on space if "common" occurs in many programs that Jack makes generally available, if Jack updates "common" to correct a bug then all the programs using it have to be relinked or they will remain oblivious of the change. Including the name of an object in a program is not the same as including the object itself, unless the object can never change.

Note that the problems which arise about naming, such as those of resolving outbound references just mentioned, are in the province of the client operating systems and not of the network filing system. However, the way in which clients may choose to solve the problems have implications for any textnaming scheme which the network filing system may implement. If the textnaming scheme is not suitably general then it may not even be sufficient for a client system to use for textnaming files. For example, an increasingly popular thing to do is to build systems which look like Unix to the user at a terminal. To construct a Unix-like client system using the network filing system is certainly a possibility. One feature of Unix is that files may have multiple textnames (links). If the textnaming scheme of the network filing system did not allow this then it would be necessary either to omit this feature from the implementation or to implement a separate textnaming scheme outside of that provided. Neither is desirable. For similar reasons, a restriction (actually imposed in Unix) that directories may not be multiply named may be unacceptable to some client systems.

Almost all systems give textnames to file-like objects, such as peripheral devices or communication channels [RAS80]. This is because it is convenient to have an abstraction of an i/o device so that application programs can use the same procedures, independently, for example, of whether output is to a file or to a printer. Sometimes the textnames are recognised as devices before resolution, and sometimes the name is resolved and it is the

resolution which is recognised as a device. The second scheme is superior since it means that all the facilities of the file textnaming scheme can be used to handle the names of devices. In a network filing system, the distinction is as to whether the device textnames appear in the filing system textnaming scheme at all. If they do not, then the names of devices either need to be built into client systems, or else kept in a separate naming scheme. Because this is undesirable, it therefore seems essential that the network textnaming scheme be able to name file-like objects. In turn this means that files must have attributes which can be used by client systems, even if only to distinguish file-like objects from genuine files. There is no guarantee that all different client systems will recognise these special files as such; a file which may be a wire-wrap machine to one client may just be an empty file to another. This is unlikely to be a problem any more than a file of executable code to one client being just a meaningless string of bits to another.

The scheme for preserving files in directories so far described uses early binding. Given the textname of a file and a directory, the filename is resolved to an internal filename and preserved in the directory. The component of the pathname has been bound at this early point, rather than later when the component is resolved. The alternative, late binding, requires the full pathname itself to be preserved as the resolution of the component. This scheme was used in CAL [LAM76] where it was called a soft link (the early bound component was called a hard link). There are intermediate schemes, although it is hard to see uses for them, where some of the resolution (say all except the last component) is done to provide a textname and the internal name of a directory in which to resolve it.

Requirements of the naming scheme

The naming scheme breaks into two parts. Firstly the internal naming scheme by which a global identifier for the file is used to access the file. And secondly, the scheme by which textnames are resolved to give internal identifiers. The first scheme certainly has to be implemented within the file servers. The second scheme does not.

We have already seen that the differing requirements of different client systems mean that the textnaming scheme of a network filing system cannot afford to have unnecessary restrictions, for otherwise some client systems may be forced to implement their own textnaming scheme. This, of course, conflicts with our criterion that all files should be uniformly accessible from all clients.

Naturally, it is possible to dream up suitably baroque requirements for two different client systems such that no textnaming scheme could satisfy them all. However, the textnaming schemes of all realistic filing systems have so much in common that it does seem to be possible to satisfy them all.

The alternative approach, as used in WFS [SWI80] or the Cambridge file server [DIO80], is not to implement any textnaming scheme at all but leave it up to each client system. Although this gives the authors of client systems complete freedom to handle textnaming as they see fit, it means that files are not easily shared between filing systems. Certainly, presenting one filing system with the textname of a file from another is meaningless. The only way to share files is to drop down to the lower level and use the internal name. Since this is likely to be something suitable for computers, such as a 64 bit number, it is unlikely to be convenient for users.

The construction of a textnaming scheme is thus a compromise. On the one hand we do not want to cramp client systems' style; on the other we want all files to be accessible from all client systems. There seems to be no point in constructing a textnaming scheme for use by client systems unless it is going to satisfy their demands, for otherwise they will implement their own. Thus the two options seem to be to implement no textnaming scheme (but provide hooks for client systems to do so) or to implement a comprehensive naming scheme.

The requirements are really quite simple. Most of the complexity occurs in some aspects of the implementation, especially in handling pathnames which cross from one server to another.

Directories are just files which bind character strings (components of pathnames) to other filenames, either internal filenames or other pathnames. Directories could be special types of file inaccessible in the usual way. With foresight, this would make the next criterion difficult to satisfy. There should be as few restrictions as possible on these character strings. It seems reasonable to restrict them to being strings of printable characters but not to restrict them to 6 characters, nor to insist that they must not contain dollar signs.

There is no restriction on which files can be bound to which character strings. In particular, files can have multiple textnames. This also applies to directories; directories may have multiple textnames. Further, files may be bound into directories on servers other than the one holding them.

There is a global context, probably outside of the network filing system proper, in which the first component of a fully specified pathname can be resolved to the internal filename of a directory.

The requirements for the internal naming scheme are still simpler, although there are implementation problems associated with locating files and, as with textnaming, deciding when a file can be destroyed.

Every file should have an eternally unique internal name. Once a file has been created with an internal name then no file will ever again anywhere be created with the same name. If internal names are to be used by clients, it is sensible to draw them from a sufficiently sparse space that the probability of a faulty client system producing a valid name is effectively zero. This is most easily done by extending the meaningful part of the internal name with extra random bits. Unless security is also controlled by the internal name, making it a sort of capability for the file, then this is just a defensive measure. Accidentally conjuring up a valid internal name and then having access to the file is even more unlikely.

Note that the two levels of the naming scheme are conceptionally independent to the extent that the textnaming scheme could be implemented in a separate server. The textname servers would all be identical and so there is no requirement that they correspond one-one with the file servers. If textnaming proves a bottleneck then we can have more textname servers than file servers proper. Alternatively, a local network may only need one textname server and several file servers. The textnaming could be implemented within the file server itself. We could also have a mixture, where some file servers and textname servers coreside and some are functionally distributed.

Most of the textnaming can even be moved to client machines. There is no problem about aspects of the textnaming scheme which only involve read access to directories, such as resolution of textnames. If write access to directories is also to be moved, then there is scope for trouble. The network filing system cannot trust the client systems to write directories correctly. However,

users have a strong incentive to ensure that they only use client systems which do maintain directories properly since their files will otherwise become inaccessible (if not destroyed). If access to directories is controlled by a proper security scheme, rather than by a self-imposed regime of the textnaming server, then a client system will not be able to corrupt directories of other than its own users. This ignores Trojan horse problems of client systems deliberately saving up passwords which pass through them. A crash of a client system while updating a directory could also cause problems; proper control of consistency will remove this as a potential source of trouble.

To implement the naming scheme, it is necessary to increase the locality and independence of the file servers without breaking down the illusion of a single coherent scheme. This inevitably leads to some compromises.

## Location of files

Having resolved the textname to an internal name for a file, we have to locate the file. Normally it will reside at the same server as the internal name was discovered, but for a binding across servers this will not be the case.

The key decision is whether or not the internal filename contains sufficient information to locate the holding server, and, if not, from where this information is to come. Note that any procedure concerned with the location of files is simply a way of increasing efficiency. The actual location of the file can certainly be found by a request to the server which holds it, and so searching all servers will always succeed (unless the file does not really exist). Procedures for locating files are merely conventions which speed up location by restricting the servers

which need to be considered. Because they do not directly affect the functionality of the network filing system it is appropriate to use heuristic techniques to perform location. It is in the nature of heuristics that they may occasionally fail, and then a full search may need to be invoked.

If the internal filename contains the server in some way, then deriving the location of the file from its name is easy. Unfortunately, we lose two desirable features. Firstly, we cannot move a file between servers without its name changing. Secondly, we cannot have multiple copies of a file without each copy having a different name.

The Cambridge file server [DIO80] goes further than this and has the internal name of a file include the disc address of its header. This is even more restrictive since the file cannot even be moved around within the server, to compact a disc for example. It is also difficult to see how an archive and recovery scheme could work since recovering a file from archive requires a particular disc page to be free.

If the internal name does not include the server holding the file then the file will need to be searched for. In a suitably rich network environment this search could include many widely scattered servers. An unrestricted search is unacceptable, especially in the case where the internal filename has been incorrectly quoted and does not, in fact, exist. The only acceptable alternatives are to restrict the search to a tractable size or to construct another method for discovering the location of a file.

Most files will not move from one server to another, so it is sensible to include the site at which the file was created in the internal name. This may, in any case, be necessary to ensure that internal names are unique in space. This still does not cover files which move between servers nor files which are read-only and

multiply incarnated, but it covers all but a tiny percentage of files.

It is difficult to construct alternative methods for deciding the location of a file, without requiring a central file location atlas. One solution to the files moving [LAM81] is to have the creating server maintain a table of all files which have moved on elsewhere. Occasionally, every server reports all the imported files that it holds to their creators, to ensure that these tables are kept up to date. If very few files move, then this mechanism is cumbersome. Nevertheless, it can cope with a wide range of movement, and cope with it automatically.

Another way of performing the location is to leave it all to the client. When a file is not found in any expected place, then the client searches for it. Users can guide this search since they may have a good idea of where the file is to be found. Once located, the client puts the couplet of internal filename and holding server into a lookaside exception list, a sort of cache of files which are in unexpected places. The search can then be bypassed if the file is accessed again sufficiently quickly.

The alternative approach is to restrict file movement. We have already considered the ultimate restriction that files may not move at all. For example, a sensible restriction might be that a file could only move between its creating server and the archive server covering it. This restriction would be too severe in some circumstances.

Note that both approaches (an alternative location scheme and restricted movement) only attempt to deal with files which move. Files which remain at their creating server are found directly. Although most files are at their creating server, files are not uniformly accessed. Many of the files actually accessed will be copies of read-only files such as the executable code of compilers. How they are handled depends, to an extent, on how they are created

since this can affect their naming.

Read-only files will need a special instruction to create them from a file which is writeable, the clone file. This instruction can take one of three forms. Firstly, a writeable clone file can be declared to be henceforth read-only. Secondly, a write-once file can be created, which becomes read-only after it has been initialised. Thirdly, an instruction can request a new read-only copy be created of a given clone file which, itself, remains writable. The second and third approaches seem preferable. This is because it will almost certainly be convenient for read-only file to be identifiable as such from their internal name. This is especially important if the internal name contains the identity of the creating server, for this is fairly meaningless for a read-only file which may exist at every server.

On resolving a pathname to the internal name of a read-only file, the strategy to locate the file will need to be different. Since the internal name carries little hint as to a likely server, it is sensible that read-only files are kept at the closest server to any client likely to use them. This is sensible, in any case, from a performance point of view. A read-only file is thus always expected to be found at a nearby server. One particularly important place that a read-only file might be found is on a client machine's local disc. This will be considered in more detail later.

## Lost files

When a file is created, a file server issues an eternally unique internal name for the file. To access the file for any purpose, this internal name is presented to the file server. One such purpose might be to delete the file. The internal name is invalidated and the space used to hold the file can be recovered and reissued (if the storage medium is multiply writable).

If the internal name is mislaid, then the file becomes unnameable. Because no client can name the file, it cannot be deleted. The file server cannot tell the difference between an internal name lost (for all time) and an internal name which has not been used for some time. It cannot, therefore, autonomously delete the file, even though no client would be able to detect that this has been done. We will call such a file a "lost" file.

As with location of files, this section is concerned with efficiency considerations, rather than functionality. It is always safe to preserve a lost file. It is desirable to be able to reuse the pages allocated to a file should it become lost, and even a write-once medium is likely to benefit from keeping its tables free of the internal names of lost files.

Almost by definition, it is not possible to enumerate the lost files. The only way, therefore, to detect lost files is to enumerate all those files which are not lost, then enumerate all the files (both lost and otherwise); the difference is the lost files. It is safe to delete these files since it will be undetectable (there is no internal name with which to perform the detection). This is just the same as the garbage collection approach used in the implementation of applicative languages. Unfortunately, it is not directly possible to enumerate all the files which are not lost since they may be preserved in very inaccessible places such as sheets of paper or burnt into memory in client machines. It is therefore necessary to use a narrower

definition of lost. A file is lost if its internal filename is not preserved in a place accessible to the network filing system.

The most common approach to doing this is to bring the textnaming scheme further into the filing system proper, and then to insist that every file has at least one textname. The explicit delete command is then done away with, and a remove textname command used instead. When the last textname is unbound, the file is unnameable (by textname) and hence unnameable by internal name (an assumption); so it can be recycled.

Unfortunately, detecting the removal of the last textname is insufficient on its own. Two directories can contain textnames for each other with no further directory containing a textname for either. Although the last name of neither directory file has been removed, both are lost since there is no full pathname to them (nor, perhaps, to other files for which they contain textnames). There are two ways to handle this. Either to ensure that such situations cannot occur, or to have a program which scans the textname graph and disposes of such cases.

The simplest way to arrange that lost files cannot occur is to insist that the textnaming graph is acyclic. If it is, then a reference count scheme will suffice to detect unnameable files. This is most easily achieved by barring more than one textname for a given internal name, or barring directories from having more than one textname. This second solution is that used in Unix, for example, although a particular form of cycle is allowed since every directory contains a link to its parent. Under either of these conditions, the textname graph will be an acyclic tree and a simple reference count scheme will suffice.

There are strong arguments [SAL77] for not making arbitrary restrictions like this since the textnaming may become too restricted for some purpose that we might use it for, necessitating further special purpose textnaming schemes within files.

Furthermore, if bindings are permitted to be made by giving the textname component, the internal name of the file, and the internal name of the directory, then it will be very difficult to detect an attempt to create a cyclic structure. If textnaming is handled in some client systems, they may not bother even to try to detect it. A true garbage collection may need to be done to destroy all lost files. Without the acyclic guarantee, it is possible to have a set of files with non-zero reference counts which are not reachable from the top-level directories in the filing system. A necessary condition for garbage collection to be required is the removal of a binding to a directory with a non-zero reference count. This garbage collection turns out to be less complex than it might appear and can be done without taking the filing system out of service [BIR78,GAR80].

An alternative to garbage collection is to distinguish one textname for each file, the principal name [LAM76]. If the file is deleted using this name then it is deleted; otherwise the name is merely unbound. If any file accounting is done, either for money or to encourage frugality, then this scheme works quite well since the owner of the principal name pays for the storage. Most of the other schemes interface poorly with any form of accounting for storage costs.

If an approach is taken where the textnaming of files is outside of the filing system, then lost files are a worse problem. Garbage collection consists of marking all the reachable files and deleting the rest. The first stage is still easy enough, but since the unreachable files may belong to another filing system they cannot be deleted.

For example, in WFS [SWI79] files are identified by a unique integer which must be presented to delete the file. Since there is no other mechanism for deleting files, losing this integer precludes ever being able to delete the file. This was originally not considered a problem but eventually this lax approach had to be

tightened up. Any files not held in the standard naming system implemented on top of WFS might be deleted [LAM81].

A compromise solution is that adopted by the Cambridge file-server [DIO80,BIR79]. Here, the textnaming is the responsibility of clients but the file server maintains a vestigial copy of the naming graph. A file is lost if it is lost from this naming graph, even if the internal name exists elsewhere. This does not completely solve the problem since a client server may consider a directory to be empty while the file server considers it to contain files. These files are reachable from the point of view of the file server but not from the point of view of the client filing system which will never attempt to reach them. Although this is a single server filing system, the garbage collection is done asynchronously (but occasionally) by another machine on the network [GAR80].

In a network filing system this garbage collection is potentially far worse. We have already decided that it is desirable to allow textnames on one server to be bound to internal names for files at another. Any garbage collection thus needs to cover all servers, which may well be an intractable task. It should even involve storage volumes which are offline.

Even if this garbage collection could be correctly performed, the unrestricted binding of textnames to files at other servers may be undesirable. A large file may be preserved at one server merely because someone has a remote binding about which they have forgotten; at best, the file should be shipped off to their local server but the binding is probably unnecessary.

The best way to restrict the binding is to have a graph of storage volume relationships. We say that a storage volume A "influences" a storage volume B if a textname on A bound to an internal name on B is sufficient to guarantee preservation of the file. Note that two volumes may influence each other. Such

volumes are called "coupled" and a cycle can occur in the textnaming graph across such volumes.

Now the way to do the garbage collection is clear. Since a cyclic structure can only occur on a single volume or in a cycle of volumes each of which is coupled to the next, we must garbage collect a coupled set of volumes. There is nothing to be gained by restricting the naming so that coupled is not a transitive relation. If volume A is coupled to B and volume B to C, then we gain nothing by not coupling A to C.

First we scan all the coupled volumes in the set and note all reachable files. Next we scan all the influential volumes for inbound bindings to the storage volumes being garbage collected. Finally, any files remaining unreachable are garbage and can be destroyed.

Offline volumes used for archive purposes will have no coupled volumes (which would imply non-archive files being named from archived directories) but will be influenced by some online volumes containing the archive directories. Unbinding the name from one of these archive directories will result in the file being deleted from the archive when it is next garbage collected.

If any volume is offline (or its holding server is not running) when a remote binding is removed then there is no problem. The file may be preserved unnecessarily until the next garbage collection but it certainly will not be deleted in error. There is, however, a problem when creating a binding if the volume holding the file management data is inaccessible. If the textname reference count is not updated then the file may be deleted in error whilst it still has outstanding textnames. It is therefore necessary to prevent creating such a binding if the reference count of the file being bound is inaccessible.

Multiple servers on the same local network will be coupled and will have unrestricted binding amongst themselves. Binding to a file on a distant server, while not disallowed, is not sufficient in itself to preserve the file. Note that we cannot have the desirable setup where each server is coupled to those near it. Because coupled is effectively transitive, this means that all the servers would need to be garbage collected together, the task we are trying to avoid.

If the server uses write-once optical discs, then the garbage collection will take place when a disc is nearly full and needs to be condensed to a new disc to gain more unwritten working space. This happens rarely and is the only time that information can be lost from such a filing system so there is no need to maintain reference counts. The concept of influential storage volumes is still needed to limit the size of the garbage collection, especially because optical discs are so large.

A workable textnaming scheme

Files are identified by internal filenames, which are actually large integers. Given such an internal filename it is possible to extract the identity of the server which created the file, and whether or not the file is a read-only file which (potentially) exists on many servers in the network or on local caches held by clients. No other explicit method of location of files than this exists, although it may be sensible to search in a few likely places if the creating server denies knowledge of a desired file. Influenced volumes also provide useful hints for likely servers to hold the file.

There is a textnaming scheme as follows. Each server maintains a particular distinguished directory, the root directory, a separate root directory for each server. An external textnaming scheme for servers exists so that the first component of a full pathname can be resolved to the root directory on a particular file server. In practice, this would be implemented by a name server or by having the names of servers well-known, in the sense that any client could convert a server textname into a network address and the internal filename of its distinguisehd directory. Resolution of a textname normally starts in the global context of the textnames of the servers comprising the filing systems. In some cases, resolution will be required to start in a specific directory and this is achieved by specifying the internal name of the directory file.

Each entry in a directory binds a component of a textname to one of three things.

The first thing to which a component can be bound in a directory is the internal name of another directory. This is normal for early components of a long pathname. The component of the pathname just resolved is removed and the rest of the name resolved starting from the directory found in the resolution.

The second thing to which a component can be bound is the internal name of an ordinary file. Any file other than a directory is ordinary in this context. Only the last component of a textname will normally resolve to an ordinary file since such a file does not directly provide a context for further resolution. Since files have attributes, an empty file together with its attributes can be used as a representation for other objects. For example, one attribute might be the network address of a textnamed network service.

The third thing is another pathname. This is the soft link discussed earlier. The pathname found is inserted as a prefix onto the remainder of the pathname being resolved to give a new pathname. This pathname is resolved starting back in the global context.

Some examples are in order. Consider resolving the textname "server12.paul.abc". The first component, "server12" specifies the server whose root directory is to be used to resolve "paul". This could, in turn, resolve to a directory in which there will be a component "abc" which is bound to an ordinary file. Alternatively, "server12.paul" could be bound to the soft link "server13.peter" which restarts the resolution back in the global context with the pathname "server13.peter.abc".

Finally, consider the resolution of "server12.magtape.abc". The first resolution, of "server12.magtape", could be an ordinary file. Since this is not a context for further resolution, the internal name resolved together with the remaining part of the pathname, "abc", are passed back to the client. The attributes of file with that internal name specify a network address. This is passed the remaining pathname and checks that the volume label of the mounted tape is "abc". The magnetic tape server then returns some handle by which the mounted tape can be accessed, the final resolution of the textname.

The storage volumes at some servers influence those at others. This influence is known at both servers but not necessarily elsewhere. When a new binding is made to a file on another server (directory or ordinary), and the server holding the binding has influence over the server holding the file, then a reference count for the file is incremented. The server holding the file must be accessible to increment this count at the time that the binding is made. When a binding is removed the reference count is decremented if possible. The other server does not need to be accessible at this time. This asymmetry arises since it does not compromise the

correctness of the filing system if a file is unnecessarily preserved, since this turns out to mean preserved when noone can detect that it is being preserved. On the other hand, destroying a file early is detectable; again, almost by definition. Occasionally, a garbage collection is done to destroy unnameable lost files, as described in more detail earlier.

This occasional garbage collection also enforces the uniformity of the textnaming scheme, even if it is implemented in client operating systems. In this case, the garbage collector has to be tolerant of finding rubbish in files which purport to be directories. Directories just bind textname components to the internal filenames for files (binding to a pathname does not affect the garbage collection). If the directory is corrupted then this is either obvious, or the internal names are those of non-existent files, or they are internal names of the wrong files. All of these are safe, in the sense that it is not possible for a client system to overwrite a directory in a way which causes files from other directories to be destroyed in error.

Most of the compromises in this scheme arise out of the fact that a global view of something as large as a distributed filing system is intractably expensive. It is not possible to garbage collect a filing system of forty servers spanning as many networks. It is not even sensible to search them all to find a file without being expressly requested to do so. This means that in minor ways we have had to forgo the criterion that the location of a file is transparent. For example, the concept of influential volumes conflicts directly with this.

Summary

The naming scheme developed is much more general than those of most filing systems. It starts from the full requirements of the naming scheme and only restricts these when necessary to make implementation possible.

Most other systems restrict the context of the internal name of a file to the volume on which its textname is preserved (and so, the volume on which it is found). This goes against our desire that the facilites and guarantees about a file do not depend on its location. Even if most systems did otherwise, their control of consistency in the face of crashes is so poor that naming across volumes would be very difficult to set up reliably.

There are three main compromises necessary to make the naming implementable. Firstly, unrestricted movement of files from server to server is not handled well. Secondly, a file is only preserved if its internal name is bound to a reachable textname. And thirdly, the concept of reachable in the last compromise is further restricted by only considering certain storage volumes to be reached from a given volume.

All of these compromises seek to restrict the global view of the naming scheme as little as possible, whilst increasing the autonomy of each file server so that the illusion of the global view is implemented out of local concepts. Whether the rather intuitive restrictions will prove to be problematic in practice is something which cannot be discovered other than empirically.

# Chapter 6: Security

Most filing systems require some sort of security scheme since not all users wish their files to be uniformly accessible to all other users of the network filing system. Even in a very open environment where noone is worried about secrecy, it is a sensible defensive approach not to allow arbitrary users to (accidentally or deliberately) overwrite existing files.

Security interacts directly with sharing and naming. If there is no way for a user to name the files of another user then we have complete security but files are unshareable. If all files are shared and nameable by other users then there is no security. The problem is how to restrict access to files so that being able to name a file is not a sufficient condition to access it.

## Levels of security

Security is never absolute. The aim of a security scheme of any kind is to make access to the secured data so hard that other approaches to compromising its information content are more attractive to an intending rogue. In turn, this means that we want to make the system secure against all believable methods of attack.

This means that the level of security may differ according to how important the data to be protected is seen to be. The defence department of a country probably processes information by computer which they feel can justify any expense to protect. This means that computer rooms need to be in metal cages to prevent stray radiation from disc drives being picked up, stringent restrictions on who can enter buildings housing computers and terminals and so on. On the other hand, a university department may wish to keep

pending exam questions in a computer filing system. Picking up stray radiation from the computer room, or tapping terminal lines are hardly believable methods of attack, for they are probably more difficult than non-computer approaches such as breaking into offices. Doing the work required to answer the questions is probably the easiest method of attack anyway.

Local networks raise a special problem as far as intercepting data is concerned. Almost all local networks are inherently broadcast. A message is sent to all stations on the network. Hopefully one station recognises the destination address and passes the message to its host. With minor hardware alteration, and possibly even by software command, a station can be configured to receive all messages on the network (this is a useful feature for a monitor station). A station configured in such a way is clearly a security risk.

The only solution to this eavesdropping problem is to encrypt all messages. There are a number of ways of doing this, depending on the encryption scheme being used [NEE78]. Most encryption schemes rely on a key of some sort, for example the data encryption standard of the US national bureau of standards [NBS77]. This key has to be agreed between both parties to the encryption and so it needs to be inconveniently transmitted by another route (the network being insecure). The most attractive scheme is one of the public key cryptosystems [DIF76]. These avoid the problem of key transmission since the keys used to encrypt and decrypt are different. Depending on whether the encryption is being used for unforgeably signing a message or for concealing its contents, only the encryption or decryption key respectively needs to be secret; the other is published for intending communicators. An elegant scheme using two such key pairs can be used to conceal the contents of an unforgeable signed message. Public key cryptosystems are unfortunately computationally very expensive, hence very slow without hardware support. A single chip implementation of the algorithms [RIV80] can currently encrypt (or decrypt) about 1200

bits per second. Hardware using hundreds of chips can do better, but is too expensive to consider adding to every network station. It is worth noting that a public key system can be used to transmit the key for use with a less computationally demanding scheme to achieve higher bandwidth. Nevertheless, current encryption rates are several orders of magnitude less than the data rates typical of local area networks.

From now on we will assume that the network is secure. We will also assume that the file servers are secure; that is, that a rogue cannot run an arbitrary program in a file server machine and hence access the long term storage directly. We will also ignore such possible breaches of security as collecting radio interference from computers or the network, of tapping phone lines used as part of a long haul network and so on. In fact, we will restrict consideration to the case of a rogue user running a malevolent program in a client machine. The operating system of the client machine is not trusted though.

## Traditional approaches

Traditionally, complete security schemes are not implemented and it is doubtful whether their usefulness would justify their complexity. A complete security scheme would need the rights to access a file to be transferable, that any particular right of access already granted could be revoked (together with any rights passed on to others) and so on. A more pragmatic approach has resulted in more restricted schemes. Access to a file can be permitted to individual users, to the general public, to people who know a secret password, to programs marked in a special way and so on. These rights of access are not transferable in any generalised way and so can be revoked comparatively easily.

With a coresident filing system, the most common way to implement security is for the operating system to supply information about the identity of users accessing the filing system and about the use to which the data will be put. In a distributed filing system, performing this identification is much more complex. If the network has a comprehensive authorisation scheme, or if the network filing system implements one of its own by insisting that users go through some logon ritual involving passwords, the identification of people is not too hard. Identification of programs being run by people other than its author is still tricky.

For example, many operating systems allow a file of executable code to be protected so that it may be executed but not read for any other purpose. Effectively, access to the file is restricted to the code loader program, and some facilities for examining memory locations are inhibited. There are two problems about this in a distributed environment. The first concerns how to distinguish requests from the loader program from requests from other programs intent on stealing the code. The second is that it is hardly worthwhile anyhow, since having correctly loaded the code a user can dump the store contents of the client machine and unpick the code at leisure.

The case where a loaded program writes a file, rather than reads it in some way, is not so amenable to attack by dumping the store of the client. Consider a mail program which updates files of messages only accessible to the mail program. The only parts of the file which ever appear in the store of the client machine are those to which the user has access anyway. However, the mail program has to be self-identifying since there is no trusted third party to perform the identification. Whatever the means of identification, whether an encryption key or a password, it has to be contained within the code of the mail program, even if it is contained in a complex way and for only a short period. In any case, at some point the mail program will have to transmit the identification across the network to a file server. Although we

are considering the network to be secure, we are not considering the client operating system to be secure. Unless the mail program can get the key into the network without using the client operating system then there is scope for interception. The scheme can be made more complex by making the key an algorithm with the server providing some input data and checking the output. Intercepting the result is now less useful since the server is unlikely to supply the same input data next time. This is analogous to how we might check someone's identity over the phone by asking them one of a large number of unlikely questions such as what their mother's maiden name is. Nevertheless, the fact that the key is contained in the insecure container of the code of the program makes it vulnerable to attack.

For a program like this, security can be achieved by running it in a separate server. For something as useful and widely used as a mail program, creating a mail server is probably a sensible act of functional distribution. However, many programs which the creator may like to invest with extra powers cannot justify their own server. A traditional example is to implement a league table for the game of Moo [ALE71] which can be updated by the Moo program but not by individuals attempting to fraudulently improve their positions.

If we ignore this problem of extracting secret keys from program code, which only occurs when a user is running a program invested with some file access which would otherwise be denied to the user, then there are still a number of ways in which the security scheme could be implemented.

The first scheme is that of access lists. Associated with each file is a list of the users who may access it. Often this list is very stylised, perhaps dividing the universe of users into the owner of the file and everyone else. There are two points at which this access list can be checked. When resolving the textname of the file the access list is checked to see whether the use to which

the resolution will be put is allowed. Or when accessing the file the access list is checked to see if the attempted access is allowed; resolution is unrestricted (actually, since directories are usually files then resolution is itself partly protected by the same scheme). One of these two access list schemes is used in almost all coresident filing systems, and in all of the early file servers.

The biggest problem with access lists for a network file server is the requirement for a global scheme for naming and validating users. For Jack to permit one of his files to Jill, it is necessary for him to send a message to the file server holding the file. This message has to nominate Jill in such a way that the server can safely recognise her when she attempts to access the file. This requires that a complete copy of the user list for the entire network filing system be held at each server, or else at a number of authorisation servers. Furthermore, the scheme for verifying identity needs to be held at these sites too; usually this entails holding an encrypted password associated with each username. Neither of these solutions is attractive, especially when adding a new user. Apart from the restriction that all usernames must be unique, the new user list must be distributed to all sites holding it. If the network filing system is very geographically distributed, it is unlikely that centralising the authority to create new users is acceptable and so there is potential either for conflicting updates or a need for a complex locking scheme.

The second scheme is that of capabilities. Files are identified by tokens which are probabalistically hard to forge since they are drawn from an intractably sparse space of possible tokens. For example, they could be 64 bit integers. This is the scheme favoured by all of the universal file servers. Possessing the token for a file is itself sufficient to access the file. In other words, the files which are accessible are just those which are nameable. This causes a problem since directories are files;

knowing the token for a directory is sufficient to read the directory and so discover the tokens for all the files which it contains. It is thus not directly possible for a user to give access to a file to another user by the usual means of transmitting the textname via some external means (such as post, electronic mail or speech). If the recipient user can resolve that textname then any other textname in the directory is also resolvable since the token for the directory must be known. The actual token has to be transmitted, and people do not easily remember 64 bit numbers (one motivation for having the textnaming in the first place). However, there is no problem about identification of users. Users know the token for their top level directory which then gives access to all their files. More probably, they actually get this token from some other trustworthy server in exchange for a memorable password.

To avoid this inconvenient way of sharing files, the filing systems built on top of capability filing systems, for example the CAP filing system [DEL80], recast the capability access as an access list scheme. Only the filing system uses the tokens for directories, and it keeps them hidden from users. The fact that the client filing system possesses the token for a file by resolving a name in a directory does not necessarily mean that the user can access the file; the filing system may choose not to reveal the token for the file. This is how a coresident filing system works, where the filing system can access any file, but it administrates which users can access which files. Apart from the problem of hiding anything in the code of an untrustworthy system, there is another problem. There has to be an agreed conspiracy amongst all the client systems to uniformly restrict access. A filing system can only sensibly release the tokens for high level directories to programs which will not themselves publicly reveal hidden tokens which are easily discovered by traversing the directory structure. In practice, this means that all the filing systems end up being independent and never release tokens for directories. It is difficult to see how to implement a single textnaming and security scheme in such an environment of mutual

suspicion.

Even the tokens for ordinary files (as opposed to textname directories) cannot be freely released to users by the filing system. We will normally want other users to have only limited access, such as read-only access, to our files but the token gives all access. Unless we have multiple tokens for files granting different kinds of access then we still have problems organising that another user has access to our files. These multiple tokens themselves are a problem since they have to be stored somewhere too. Further, since they may also be stored in places unknown to the filing system, it is not possible to guarantee being able to revoke a granted access.

## A hybrid scheme

The problem with the access list approach is that it is clumsy to use over a network. The problem with the capability approach is that it is too inflexible to have only one token for each file since any finer grain of protection has to be provided outside of the filing system by agreed restriction within client operating systems. Even having multiple tokens for files only solves some of the problems. A solution lies in using a more hybrid approach.

In some ways the access list scheme and the capability scheme with multiple tokens are much more similar than appears at first sight. Access list schemes restrict access to a list of users. These users identify themselves by presenting their name and password either to the file server itself or to a third party which exchanges them for some form of hard-to-forge authorisation token. Since users demonstrate their identity with passwords, it is only a short step to regard authorisation lists, not as lists of users,

but as lists of passwords. To access a file it is necessary to have an appropriate internal filename and password pair, and these are hard to guess since the passwords are drawn from a large sparse space.

Correspondingly, the file capability scheme also constructs such pairs. Although it may not be possible to directly extract them from the token, a token splits into two parts, one which identifies the file and the other which is hard to forge which both validates the first part and discriminates among the different possible tokens for the file.

Both schemes therefore check the validity of access by requiring extra information, which we shall call a key. In the access list case, the keys are chosen by the users concerned; in the capability case they are chosen by the filing system. Even if the keys are chosen by the users, the internal filename will need to be drawn from a sparse space to ensure that an out-of-date internal filename is not rediscovered and accepted in error.

The second scheme, where the keys are chosen by the filing system, has a major disadvantage. Since the keys have no significance to users, they will need to be kept somewhere in directories. These directories are themselves files and so they will also have keys. Directories thus contain keys and so cannot be read by untrusted programs, which is something which we are seeking to achieve. Also, each user has to either remember at least one key, or rely on a trusted third party to provide it in exchange for validated identification, probably a name and password.

The second scheme, with keys chosen by users, is very much better. Now, since the keys are remembered by users, they do not need to be stored in directories. Directories can thus be comparatively unprotected objects since discovering the internal filename for a file from a directory is not a sufficient condition

to be able to access it. There are, however, two problems.

The keys are just passwords, or more probably transformed passwords. Since passwords need to be typed in public places and are occasionally written down, they occasionally become compromised. Users therefore need the ability to change their passwords. It is convenient if this does not require re-keying all their files, although changing passwords is rare enough that this might be an acceptable overhead to avoid further complication.

The second problem is that a user needs to be able to permit access to a file to another user without knowing the recipient user's password. One way to do this would be for donor and recipient users to choose a key (password) specially for this purpose, although this raises the problem of key transmission if electronic mail or the telephone system are considered insecure, and the two parties are too remote to meet. The need for this advance agreement may, therefore, be inconvenient. One situation in which no problem arises is when the author of a program wishes to permit it access to a file. Here donor and recipient (user and program) are really one individual, and so there is no problem about agreeing and transmitting a key.

Both of these problems can be attacked by creating a trusted identification server and adding a level of indirection. Instead of keying files directly by the password, they are keyed by special hard-to-guess authorisation keys. Since these are never typed nor displayed, they do not need to be changeable. A user (or the client operating system) aquires their authorisation key from the identification server, probably by presenting a name and password although it could be by inserting a physical key or card, or by some more futuristic identification scheme such as fingerprint recognition.

Now users are free to change their passwords without significant overhead. Permitting access to a file requires the donor user to request the identification server to add the recipient user's authorisation key to the file's access list, without revealing this key to the donor. This is really the way coresident filing systems work, with the operating system performing the functions of the identification server.

Unless the network has identification servers for other reasons, which is possible but by no means certain, constructing them especially for file security is unattractive. This would require that the usernames, the authorisation keys and any data, such as passwords, needed for verification be distributed to them all. As already discussed, this leads to update problems. In the wider context of the rest of the network, this expense may be justifiable. For the filing system alone, it is not. Here the identification server is not so much for identifying users as for allowing a file donor to instruct a file server to add another user's authorisation key to a file without needing to know it (or, indeed, being able to discover it). Mere verification of identity can be performed without needing an identification server by using passwords as follows.

The authorisation key that is used for file access is encrypted with the password, and the encrypted form is kept in a known (but insecure) place. Users aquire their authorisation keys by fetching the insecure encrypted form and decrypting it with the password to get the authorisation key. Changing a password is effected by re-encrypting the authorisation key with the new password and storing it back in the insecure place. Instead of keying their files directly with passwords, users key them with the authorisation key; since this is never typed or displayed it cannot be discovered easily and so need not be alterable. However, the current password is necessary to discover it. Passwords still need to be agreed to allow access to other users, but the massive re-keying necessary when users change their everyday passwords is

avoided.

One problem with the presentation of keys still exists. Users have keys to access their own files and a number of keys to access other users' files. The client operating system presumably manages these keys but it has a problem deciding which key it is appropriate to present to the file server. Access to the files of other users is either the exception (under normal circumstances) or the rule (when working on behalf of the other user) and so cycling through the keys in an order defined by the user will normally be efficient. Alternatively, multiple keys can be included in file access requests to the file servers. Two keys seems a good compromise, although almost any number can be justified by a suitably contrived example.

This leaves us with the following scheme, which has the required versatility as far as permitting access to files goes. Further, it allows textnaming to be handled in client systems without compromising the entire textname graph. Other systems either perform all textnaming at the server or rely on client textnaming systems to adhere to a self-imposed scheme to restrict file access, due to naming and security not being distinguished.

Files are identified by an internal filename drawn from a sufficiently sparse space to make it difficult for an invalid filename to be accidentally accepted. For reasons connected with naming rather than security, these internal filenames should be unique across the whole network.

Associated with each file are a number of keys, and associated with each key are some access rights. These rights permit or bar various operations on the file, in particular, to write the file, to read the file, and to issue further keys for the file. When a file is created a key is provided by the user which will grant all rights to the file. One of these rights is the ability to create further keys and so additional keys can come into existence,

presumably for the use of others. Granting a key of zero any rights at all conventionally grants the associated rights to all keys; this can be used to make a file readable by the general public, for example.

Whenever a file is accessed, a key must be presented which must grant the type of access being attempted. Directories are just ordinary files, but they have an attribute which marks them as a directory since the textnaming scheme needs to be able to distinguish them.

Changing a password requires the key on each relevant file to be changed and so it is an unfortunately expensive operation. This is done by adding a new key granting all rights and then rescinding the old one. To reduce the overhead in the case of a user's ordinary password, the technique of encrypting the authorisation key (as described above) can be used. Forgetting a password, of course, renders any password alteration impossible. In the same way, forgetting a password on a conventional terminal system prevents access to reset the password. In both cases, some form of system manager has to intervene. This requires the existence of a loophole of some sort, a universal password which grants any access to any file. All filing systems have some similar facility although it is usually little discussed since it exposes the security in an obvious way.

Permitting access to another user requires an agreed intermediate key. A key is added granting the appropriate access and the recipient presents the key when performing the access.

In practice, a client system might make the facilities visible as follows. A user logs onto the client system by presenting a username and password. These serve to set defaults for working directory textnames, and to set the first key. Other operations such as obeying some startup command file may then take place. At any time the user has a set of quoted passwords, each of which is

turned into a key by some standard well known algorithm, the same algorithm for all client systems. Passwords may be added or removed from this set. When attempting a file access, these passwords are used in turn if the access fails through presenting an incorrect key. The scheme is thus fairly transparent to the user (apart from having to quote the passwords) although occasionally the password search path could get long, and hence slow. When a client logs off the client system, all passwords are erased to prevent a passer-by inheriting access rights.

To permit file access to another user a password is agreed. The recipient may have a comparatively low security password used for such purposes anyway, or else a new one could be invented. The donor adds the password (turned into a key) together with the required rights to the access list for the file, and the recipient adds the password to the set of quoted passwords when access is desired. If the donor user decides to rescind the access, then the password (turned into a key) is removed from the access list for the file. To allow free access to the file, a null password (conventionally turned into key zero) is added to the access list granting the appropriate rights.

Alternatively, a client system could choose to cast the security scheme more traditionally, where file access is permitted to named users. It could do this by having a means of associating a key with each user (either by secret algorithm or by a special file containing a table). To permit a file to a given user, the recipient user's key is added to the access list. When users successfully log on to client systems, their keys are set up for inclusion in all access requests. The permitted access would only be available through the particular client system implementing the scheme. This is preferable to the case where the security is breached by switching to a different client operating system.

## Summary

This scheme is fairly simple to implement in both the client and in the server, it is easy to understand at the user level and it is efficient in almost all cases. It is really an extension of the scheme which we used in our early filestore, where directories were protected by quoted passwords. Overall, it is well-suited to distribution since the global security scheme is constructed out of independent local implementations at each server. There is also no requirement for third party servers to validate identity.

# Chapter 7: Consistency

One of the major problems in the design of any filing system is that of preserving consistency in the face of equipment failure and competing access by several clients. We would like the filing system to remain consistent at all levels at all times. This is unlikely to be achievable since almost all changes which take the filing system from one consistent state to another do so through a number of inconsistent states. For example, when creating a file, there is almost certainly a stage at which some free disc pages have been allocated to the nascent file but which have not yet been properly inserted into the file location map.

It is, however, possible to create the illusion of consistency. This will involve careful control of just what data is visible to clients at any time, and a carefully designed recovery strategy to cope with the system returning after equipment failure.

This problem is far worse in a multiple server network filing system than in a coresident filing system. Firstly, only a proportion of the servers may crash and we want the crashed servers to recover without interrupting service at the others. Secondly, locality considerations mean that it is so much more difficult to check consistency (by running any sort of file structure verifier, for example) that it has to be prevented from occurring. This means that cooperating servers need to be synchronised in case one of them should crash before completing its share of the operation.

This section starts from the loose concept of consistency and develops the outline of an implementable distributed filing system which preserves consistency. The way in which this is done depends on a number of assumptions.

Firstly, it is assumed that the guarantee of consistency is a good thing. Even though some of the events which may cause inconsistency are rare, the lack of a guarantee forces an inappropriate style of programming and operating system development. Ostensibly simple operations become very complex if applications programs have to preserve consistency themselves.

Secondly, it is assumed that the efficiency of the filing system is important. The servers of the filing system are shared resources and many of the operations that they perform are done in synchrony with client programs; a user will often be waiting at a terminal.

Thirdly, it is assumed that having multiple versions of files, as found in DEC files-11 [DEC78] or Tenex [BOB72] is desirable. Apart from the obvious advantages of being able to look back at the past history of a file, one particular advantage that this gives is the ability to write a new version of a file whilst reading the old version; this saves the contorted way in which editors and similar programs (in systems without this facility) have to write a temporary file and then either copy the data back to the original file or rename the temporary file to replace the original file. Further, it gives users the confidence to run a program to create a new version of a file, safe in the knowledge that the old version will still be around even if the new one is erroneous in some way.

Finally, it is assumed that disc storage is cheap enough for the ideas to be workable. Many systems already provide multiple versions of files, even files for which multiple versions are not really required. Systems which do not provide multiple versions of files often use storage up in more hidden ways, when users create their own multiple versions using different textnames.

Notions of consistency

Consistency is a rather vague notion and covers a huge range of constraints. At one extreme, these consistency constraints merely reinforce our idea of what a filing system should do. If we write a file and read it back again, then the filing system is inconsistent if we get data from a file that we have never seen before. At the other extreme the constraints are those common in the implementation of databases. The classic example is the bank balance problem. We withdraw a sum from an account held in one file and credit it to another; the system is inconsistent if the total of the amounts held in the two accounts taken together varies. Consistency is thus a guarantee that a filing system performs to our expectations; to put it another way, it is the things that we expect to be true. We are unlikely, as an ordinary client, to be able to verify whether they are, in fact, true, so we need guarantees from the filing system instead. Even in cases where we can check consistency, guarantees are more useful. Detecting inconsistency is only half the problem since it needs to be corrected; it is far better not to allow inconsistency to develop in the first place.

In another sense, a well designed filing system is always consistent. All that varies is the level of consistency. When the filing system goes from one consistent state to another, it will normally do so through inconsistent states. If these states were really completely inconsistent, then it would be impossible to perform any sort of recovery following a crash. These states are merely less consistent. However, this lower level of consistency should allow us to recover to a higher level, and from there to a still higher level and so eventually to the level which we guarantee.

It is sensible to have a more precise definition of just what is meant by consistency. Consistency is a set of assertions about the state of the filing system. The filing system is consistent if

every assertion in the set is true; if any is false then the filing system is inconsistent. It is important to note that some of the assertions will be at client application program level, and hence unknown to the filing system.

Consistency can be viewed as a two-dimensional concept. One dimension describes which data within the filing system is acceptable as a domain for the consistency assertions. The other dimension describes which data outside of the filing system is also acceptable in the domain. Loosely, one dimension describes the strength of the consistency and the other the context in which this strength can be tested.

## Levels of consistency

The levels of consistency form a continuum. A precondition to even testing for consistency at one level is that the filing system is consistent at lower levels. Out of the continuum of levels of consistency, three levels in particular stand out. We shall call these levels zero, one and two. They correspond to allowing the domain of the consistency assertions to be physical addresses of the stable storage, the filing system structures themselves (but not the data in the files) and the data within the files. Correspondingly, the assertions allowed therefore correspond to those which we will guarantee to be always true, those which the filing system itself can test, and those which are unknown in detail to the filing system.

Level zero consistency allows the domain of the assertions to be the physical addresses used to recover data from the stable storage of the filing system. The important exclusions from the domain of level zero consistency assertions are the structure or contents of any data structures held in the store of the file server or client machines, and the actual contents of the stable storage. The

in-store data structures are considered too volatile to form any basis for recovery. In the absence of such in-store data, the stable storage of the filing system consists of a lot of bits and to deduce any structure requires a context in which to interpret them. This context is provided by the level zero consistency assertions.

Note that we have already assumed some assertions to be true to reach this level. For example, magnetic discs actually consist, not of bits, but of magnetic patterns, and it requires a context in which to interpret these as bits. Loading a disc written with one disc controller into a drive using another controller makes it clear that this context is not preordained. Since we are interested in the design of filing systems and not of controller logic, we assume the ability of the stable storage hardware to correctly interpret the stable storage. For current controllers this is a reasonable assumption. If there were a disc controller with a completely writable microprogram then we would need additional assertions about the microprogram to be used to ensure a uniform interpretation of the magnetic patterns as bits.

We assume that the hardware, on being given a physical storage address, will deliver either a bitstring or an error status. The level zero assertions relate certain physical disc addresses to interpretations of the bitstring, or parts of the bitstring, so delivered.

We need this almost trivial definition of level zero consistency. Since any part of the system may crash at any time, there have to be assertions which can be guaranteed when crash recovery begins, before accessing the stable storage. During the recovery process following a failure, the recovery program can take as a guaranteed precondition that the stable storage is already consistent at level zero; all the level zero assertions are true.

Most filing systems have only a few level zero consistency assertions. For example, consider the Unix [RIT74] filing system. The two critical disc storage structures are the super-block, which contains the parameterisation of the disc and some of the chain of free pages, and the i-list, an area of disc containing all the file headers. The level zero assertions are that page 1 on the disc is the super-block and the i-list is a contiguous area of disc starting in page 2. A crash recovery procedure can thus locate the super-block and the i-list, an essential first step. A more subtle example is given by the operating system for the Alto [LAM74]. In the filing system of this operating system, disc pages are labelled with a few bytes of housekeeping data; the level zero assertion is simply that each label does in fact correctly describe the data in its page.

Level one consistency is file structure consistency. The domain of the consistency assertions are the stable storage structures managed by the programs comprising the filing system. The notable exclusion from this domain is the contents of files that are maintained by the filing system as uninterpreted raw data. Changes to the state of the filing system which take it from one level one consistent state to another will do so through states which are not consistent at level one. For level one consistency to be useful we require that it is reconstructable. If the system should stop in a level one inconsistent state, then it must be possible both to detect that this has happened and either back out to the old consistent state or continue to advance to the new level one consistent state. By definition, it will be consistent at level zero. On its own, this is not enough. We also need assertions at a level of consistency intermediate between zero and one. These relate to the type of level one inconsistent states which may arise. They will normally be concerned either with the order in which critical changes are made when moving from one level one consistent state to another, or else with the reconstruction of redundant data. If this intermediate level of consistency is well designed, then it will be possible to reconstruct level one

consistency following a failure, either by returning to an old level one consistent state, or by completing the changes needed for the new level one consistent state under construction at the time of the failure.

For an example of level one consistency, consider the Unix filing system again. One level one consistency assertion is that every page on the disc is either in a file, or in a chain of pages for allocation. One of the intermediate level consistency assertions is that if a page is both in a file and in the chain of pages for allocation, then it is really in the file; if a page is neither in the chain nor in a file, then it is really in the chain of pages for allocation. Since part of this chain of pages is held in store, a sudden stop of the whole operating system may leave the filing system in a state where the level one assertion is violated. One way to restore level one consistency would be to delete any file containing a page also in the chain of pages for allocation, and to create a special file containing all the pages which are neither in a file nor in the chain. The intermediate level assertions make this the wrong thing to do. It should be restored by reconstructing the chain of free pages to be all pages not contained in any file.

Level two consistency is file-data consistency. At this level, the data in files is consistent at all levels of abstraction imposed by clients. In the absence of competition from other clients and of equipment failure, this level of consistency is the responsibility of client application programs. Since the filing system can only directly control level one consistency it must provide suitable primitives to clients to allow level two consistency to be couched in terms of level one consistency. These primitives must ensure two things. Firstly, no two clients must be allowed to interfere with each other since only in the absence of such interference can a client guarantee level two consistency. Secondly, that if a crash should occur, then the filing system is in a state where it may either be restored to the old level two

consistent state or advanced to the new level two consistent state.

As an example of level two consistency, consider a file containing a large number of records giving names and phone numbers. Two secondary index files contain tree structures to map respectively names and phone numbers to record numbers in the main file. A level two assertion would be that these secondary index files are correct. Every record in the main file is correctly indexed in the secondary files; every entry in the secondary files does correspond to a record in the main file. This is not trivial to guarantee if, for example, the system should crash whilst adding a new record. Since this requires alterations to three files, things may well get out of step. The problem is clearly further complicated if the files are held on different servers (admittedly unlikely in this example).

## Contexts for consistency

The other dimension of consistency is that of the context in which it can be seen to be correct. The notions of consistency presented so far have taken no note of anything outside of the filing system. Perhaps the most important outside factor is that people using filing systems remember (or write down) the textnames of their files, or remember (or write down) their contents. Whether we permit data from outside of the filing system to be in the domain of the consistency assertions greatly affects their practical usefulness. We will distinguish two contexts for viewing consistency.

The weakest context is when we do not allow any external information to be part of the domain of the consistency assertions. We shall call this internal consistency. On its own, this is not a particularly useful level of consistency since it can easily be

achieved by drastic measures such as reinitialising all discs to be empty following a crash. Since the purpose of a filing system is the long-term storage of data, achieving consistency by losing data and then erasing all trace of its having been there is not particularly useful. It is, however, the only context in which the filing system can test the consistency assertions.

On the other hand, we can allow information from outside of the filing system to be in the domain of the consistency assertions. We shall call this external consistency. Since we could duplicate elsewhere any data stored in the filing system, and have assertions that the duplicate is in-step, external consistency is a strong requirement for long-term storage. Once the filing system has confirmed successful storage of some data, then it must never autonomously lose it since it would violate an assertion that the copy elsewhere was the same as the stored copy. This is a much stronger guarantee since it places restrictions on the operations which crash recovery can use. Naturally, if the system should crash, then there will be uncertainty about whether some initiated operations have completed. Internal consistency implies that they must succeed completely or fail completely. External consistency requires that it must not have been possible to find out which until it was certain. In particular, confirmation of operations must not be given until they have reached a point of no return which crash recovery will not undo. For levels lower than one, there is no external data which could sensibly be added to the domain of the assertions.

Table 1: Summary of consistency assertions

| Level | | Internal example | External example |
|---|---|---|---|
| 0 | Axioms. | The first readable page on the disc gives the disc addresses of the roots of all the filing system structures. | No relevant external data. |
| 0..1 | Assertions about how level one inconsistency is permitted to occur. | The freepage bitmap is always flushed to disc before the allocated pages are recorded as part of a file. | No relevant external data. |
| 1 | Assertions about filing system structures. | No page is in two files. | There is a file called A in existence. |
| 2 | Assertions about file contents. | The contents of file A are a copy of the contents of file B. | The contents of file A are a copy of this paper tape. |

Failure modes

In the event of a crash of the filing system, the recovery procedure will restore consistency to some level and some context. We would like to be able to provide guarantees as to exactly what these will be, but to do this requires some consideration of the way in which the system can fail.

The simplest failure of the system is a sudden stop of a file server. This can occur for a large number of reasons ranging from a detected hardware error such as a memory parity error, to a failure of the mains power supply. In a multiple server filing system the situation is complicated by the fact that varying numbers of servers can fail and by the fact that the recovery procedure should not involve loss of service on the servers which have not failed. The characteristics of a sudden stop are that some of the servers in the filing system are reset to a standard state [LAM79], normally by reloading the code which runs in them. They then have to recover the consistency using only the information in their stable storage and at other servers which have not crashed. This is the most common failure, occurring anywhere from once a day to about once a year, and any filing system should be able to survive it. It should be less common in a file server than in a filing system coresident with its operating system. Most crashes in this case are due to the underlying operating system collapsing.

Another similar failure to the sudden stop of a server is the failure of the network. This is much simpler to deal with than the failure of servers in the filing system. If the network restarts quickly and cleanly enough, then this failure may require no recovery at all. On the other hand, a complete clean up of all servers may be needed if the network is broken for so long that operations in progress are not considered to be worth restarting where they left off. This is fairly simple, since the servers will not have lost their in-store data structures (unless they have also

crashed in the meantime).

The next simplest failure is that of a sudden stop with some corrupt stable storage. Magnetic discs are divided into units which we will call pages. A page is the smallest amount of data which may be written to the disc. Error detection is done on a per-page basis. If the power to a disc drive is cut whilst it is actually writing a page, then it is possible for that page to be left in a state where neither the old data in the page, nor the new data being written is readable. The implication of this is that some data must either be duplicated or not updated in place. In either case we can guarantee the availability of either a good copy of the old data, or a good copy of the new data. We do, however, assume that the page is detectably bad. Modern disc controllers write an error detecting code of enormous width with each page and other forms of stable storage such as optical disc or bubble memory have similar characteristics since the data has to be divided into units for error detection purposes.

The most difficult failure to deal with properly is that of data loss. This lies in a continuum between two cases. At one end, there is a single page degrading. A page on the disc which previously read without error (probably immediately after being written) becomes unreadable. At the other extreme is whole volume loss. All the data on the disc becomes unreadable. This can be due to a head crash or other mechanical failure destroying the physical media, or due to a disc controller error destroying large quantities of data (by leaving the write head turned on, for example). Many external catastrophes, such as a destructive fire, would come into this category too. Modern Winchester discs have a much lower head pressure than older discs, and can survive the head coming into contact with the disc. While not completely eliminating head crashes they make whole volume loss much rarer.

To preserve consistency in the face of such a failure is very difficult, since it clearly requires multiple copies of all the data written. The normal compromise between proper journaling [AST76] and no multiple copies at all is to take occasional snapshots of part or all of the filing system, usually to some cheaper medium such as magnetic tape or offline disc packs. In the event of loss of data, then any lost data can be restored to its state when the last snapshot was taken. These backup snapshots are usually taken in a rather ad hoc way, so that few guarantees can be given about consistency after a restoration. Whether more drastic measures are needed is a question of the balance between how catastrophic each disaster might be, weighed against its small probability of occurrence.

If we decide that regular backup provides insufficient safety, then some form of journal is required which preserves duplicate copies of all pages written to the disc (together with a certain amount of housekeeping data). For a filing system, it is probably too expensive to keep a journal using one of the techniques from database implementation. A database journal usually preserves all the updates. In many databases these are comparatively rare. Furthermore, only the updated records need be journaled, rather than the entire physical disc pages in which the records occur. The data which the filing system sees as changed, the disc pages written, is large although the data actually altered may be small. It is clearly more economic to preserve the editing commands from a session than both the old and the new files, though it would be almost impossible to detect these automatically within the filing system. If regular backup snapshots are taken, then it is only necessary to preserve duplicate copies of all data written which has not yet been backed up. If only the latest copy of any particular page is preserved then the space required may be acceptable. This other copy should be kept on a different disc drive in case of head crashes; ideally, it should be kept on a disc drive on a different disc controller in case of an undetected controller failure; ideally, on a different processor and so on.

Complete safety is an unattainable goal; few of these backup measures would survive a nuclear war, an event with a depressingly non-zero probability. Even to survive other catastrophes such as a fire, the second copy should be kept spatially remote from the main copy. However, the most likely failures can be countered by keeping the backup copy on a different disc drive from the main copy. The probability of errors too drastic for this to be safe is tiny; in many environments, such errors would be disasters on a scale where the loss of the filing system would be a minor part of the total cost.

Most filing systems offer few guarantees in the face of failures of one sort or another. Although such failures are rare, the lack of guarantees has much wider repercussions. Filing systems themselves minimise the effects of failures by carefully controlling the order in which pages are updated on the disc. User application programs become extremely contorted if they have to take the same sort of precautions without being in the same privileged position. For example, on systems which keep a cache of disc pages it is often impossible for an application program to discover whether or not a given updated page has reached the disc.

Most filing systems preserve internal level one (file structure) consistency over most crashes, doing their best to provide external level one consistency. This is achieved by running some form of check or rebuild program, and often following this with some wizardry requiring fairly detailed knowledge of the filing system should trouble be detected. For example, Unix [RIT74] has two programs, dcheck and icheck, to check that all the filing system's level one assertions are true. The system manager is exhorted to run them after every crash [BEL79] and in a worst case accident, user files may be deleted to ensure that the filing system is level one consistent. Not only is no level two consistency guaranteed, external level one consistency is sacrificed to recreate internal level one consistency, since existing files may be deleted.

With only a guarantee of level one consistency, a client recasts any desired level two consistency as level one consistency by only ever altering whole files. Files are completely read and completely rewritten, even when it is inappropriate to do so. The problem is then pushed down into the filing system; either the new file has been created or the old one still exists. This procedure is usually unsafe since the filing system takes no particular care to guarantee it. Luckily, crashes are rare and crashes which corrupt a particularly critical filing system structure page are rarer. A strategy such as this which minimises the number of critical pages is often, therefore, successful. This very success, in the absence of strong guarantees, can itself be a problem. Due to overconfidence, no other method is used to preserve consistency, and the feared catastrophe of losing both the old and the new versions can be extremely costly.

In the case of catastrophic data loss, most filing systems enable files to be restored from a previous backup snapshot. This is, unfortunately, done only to internal level one consistency. We would like internal level two consistency (assuming that journaling has been considered too expensive to make recovery to external level two feasible). Thus two files may be restored to different previous states. Luckily, most files are unrelated to others and this may not be a problem. However, even source files of programs are often inter-related and it is certainly undesirable to be presented with a set of sources which will not compile together.

We have not considered the possibility of software error causing a failure. This amounts to assuming that all errors due to software are detected before they affect the contents of the stable storage of the filing system. At this point they are either corrected or a suicidal system crash is induced. This assumption is, at best, questionable. A certain amount can be done to make the on-disc data structures more resilient by carefully constructing them so that it takes more than one error to destroy the structure [TAY79]. In the case of a filing system, this

translates into a requirement that more than one erroneous disc transfer is needed to cause trouble, and checking for the occurrence of trouble is quick and simple. Ultimately, however, there is no substitute for correct software.

Table 2: Summary of failure modes

| Failure | Example causes |
|---|---|
| System suddenly stops. | Processor fault. Detected software error. |
| System suddenly stops and corrupts the page of stable storage being written. | Power supply failure. |
| A page of stable storage degrades and becomes unreadable. | Very rare; air filter failure on magnetic disc drive. |
| A whole volume of storage become unreadable. | Magnetic disc head crash. Optical disc melts in a fire. |
| Many volumes of storage become unreadable, including offline volumes. | Natural catastrophe such as fire or earthquake. |

Multiple versions

Many filing systems have some notion of multiple versions (sometimes called generations) of a file [DEC78,BOB72]. These are normally files of the same name, but with a distinguished version number. They are usually unrelated in anything but name and are certainly not viewed as different states of the same object, as a sort of history of the file. Unfortunately these multiple versions only exist for files which are completely rewritten and not for files which are updated, and so they are of no direct use for preserving consistency. Their main use is in correcting user errors by allowing a single file to be rolled back to a previous state, or by allowing a comparison of the current state with a previous state. This is not unimportant since the main cause of loss of data from a filing system is accidental user requests to destroy files. However, the most structured files tend to be those we update in place. It is these files from which it is easiest to lose information by losing the structure, either due to user error or due to failure. Multiple versions are thus of limited usefulness while they exclude these structured files.

On the other hand, there is a class of files for which multiple versions are completely unnecessary. These are redundant files. Redundant, in the sense that their contents can be regenerated from other files within the filing system. For example, the object code of a compilation can be regenerated from two files, the source code of the program and the executable code of the compiler. Preservation of multiple versions of such files - indeed, preservation of consistency at all - is usually more expensive than the regeneration of the file in the unlikely event that it should be lost.

Traditionally, filing systems have split files into these two classes: files which are rewritten completely and files which are updated in place. The consistency guarantees are completely different for these two classes of files. Files which are

rewritten completely are not rewritten in place; that is, new disc pages are allocated for the new copy of the file so that a fallback to the old is available in the case of failure. Many filing systems also preserve these old versions for additional safety. Files which are updated in place have no old versions preserved and have no guarantees about their state if a multiple page update should be interrupted by a failure.

However, there are two classes of files for which users would like different consistency guarantees: ordinary files and redundant files. This corresponds to a split in the way the user sees the files, rather than a split in the way the filing system sees the files. We are prepared to pay for strong guarantees about non-redundant files by requiring no guarantees at all about redundant files. With the exception of the Cambridge file-server [DIO80], current filing systems only allow this distinction between normal and redundant files to be made, if at all, for archive purposes. Redundant files are marked as vulnerable and not to be archived. It is, though, a more useful distinction than this; they are so vulnerable that no special care is needed about their preservation at all. The Cambridge file-server guarantees external level two consistency for updates to individual files if they are marked as important. If a client updates two such files, then although each one is either completely updated or unchanged, there is no guarantee that both are updated or unchanged; one could be unchanged and one updated.

If we make this distinction, then a file can be regarded as a two dimensional object, it now has a time dimension. The multiple versions of a file form a history in much the same way as Minkowski's life-lines in the theory of relativity. With the prospect of write-once video discs, and the decrease in the cost of disc space, this idea has started to gain respectability [REE78,SCH77,COP80]. However, there is an important difference between the state of a file and the life-lines of relativity. We want our files to move discretely from one state to the next, not

continuously. We construct the new state elsewhere and instantaneously instantiate it. In the event of failure we can discard the half-built version. Provided that we maintain multiple versions of all non-redundant files, then these life-lines begin to form the basis for a filing system which preserves level two external consistency at all times.

If we always create new versions, then there is no update in place - no page is ever overwritten with a new updated version of itself (we need to except redundant files). This is desirable [SCH78,COP80] since update in place loses information and is merely one way of achieving a balance between long-term storage costs and other costs. As the cost of long-term storage drops, discarding information for economic reasons becomes less necessary. With write-once video discs on the horizon it may even become impossible.

If more than one file is updated by a client, we want to make sure that either all the new versions, or none of them, are instantiated. Moreover, since the versions are distinguished by their time components, we want their times of instantiation to be identical. If not, there will exist times in the past when our consistency assertions will fail. If we choose to examine the states of the files at a time between instantiation of the first and instantiation of the last, we may detect inconsistency. This is particularly important since backing up files for safety is likely to make extensive use of the ability to access old versions of files. If we backup all files as they are at midnight then we get a consistent version of all files. Furthermore, this can be done without taking the filing system out of service as, typically, needs to be done with current filing systems.

Rather than using time to distinguish different versions of the same file, we could have created a new internal name. Every version of every file has a different name. This suffers from the same disadvantage as is common with applicative languages in that

changes have to be propagated throughout data structures. If we give a new version a new name then the new name will need to be stored in a directory; this is a file so we will get a new name for the directory which will need to be stored in the next higher level of the directory structure, and so on up to the root of the filing system. Directories would also need to be much larger, since the names of all existing versions would need to be stored rather than one name for all the versions. Using a single filename with a time component actually reduces the amount of directory manipulation since there is no new name to be stored in the directory; the new name is the same as the old. Directory manipulation only needs to take place when a file is created or finally destroyed.

Multiple versions of files updated in place do not require vast amounts of disc storage for the different versions nor the overhead of creating them. By choosing a suitably indirected disc structure for the representation of a file, pages can be shared between consecutive versions if they are unchanged. It is not necessary to write more than a handful of pages to update a single byte in a megabyte file; certainly, it is not necessary to write a megabyte to create the new version.

## Time

If we are going to control the association of versions of different files using timestamps, then we need a clear view of what these timestamps actually mean. We want the timestamps to provide a global view of time but unfortunately time is a local concept. This is true in the real world, but it is only a problem when the transmission speed of a message (of some sort) is of a similar order of magnitude to the accuracy with which we wish to measure the time, and so it is not a problem in everyday life. It does become a problem though in, for example, relativistic physics or in

the design of synchronous integrated circuits. Taken to an extreme, it is clearly ineffective to synchronise watches by post.

To derive some constraints on how we must manage time locally to give the illusion of global time, it is sensible to consider the ways in which a view of global time may break down. The most obvious way is for time to be seen to run backwards.

For example, two events A and B which are detectably ordered with A preceding B are assigned timestamps which indicate that B preceded A. For example, a program source file is updated and receives a timestamp of 3pm. This source file is then compiled and the object code file receives a timestamp of 2.59pm. The illusion of a global time has broken down. This is an internal failure in that the inter-relationships of the files are all stored within the filing system itself.

There are also external failures of the global view of time when at least one of the inter-relationships of files is external to the filing system. A file is updated and receives a timestamp of 3pm. This fact is communicated to another client which logs it in a second file which receives a timestamp of 2.59pm. If this is unconvincing, it is because it is hard to imagine files which are loosely enough related that they are not updated together but closely enough related that the timestamps matter. Provided our clocks are reasonably synchronised (not hours apart, for example) external failures of time are not a problem. After all, a crash could intervene to prevent the second update happening at all.

The internal failure modes are failures of a sort of causality constraint. If the new version of file A is dependent on information held in a version of file B, then the timestamp for A must be later than that for B. The external failure modes constrain by how much local time at different sites may be out of synchrony. This error must be less than the time from creating one new version, communicating that fact and creating another new

version of a different (or, indeed, the same) file. This will be some function on the speed of the network, the speed of the disc and the details of the implementation. As a lower bound we can use speed of light constraints, but these are rather too severe to live with.

The internal failure modes constrain events which are known to be associated. The external failure modes constrain events which could be. Whilst they are not known to be associated, they are spaced apart enough in time that they could be. There remain a class of events which are not associated, and, furthermore, are so close together in time that they could not possibly be associated; it would be impossible to communicate the occurrence of the first event to the initiator of the second. We call such events concurrent, and their timestamps may occur in either order. For example, if two people, one in New York and one in London, create a file on their local server within a fraction of a second of midday GMT, then these events are concurrent. There is not sufficient time for the creation of one file to influence the creation of the other (the intention to create may have influence, but not the confirmation that this was, in fact, done).

The timestamps actually serve to order events. In [LAM78] a theory is developed for arbitrary events. We shall restrict events to the creation of a new version of a file within a filing system. To order two events, we compare their timestamps. If we compare the timestamp for version A of one file with the timestamp for version B of another, then there are three outcomes. Version A came into existence before version B, they came into existence simultaneously, or version B came into existence before version A. If the two timestamps are taken from different clocks, inevitably only synchronised approximately, then there are again three cases. Version A must have come into existence before version B, even if the clocks have drifted to be out of synchrony by the maximum amount. Version B must have come into existence before version A under the same worst case conditions. Versions A and B were

created within the error window of the two clocks. These three cases do not directly correspond to the three orderings of the timestamps and so we have a problem. We want to ensure that if the timestamp of event A is less than the timestamp of event B then A preceded B or was concurrent with it.

Assigning timestamps to new versions of files places us in a considerably better position than a system assigning timestamps to arbitrary events. Associated with each new version of a file are three times. Firstly, the time at which we actually complete production of the new file. Secondly, the time which we assign to the new version; in effect, the time at which we say that we created the new file. Finally, the time at which we allow the new version to become visible to the outside world. If we ensure that this last time is later than the other two, then these other two events can occur in any order. In particular, we can assign a timestamp to a new file for a time which has not yet occurred. For example, at 2.59pm by our local clock we can create a new version of a file timestamped at 3.00pm, provided we let noone see it until 3.00pm.

Many of the problems associated with timestamping events can be obviated by delaying the time at which they become visible outside of the system. We cannot, however, delay visibility for much longer than it takes to create and instantiate a new version of a file. Otherwise we may update a file and then receive a request from a remote server to create a further version with an earlier timestamp. This problem only occurs when the second update is initiated remotely; this gives slightly more latitude in the acceptable clock error in a desirable way. The further apart the two servers are, the larger is the acceptable error in the synchrony of their clocks.

The requirement for close synchrony can be considerably relaxed by having all servers involved agree on a suitable time before starting to instantiate the versions. Unfortunately, it will turn

out that this agreement cannot easily be combined with the messages needed between servers to coordinate the instantiation. Either the timestamp needs to be known from the start, which requires a message between the servers to decide it, or else each new version will need to be written twice, once to protect against a crash and once to timestamp the version correctly.

The other key thing which we get from this strategy of delayed visibility is the ability to create versions of more than one file, all of which receive the same timestamp, even if the different files are held on different servers with different local times. This is particularly important. If several files are updated around 3.00pm and a backup taken later in the afternoon of all files as they were at 3.00pm then we probably do not particularly care whether we get the new versions or the old versions; but we certainly do not want a mixture due to the different versions receiving different timestamps.

This technique of delaying visibility directly handles the internal failure modes since it is possible to know how long it is necessary to delay. We can only avoid the external failure modes by actually keeping our clocks acceptably synchronised. In [LAM78] an upper bound on the drift out of synchrony is developed. The maximum drift is a function of the unpredictability in the transit time for a message, the frequency of messages containing timestamps, the connectivity of the site to site communication graph, and the drift in the hardware if the clock if left to run freely. For sites on the same local network, every site is directly connected to every other. Even if we assume that they communicate as slowly as once a day, for a crystal controlled clock the error is dominated by the unpredictability in the transit time for a message. For distant sites, perhaps accessed through gateway machines to a long haul network, the error is likely to be dominated by the lack of communication as well as the unpredictability of the transit time for a message across the network. Two clocks at different sites are difficult to keep

synchronised if the sites only communicate once a week across a network with very variable transit times. This may be especially problematic since two clients to the servers on this network may have a very fast backdoor communication link. They may be in radio contact, for example.

All this study of clocks assumes that the clocks are independent. This is rarely the case. Whilst local clocks may be crystal controlled on a battery in the event of power failure, they normally use an external time-base for reference. In Britain, there are two important external time-bases: the mains frequency and the VHF time signals broadcast from Rugby.

The 50hz mains frequency can vary locally according to the mains loading, especially if load varies suddenly. However, over long periods, the number of cycles in the mains is accurate. Two clocks counting mains cycles anywhere in Britain will not have any long term drift. At any point in the future the expectation of the error (in the probability sense) is zero. This is not the same as the absolute error in the clocks since this depends also on the initial error when the clocks are set and started, and on short-term local perturbation in mains frequency.

The VHF time signals consist of astronomical time derived from an atomic clock and distributed by radio. They are thus a single timebase distributed at the speed of light. Even at the highest levels of tolerance, it is not possible for a site to receive a time signal, then send a message to a second site which arrives before the time signal; two sides of a triangle are at least as long as the third. Similar accurate distributed timebases exist in other countries, and they are kept mutually synchronised to astronomic time by the most sophisticated techniques available. Two sites receiving such a radio timebase will be synchronised to a tolerance less than the time it takes them to inter-communicate.

Using all the above techniques we can distribute time with acceptable error. For two sites on the same local area network the error is limited to the unpredicatability of the transit time of a message, the difference between the fastest possible transit time and the slowest possible transit time. A weaker condition, since it is not a guarantee, is that the actual error will depend on the actual maximum transit time less the minimum possible; this is likely to be considerably better than the worst possible case. Due to the delay between timestamping a new version of a file, writing it out to disc and sending the minimum number of messages to detect an external inconsistency, this is probably acceptable for the current types of network in use. Sites on different networks connected by a slower long-haul network can maintain this global view of time using a standard distributed timebase such as the VHF radio time signal, or by using mains frequency after an initial accurate setting. This initial setting can be effected by two means. A clock can be taken to one site and there synchronised to it; it is then taken to the other site and used to set the clock. This is analogous to setting your watch to the office clock, and then using it to set your kitchen clock at home. The other means is to use one of the widely available distributed timebases, such as the speaking clock or the pips broadcast on some radio channels on the hour. Using these means we can distribute time globally to some fraction of a second, and locally to a better tolerance than this.

## Locking

If we are to use timestamps to identify different versions of files, we are forced to be able to hold off competing concurrent access. Although it can be disguised in a number of ways, we have to be able to lock files to ensure exclusive access to a single client when necessary. Ideally, we would like to be able to lock arbitrary amounts of data within a file but this may prove to be impractical. Most updates to a file depend on more than just the contents of the particular version of the file read; they also depend on the fact that there is no later version in existence. Since an update normally takes a file from one version to the next, it is essential that any data read from the file to create the new version comes from the current version. In effect, the data read to create the new version must still be accurate at the moment the new version is instantiated. Further, noone must be allowed to see the new version before it is instantiated since it may be subsequently altered further, and worse, it may not even be instantiated.

There are cases when an old version of a file is acceptable or even necessary. In these cases no locking will be necessary. It is not possible to change a file version or its timestamp, merely to alter how many subsequent versions exist. For example, we may want to copy all of our files as at midnight to a backup medium. If some of them are active it does not affect the data which we want to read.

In this context, an old version may also be the current version. Not locking it merely means that we do not insist that it remains the current version. For example, if we compile a program we probably do not need to lock the executable code of the compiler against alteration while the compilation proceeds. It is hard to think of cases where it would be important that the version of the compiler in existence at the time that the object code of the compilation was instantiated was the one used for the compilation.

From now on we will consider locking whole files. There are strong arguments for large granularity of locking [RIE77] but they depend heavily on the locks being placed accurately [RIE79]. For a filing system, it is rare for whole file locking to be unacceptable. The exception is where a database is maintained on the filing system. In such a case it is unlikely that a locking scheme based entirely on the physical storage layout of the database is acceptable, and there are good arguments [HOA74] for not using the low-level scheduling provided by data locks to implement high-level scheduling decisions, such as who should get the data next. For this case, a database lock manager (perhaps itself distributed) should lock the file containing the database, and then implement whatever locking policy is desired. Alternatively, the database can be suitably divided amongst several files so that whole file locking remains appropriate.

One feature which arises in any locking scheme is that of phantom records [ESW76]. A phantom record is a record which would have been locked had it existed. The information locked is the non-existence of the record. For example, consider maintaining a phone directory under a record locking scheme. Two competing people may see if John Smith is in the directory, discover that he is not (no records locked so far), each create a new record (independently locked) and add their new record to the directory. Since neither of the newly created records is examined by the other person, no locking conflict occurs and Smith is added twice. In this simple case it is likely that the conflict will be discovered when the alphabetic index is updated, but it is easy to invent complex examples. We are concerned only with locking files, so phantom records will manifest themselves as phantom files. The problem is simple in many cases, since what we test is not the existence of the file but the presence of the textname in some directory. If directories are files then the directory will be locked while this happens, and only one of the competing creators will get write access to the directory (we ignore the deadlock possibility for now). If files have a low-level name independent

of any textnaming scheme, then there is no problem if the server chooses the names for created files (they are more likely to be large integers than textnames). In this case, it is not sensible to ask whether a given filename exists since either the name was a lucky guess and the contents of the file will be of no interest, or it was discovered somewhere and so must already exist (and so can be locked). At the other end of life, a file may have been deleted. This is no problem since the access allowed to the latest version of a deleted file will forever be the same as the access permitted to a non-existent file. It is inaccessible.

There are many locking schemes proposed [BER79] but most are inappropriate in our case, either requiring advance knowledge of all locks that will be claimed, or failing one client in the event of a conflict. One locking scheme which is provably [ESW76] sufficient is a two phase one. This will ensure that a client cannot see an inconsistent view of the data due to a competing client. A client will, of course, be able to see an inconsistent view of the data due to itself as it takes the data from one consistent state to another through inconsistent states. In the first phase, all the locks are acquired. In the second phase, they are all released. No lock may be released in the first phase and no new locks acquired in the second phase. This is a slightly stronger condition than necessary since some data read may have no effect on the data written. For example, a program might search a file for any all-zero page in which to insert some new data; the contents of all the non-zero pages examined in the search do not affect the data written. Under these circumstances it is safe to release the read locks and allow others to update the data read. However, knowing that this is the case depends on the contents of the files and not on the operations performed. It is thus not something which the filing system can detect and so we ignore it. We merely note in passing that we might permit clients to unlock data which they detect that they read in error. Consistency will depend on their never unlocking data which they should not.

The first phase, the growing phase, is the phase in which the client performs all the reads and writes necessary to evaluate the contents for the new versions of the files to be updated. At the end of the first phase, all the files read are read locked (under the usual multiple reader and no writers discipline) and all the files written are exclusively locked. At this point, the new versions are complete but not yet instantiated. The second phase begins. All the read locks can safely be released. Each new version is instantiated and then all the write locks can safely be released. Under this scheme, no interference between competing clients is possible. Note that the second phase is entirely under the control of the servers comprising the filing system, once it has been initiated by the client.

We will formalise the loose concept of the files touched by a client into a transaction. A transaction serves mainly to name the set of files read and updated by a client. Later, it will be integrated with crash recovery techniqes to form an atomic transaction which either succeeds in its entirety or has no effect at all. A client starts by issuing a create transaction request to the filing system. All the reads and writes of files during the transaction cause the file to be locked, and marked as owned by the transaction if they are updated. Finally, at the end of the growing phase, the client issues a commit transaction request to the filing system. The filing system then autonomously instantiates the updates and releases all the locks. The transaction is then finished. For convenience, we also allow the client to issue an abort transaction request, which is an alternative second phase in which all the updates are discarded and the locks released. Under certain circumstances, such as deadlock, the filing system will take it upon itself to abort a transaction, exactly as if an abort request had been received.

As in any locking scheme, deadlock must be considered. Indeed, one such situation has already been hinted at. Two clients read a file directory to see if a given textname exists; on finding that

it does not they both attempt to write the new name but neither can have exclusive access to perform the write since the other holds a read lock. As with any deadlock, there are a number of different approaches to take to handling it. Firstly it may be detected in advance, often called deadlock avoidance. This requires examining the graph of transactions and locks held or wanted for cycles. This is most easily done by calculating the transitive closure of the matrix of lock interrelations. When more than one server in the filing system is holding files locked by a given transaction, this is unattractive since the matrix has to be collected from the various servers before the calculation can be done. The second approach is to detect that deadlock has occurred. This can be done when the situation looks suspicious and there are a lot of blocked transactions all waiting for locks. This suffers from the same problem that the matrix needs to be collected. Since it only happens occasionally we may be prepared to accept the overhead. Unfortunately the most common deadlock in a distributed system arises when clients crash. This is not detectable by examining the lock graph, since no cycle will be present (in effect, the dead client may only be sleeping and on the point of waking to release the lock). Since, by definition, crashed clients are not in a position to announce that they have crashed, this has to be detected by other means. Some network protocols have an active idle state and detect a site crashing when it ceases sending anything at all. Most local area networks do not have any such state, and in any case it only detects those crashes which inhibit the lowest level network software. In any event, a network filing system has to be able to cope with a client undetectably crashing. If the crashing client holds any locks, then they will, at best, be released when the client recovers; at worst, they will never be explicitly released.

The only means for handling this case is that of timeouts. This is rather unsatisfactory since it is difficult to predict in advance what is a sensible timeout to use. A program may write a heading to a file and then perform on-line monitoring awaiting an

event hours hence before writing the next line. For files which it is intended should be shared, it is almost certainly bad practise to hold a lock for a long time. The locks should be used as a low-level scheduling mechanism, perhaps to implement a higher-level locking scheme if necessary. Apart from the small resource used in the server to maintain the lock, locking a sharable file is only a problem when an attempt is made to share it. The timeout mechanism handles, if somewhat crudely, the other forms of deadlock caused by a cycle occurring in the graph of locks held or desired. This cycle can only appear when attempting to place a lock on an already locked file whilst already holding a lock on a file on which someone is waiting. This is not a necessary condition, merely a sufficient one. All this points to a two level timeout. If a lock has been held for an excessively long time then break it. Otherwise, only break the lock if it has been held for a reasonable time and a conflict occurs; someone attempts to set a competing lock. The easiest way to break a lock is to abort the transaction(s) which own it. There are schemes [STU80] for notifying clients of broken locks rather than aborting the transaction, and there are good reasons for doing so. For example, consider a program attempting to total all the accounts in a bank during the working day; it has little chance of not being aborted before it has finished. Many of the problems of always aborting are obviated by allowing old versions of files to be read. If the bank accounts are changing then the total is only momentarily valid so why not calculate the total as valid at the time the program commences rather than completes. This requires no locks since historic states can never change, even if they also happen to be the current state of the files.

Atomic transactions

So far, we have introduced transactions as a means of associating locks on files. This restricts the access to files in a way which means that it is hard for clients to interact undesirably. The outcome of a transaction is that it commits successfully or aborts completely. We would like this to be so even in the face of failures of components of the filing system. We call this an atomic transaction.

We are thus also going to use transactions to control the granularity of the operations which are seen as atomic by the client. It is not necessary that the unit linking file locking and unlocking, and the unit of atomicity be the same.

One way round, we may perform an atomic checkpoint operation without releasing all the locks. No competing client can gain access although if a subsequent crash occurs, intermediate new versions of the changed files will have been instantiated. For this continuity of locking to be useful, it has to be guaranteed; for it to be guaranteed it needs to hold even in the face of crashes. This means that locks must be preserved on disc in a manner such that they can be reset during crash recovery. There is clearly a risk of creating locks which are not going to be released since they have been forgotten, and cannot lightly be timed out since the original program which set them may not be run until some time after the return of the server. There are applications in databases which may benefit from such a checkpoint scheme. It is hard to think of situations in which it would be particulary useful in a filing system.

The other way round, we may have a number of transactions all regarded as a single atomic action. Since the atomic action may never eventually occur, the locks cannot really be released but merely seem to be to the owning transaction which completes normally. This is the useful concept of nested atomic

transactions. It can, however, be implemented wholly within the client. Nested transactions are useful since they enable parts of programs to be written with no knowledge of the transaction environment in which they will run.

We shall only consider atomic transactions where the unit of locking and the unit of atomicity are the same. Nested transactions are useful but not primitive; long term locks are not useful enough in a filing system to justify their complications. If we use atomic transactions in this way, then we have a very strong guarantee:

When a transaction completes it either has:

* no effect

or

* the new versions of all the files are simultaneously (they have the same timestamp) instantiated; all the data read to create these new versions is unchanged at the moment of instantiation (no new version of any of them is created with a timestamp between the version read and the timestamp on the newly instantiated files).

This guarantee holds even if the files are dispersed among several servers in the filing system. It also holds in the face of crashes of clients, servers in the filing system, the network. If we have designed the system to preserve suitable backup duplicate copies, it holds even in the face of data loss due to such failures as a disc drive head crash.

This amounts to a statement that external level two consistency can be guaranteed after a crash, and that it will be impossible to detect a level two inconsistency while the system is running normally. The statement does depend on a client couching all desired consistency in terms of single transactions. If a client has more than one transaction then they are regarded as independent. Apart from being a recipe for deadlock, there is no

guarantee, for example, that data read in a transaction which commits early will be still be valid in a transaction which commits later; the second transaction has to read the data again (although one can devise mechanisms to obviate unnecessarily transmitting large amounts of unchanged data).

Commitment protocols

Achieving this guarantee when files are dispersed among several servers and in the face of crashes is not trivial. There are two parts to achieving it. Firstly, there is a protocol by which the servers communicate to complete the transaction. Secondly, there is the protocol by which a server recovering after a crash discovers what to do about new versions of files which have not yet been instantiated, together with ways of storing these new versions to ensure that they are rediscovered. There may be a third protocol which the remaining servers in a transaction use to clean up when it is detected that one or more of the servers has crashed.

The protocol used by the cooperating servers to complete a transaction is called the commitment protocol. There are several desirable features about any commitment protocol. It is part of the normal running of the filing system, and so we want it to involve as little work as possible, provided that we can still guarantee external level two consistency following a crash. Since crashes are, hopefully, rare we would like to have normal running simple at the expense of a more expensive crash recovery. That is, we would like fast commitment even if we have to tradeoff this for a slow recovery after a crash.

A minimal protocol would be for the server receiving the commit request from the client to send commit messages to all the involved servers. This is clearly not good enough since some of the servers may instantiate their versions whilst other servers may crash before doing so. We need an extra round of messages to ensure that

the commit is possible [SKE80,GRA77,LAM79].

A typical two round commit protocol proceeds as follows. The server receiving the commit request from the client becomes the master server in the protocol. It ensures that all new versions of files that it holds are ready for instantiation and marks them as such. It sends messages to all the other slave servers to tell them to do the same. If any server crashes or otherwise fails to successfully ensure that all new versions are on disc ready for instantiation, then the transaction aborts and the master server tells all the servers to destroy the new uninstantiated versions. Otherwise, the master server commits the transaction by instantiating the new versions of the files which it holds (we assume, for now, that this can be done in an atomic way). From now on, the transaction is destined to commit irrespective of any crashes. The master server tells all the slave servers to instantiate the versions which they hold. When they have all reported that they have done so back to the master server, then the transaction has successfully committed. The master confirms that the transaction committed to the client.

Recovering from a crash is handled as follows. If a slave server crashes then it will discover the ensured new versions and enquire of the master server whether to commit the transaction and instantiate them, or whether to abort the transaction and destroy them. Any versions which have not reached the ensured stage are always destroyed and the containing transaction aborted. If the master server crashes, then on its recovery it sends all the commit messages again. These may be rejected since they may duplicate messages already sent and acted upon.

There is a problem if, for example, the master server crashes together with all the slaves bar one. The remaining slave cannot decide whether to commit or abort the transaction, since the master is not available and other slaves may have committed or aborted the transaction, precluding a unilateral decision. The transaction

cannot proceed and is blocked. A protocol in which this can happen is called a blocking protocol. Construction of a non-blocking protocol requires three rounds of messages [SKE80,SKE81]. Such a non-blocking protocol always lets the remaining servers commit or abort the transaction without reference to the crashed servers. When the crashed servers return, they will need to discover the decision made in their absence, so files can become locked here in pathological cases where all the participating servers have then crashed. However, a crashed server will never have to reverse a final decision to commit or abort. The first round is as before, with the master and slaves all ensuring that the new versions of all files are ready on disc for instantiation. The second round of messages is the extra round when the master tells all the slaves to enter an intermediate buffer state, prepare to commit (which must also be recorded on disc). In the third round of messages the master tells all the slaves to commit.

If the current master server should crash, then one of the slaves becomes the new master server. This requires the set of servers involved in the transaction to be distributed to all slaves at the start of the protocol. The new master server decides whether or not to commit the transaction only according to its own state. If it has not started the protocol, it is ensured or it is aborted, then it aborts the protocol. If it is in the intermediate buffer state or it has committed the transaction (locally) then it commits the transaction. If it is not already in the commit or abort state then it first sends messages to all other slave servers to set their state to its own. The protocol is such that this will always be an acceptable thing to do. This round of messages ensures that if the new master server should crash, then the server taking over as master server will make the same decision. Then it sends a round of messages telling the other slave servers its decision (commit or abort). There are still some circumstances under which transactions may block. The most obvious is that if the network should fail then transactions will be blocked until it returns. The other obvious case is if all the servers involved in

a transaction should crash. This is much more serious since single server transactions are likely to be the most common type of transaction. If this single server should crash then the transaction will block. Since we expect single server transactions and single server failures to be the most common of each class, it is doubtful whether the extra overhead of the extra round of messages and the extra complexity of implementation are justified.

These commit protocols are examples of centralised commitment protocols in which one server coordinates the operations of the others. There are also distributed protocols in which there is no distinguised server. These are less attractive since they usually involve significantly more messages being transmitted and are more complex to implement. For the current range of matches between processor and network speeds, constructing and decoding messages from the network uses a significant fraction of the available processor bandwidth. These protocols are thus expensive. However, the guarantees which they back up are strong enough to make them worthwhile, especially if they are infrequently invoked.

Most distributed filing systems exhibit considerable locality since most users place all their files, or all their files concerned with a particular area of their interest, on a single server. The arguments for using multiple file servers on the same local network are very weak unless this is so, for they waste the time in cooperation that they save in parallelism. It is sensible if they have very different characteristics (fast-small and slow-large). However, there may be more than one network, inevitably more slowly accessed through some form of inter-network gateway and then multiple file servers are sensible. Overall though, most transactions will be single server and require no additional messages. This is, of course, to some extent a self-fulfilling prophecy since the overhead of multiple server commit protocols is a strong incentive to keep a high locality in file placement.

Single site atomicity

The above discussion about commitment protocols started from the assumption that the update of one or more files at a single server in the filing system was already an atomic operation. We have to consider how to achieve this. It is such a frequent operation that it needs to be efficient. Between requesting a commit and receiving confirmation, the client will normally be waiting. This waiting time should not be excessively long. Again, we are prepared to pay for a quick commit, which is common, by accepting a longer crash recovery procedure, which is rare.

This efficiency or otherwise of this single site commit is much more important than the multiple server case. As discussed above, most transactions will involve only a single server.

There are many different ways of achieving this [STO76,ISR78,BAR78,PAX79]. Most of these rely on some form of deferred update. The pages to be altered are recorded in some rediscoverable place, such as a fixed area of disc, in a journal file or in a file created for just this purpose. These changes to be made are called an intention. There are a number of important properties required of an intention. Firstly, it must be constructed in such a way that it will always be discovered during crash recovery, or when an attempt is first made to use a file taking part in the transaction. It must be possible to tell whether or not it is complete. It must be possible to tell whether or not it has been completely dealt with. After writing out the intention completely, changes are made to the files as specified in the intention. Finally, the intention is marked as dealt with (or destroyed). During crash recovery, intentions are rediscovered. If they are incomplete then they are destroyed. Transactions which have only reached this point are aborted and backed out. If they have been dealt with, then they are ignored. If they are complete but have not been completely dealt with then they are carried out; the changes are made (possibly for a second time). Care needs to

be taken to make the changes idempotent so that they can be performed completely or partly many times. The intentions should be dealt with during crash recovery in a manner which is tolerant of further crashes occurring during crash recovery.

This form of deferred update can be very expensive. One optimisation is to store the data pages to be altered in unused disc pages and then the intention consists only of changes to file map structures. Since many changes may be made to the same file, this may involve considerably less duplicate writing than putting the actual data to be changed into the intention.

Another scheme for commitment uses the idea of a difference file [SEV76,BAR78]. Here, most of the data is kept unaltered in a read-only shrine and a writeable file of errata is maintained. This scheme seems to work well for updating a large record store, where the alterations are few and small. It is not clear how to sensibly extend the ideas to a filing system without being able to label disc pages. Most disc controllers do not directly support this labelling. To support it indirectly by using some of each sector to perform the labelling results in a page size which is inconvenient to clients.

If multiple versions of files are created on one side, then we are only concerned with how they are instantiated. One way to do this is to make the versions self-defining and regard any atlas used to locate them as a hint [LAM74]. Since we are preserving both the old and the new versions of files when they are instantiated, we cannot update files in place. One simple approach is to have each file defined by a file header, a single page containing nothing but the management data for that file. It could also contain some data from the file, especially if the entire file is small enough to fit into the rest of the header. If disc pages can be labelled, then any pages can be used for such headers; otherwise, a special fixed area of disc will need to be allocated to hold them. In either case, the existing file headers can be

discovered as part of crash recovery. As an optimisation, if a disc page is profligately larger than a file header we can group headers for new versions for the same transaction in the same page. This has little effect on the algorithm but may save some disc space.

An atomic update of multiple files then proceeds as follows. The idea is to make the instantiation of all the new versions dependent on the instantiation of the new version of one distinguished file, and to perform this instantiation in a single disc write to an otherwise unused disc page. This last write is an atomic action in that it succeeds completely or fails completely (the page is unreadable or unchanged).

One of the items of data held in the header is the state of the file, the point the owning transaction has reached. When a new version of a file is to be created, a file header with state "unused" is found (probably from a list of such headers). If it is necessary to write this header out to disc before commitment of the transaction begins, then it is written with its state set to "volatile".

When a commit request is received, one file being updated is selected and becomes the master file for the transaction; say it is file number N. If no files are updated then the transaction is read-only and commitment only involves releasing locks. The timestamp for the commitment of the transaction is taken from the local clock; say it gives a time T. At this point, all locks on read-only files can be released.

If there are new versions of other files to be instantiated, then they have their headers set to "dependent on N", and their timestamps set to T. A flushing step takes place during which it is ensured that these dependent headers and any other pages from any of the files (including the master file) concerned in the transaction are flushed out to the disc. At this point, every page

altered in the transaction except one is up to date on the disc. The one page is the header for the master file, which, if it is on disc at all is not necessarily up to date and is marked "volatile".

Finally, the master file header has its state set to "committed" and its timestamp set to T and it is flushed out to the disc. All locks held by the transaction are released. The transaction is committed the moment that this last page is successfully (readably) written to disc. Table 3 summarises a normal multiple server commit.

Table 3: Commital of transaction X

| MASTER S | SLAVE i |
|---|---|
| Receive commit X from client | |
| Release shared locks owned by X | |
| Select global master file G | |
| Get timestamp T from local clock | |
| Send ensure XGT to all slaves | -> Releases shared locks |
| Flush non-header pages | Select local master Li |
| Mark all files other than G | Flush non-header pages |
| as dependent on G,T and flush | Mark all files other than Li as dependent on Li,T and flush |
| | Mark Li as dependent on G,T and flush |
| Receive all confirmations | <- Confirm to master |
| Mark G as committing and flush | |
| - - - - - transaction now irrevocably committed - - - - - | |
| Send confirmation to client | |
| Send commit X to slaves | -> Mark Li as committed and flush |
| | Release exclusive locks |
| Receive confirmations | <- Confirm to master |
| Mark G as committed and flush | |
| Release exclusive locks | |

If a transaction is aborted, then the disc pages can be returned for subsequent reallocation. Headers which are "volatile" can be marked as "unused", but it causes no problems if this is not done.

If a crash should occur, then the recovery procedure discovers all the headers of all the files on the disc.

Any file with state "committed" is a genuine version of that file; all older versions with states other than "volatile" are previous versions of the file.

Any file with state marked as "dependent on N" with timestamp T is committed if a version of file N exists with a timestamp of T or later; all older versions with states other than "volatile" are the previous versions of the file.

Any file with state "dependent on N" with timestamp T and no extant version of file N with a timestamp of T or later is a complete version of a file in an incomplete transaction; its header is reset to state "unused" and returned for reallocation.

Any file with state "volatile" is an incomplete version of a file; its header is reset to state "unused" and returned for reallocation.

As described, this algorithm requires a sort of lookahead, since it is possible to discover a file before its master file has been discovered. The decision about whether the version is genuine or not must be delayed. In chapter 9, details of an algorithm which actually converges properly on the correct state of the disc will be presented.

Having decided which versions of files are genuine and which are to be discarded, a new atlas of the headers can be constructed. For a server with hardware paging this could well be constructed in store since it is regarded as volatile and lost on each crash. Finally, the free disc page list can be constructed by following each file page map to discover all the allocated pages; all the other pages are free. This reconstruction gives us another advantage. The atlas and the free page lists do not have to be accurately maintained on disc during normal running. They do, of course, need to be accurately maintained in store. Since files tend to be intermittently active, the freedom for the disc copy of

the atlas to be out of date is an advantage. Only when a file ceases to be active and the in-store tables are full do the new locations of the current headers need to be flushed to the atlas.

If a file header contains only the roots of the file map pages, the timestamp, state and size of the file, and some user attributes, then most of the data for the new header will be either unchanged from the previous version or else held in store while the version is being created. In this case, the new header will rarely be created until the transaction commits, and so there will be no writes to disc which would not take place to create the versions in any case. There is an overhead in maintaining the multiple versions. Updating a large file requires index pages to be written. Only for a large file updated in place will this be an extra overhead. The algorithm thus appears acceptably efficient.

However, crash recovery is slow to pay for this. Let us assume a 300Mb disc of current speed (3600rpm, 8ms track to track) with 512 byte sectors. We might reasonably have 50,000 file headers. If there is a shortage of main store, then we may have to rescan some of them; 10,000 is a pessimistic estimate. Perhaps two percent of the rest of the disc will be taken up with file map pages. For a full disc, about 10,000 pages scattered around the disc. If the file headers are contiguous then the 60,000 headers can be read at about one every 20ms (not much seek time) and the 10,000 scattered file map pages at about one every 50ms. This gives a rebuild time of 1700 seconds - half an hour.

Extending to multiple servers

The above atomic instantiation extends to multiple servers, using the two phase commit algorithm described earlier. The idea is the same, to make the commitment of the entire transaction depend on the commitment of a single file, which is a single disc write to an otherwise unused disc page, itself an atomic action. However, now the file is not necessarily held locally.

The algorithm also depends on one server, the master server, knowing which other servers are involved. This is most easily achieved either by making a transaction number contain its originating site, so that a server receiving it for the first time can make itself known to the master server, or else by insisting that the client explicitly issue an instruction to add the server to the transaction. For simplicity, the commit request is only accepted at the master server, the server which first started the transaction.

On receiving the commit request, the master server selects a file which has been updated there. If there is none, as would happen if files were only read at the master server but written on other servers, then either it creates a file header specially, or it passes a server which does have an updated file the list of involved servers. If there is no such server then the transaction is read-only and there is no problem about consistency. Let the number of the selected file be N, and the master server site number be S. We call the file N the global master file. It may well be possible to deduce which server holds a file from its file number; in particular, to deduce S from N. This slightly reduces the amount of data to be dealt with in the commitment. The master server reads its local clock to get a timestamp for the commitment; let it be T.

The master server then tells all the other servers to flush the files in the transaction with master file N on server S at time T. Each slave server selects a local master file (if it has any updated files) and performs the flushing procedure described in the single site case, so that everything except the header for the local master file is flushed out to disc. It can also release all locks on read-only files. Finally, the local master file header is set to state "dependent on N on S at T", and a confirmatory message sent to the master server. Meanwhile, the master server has performed the flushing procedure marking all files except the global master file as dependent on it, but not flushing the header for the global master file.

When all the confirmatory messages have been received, the files are all in a commitable state. All but one file on each server are marked as dependent on that one; for all but one server these local master files are marked as dependent on a global master file. The global master file is still "volatile".

Now the transaction actually commits. The global master file header has its state set to "committing" and it is written to disc (to a new page). At this point, confirmation can be sent to the client since the transaction will commit whatever happens. However, it may be better to leave this until completion of the protocol. The master server can now release all exclusive locks owned by the transaction. Another round of messages ensues when the master server tells each slave server to commit. Each slave server sets its local master file to "committed" and writes it out (to a new page). This step is logically unnecessary, but serves to decouple the servers; it is now no longer necessary to send a message to the master server to find out whether a transaction has committed. It then releases all exclusive locks; if local time at the slave has not reached T, then this releasing should be delayed until then to avoid inconsistent views of time being seen. A confirmatory message is sent to the master server.

When the master server has received all the confirmatory messages then it sets the state of the global master file to "committed" and rewrites it (to a new page). Again, this step is unnecessary but serves to decouple the servers.

If the transaction is to be aborted, then the master server sends messages to all slaves to tell them this. Any headers which have got past the "volatile" state are reset to "unused" and written back to their current location. This destroys the "dependent" header; if the write is interrupted or leaves the page bad then there is no problem. Crash recovery will continue the destruction of the new version. The new versions can be destroyed and any pages used returned to free page lists. The slaves can release any locks held on files participating in the transaction. Each slave confirms that the aborting is complete to the master server.

If a crash occurs, then there are several cases to consider. If a slave crashes before the first confirmation has been sent to the master, then the master will eventually give up waiting for the confirmation and abort the transaction.

If a slave crashes after this confirmation, then the transaction may commit or abort. As part of crash recovery, the slave will discover the file dependent on a global master file N on server S at time T and enquire whether such a file committed. It can then instantiate or destroy the new version, and any other new versions dependent, in turn, on it.

If the master crashes, then the slaves must await its return. This is the weakness of centralised commitment protocols, they are very vulnerable to failure of the master server. If the slaves detect that the master has crashed then there are some files about which nothing can be decided. These are files which form part of a transaction for which the first confirmatory message has not been sent. They can be aborted locally since the transaction is certain to abort anyway.

The only complex case is that of a file dependent on a global master file held at another server. When the other server returns, the slaves must enquire whether or not the transaction with global master file N at time T committed. If the master server has a version of file N timestamped at T (it will have state "committed" or "committing") then the transaction committed; otherwise it aborted. Mark N as "committed T" and flush.

## Space reclamation

If we maintain all versions of all files then the storage requirement for the filing system will grow with time. There will come a time when the storage requirement for the filing system approaches the storage available. Ideally, we could drain old versions of files to a cheaper medium, with a slower access time, such as offline disc, magnetic tape or a mass-storage unit. For a write-once device such as a video disc, this will require the live data to be copied to a new disc and the old disc remains, preserving the old states as long as it is kept. It would still be possible to enquire about the past history of a file, it would just take longer. If this is impractical then some old file versions can be destroyed. To distinguish this from deletion of a file, which does not destroy all the old versions but merely creates a new version consisting solely of a tombstone, this process is called digestion. In some ways, this is transference to the medium with the slowest access time and the cheapest storage cost of all.

There are a number of ways of deciding which old versions to digest or drain to a cheaper medium. The simplest is to do this simply by age - the oldest versions are discarded first. This is the easiest to implement since an obsolete version can simply be added to the end of the queue for digestion when it is made

obsolete by a new version being created. However, it does assume that the old versions of files are equally important, an assumption which is unlikely to be valid. One other problem associated with making the decision automatically is that, in the absence of any other constraints, a user writing a large file (perhaps in error) can cause digestion of all the old versions held at the server.

The alternative is to give clients control of the choice. The most flexible way of doing this is to allow clients to specify for any file either one or both of two constraints. Firstly, they can specify the minimum number of versions to be retained. Secondly, they can specify an age; files must certainly be preserved until they have been in existence this long. For such files as source files of programs, three versions are probably better than all the versions in the last hour. A mistake made last thing at night may be irretrievable by morning. For files which really do have a time component associated with them, such as the contents of a stock-room, all the versions in the last two hours are probably better, since they permit a time-consuming transaction to perform some form of stock-taking while the stock-room is active. For some files which are maintained automatically by a program, explicit control may be useful. There is little point in maintaining old versions of file textname directories if the only difference between the old versions and more recent ones is the inclusion of the text-names of digested files.

There is a subtle interaction between retention of files and the multiple server commitment algorithm described above. It is essential that no global master file version be drained or digested until its transaction has completed since whether the transaction committed or aborted is deduced from the presence or absence of a version with the correct timestamp. There is only one point at which this is likely to be a problem. If the master server commits a transaction but crashes before sending the instruction to commit to a slave, and that slave is crashed when the master returns. When the slave eventually returns, it is going to ask the master

whether or not the transaction committed. If it has been digested the master will reply that it was aborted when, in fact, it was committed. This is simple to detect since such a master file will be in a state of "committing" rather than "committed".

## Usability

The primitives provided by this filing system differ from those provided by most filing systems. It is necessary to investigate how easily these primitives can be used in practise.

Most application programming languages support the notion of a file in an abstract way. Programs which have been written should work using the primitives provided by the filing system we have described.

The simplest way to do this is to arrange that the command interpreter start a transaction when it runs a program, and that any files accessed by the program are part of the transaction. When the program completes successfully, the command interpreter commits the transaction. Running a program is thus an atomic action; either all its output files are completely updated or none of them are. If the program should fail in some way, the user can have the option of committing the transaction, aborting the transaction or, possibly, investigating the state of the new versions before making a decision.

More sophisticated programs may wish to do their own transaction management, and these will not be runnable on other filing systems. This is normal in a filing system; programs which make use of specialised features of filing systems are not portable.

## Summary

This section has started from the notion of consistency and defined it more rigorously. Existing filing systems have been criticised for failing to provide sufficient guarantees, forcing users into an unnatural style of programming. The notion of multiple versions of a file has been refined into the notion of a file with a history accessible through its time component. The distinction between ordinary files and redundant files has been introduced, partly as a genuine distinction but partly as an efficiency measure for those files for which old versions will never be required. A method of implementing atomic transactions has been shown to be feasible, at least in outline, using seemingly simultaneous instantiation of new versions of files, perhaps held at more than one server in the filing system. An efficient implementation has been outlined. This implementation is considered in detail in chapter 9.

Making sensible use of local discs within our constraints is very difficult. Using a local disc for some file accesses clearly reduces the demand for network and server processor bandwidth. It also increases the autonomy of the client machines, perhaps to the stage where they can operate for long periods independently of the network filing system.

It is worth reiterating that the term local disc is used in a general way to imply local large volume storage, almost certainly non-volatile. This includes all media such as floppy discs, small Winchester discs, large high-performance magnetic discs, laser optical discs, bubble memories. In some ways, it also includes the main memory of the client machine if it is sufficiently large (and the client operating system suitably organised) that an in-store cache could hold an appreciable amount of data. The important characteristic is that a local disc is a large sized cache.

In the past, local discs have been used as the primary filing system medium, perhaps with network file servers also accessible but not integrated into the client filing system. Mobility of users is achieved by using removable media; users remove their own personal discs and take the discs with them. Sharing of files is done using an unsophisticated file server as an intermediate store, with special programs to transfer a file between file server and local disc. This is unattractive for several reasons.

Firstly, carrying disc packs around is inconvenient for all but the smallest types of disc. In any case, Winchester technology discs tend to be non-removable, especially the cheaper designs. This sort of scheme only really becomes acceptable if users rarely move between client computers. In turn, this only happens when all the client machines are the same and all users have their own. In other cases, mobility is desirable.

Secondly, the sharing only works well for files which are never, or almost never, altered. Otherwise there is the perennial problem of keeping track of which version is current, which are old out-of-date versions, which are up-to-date backup copies and so on. When the textnaming scheme at the file server and at the client are different, this problem is even worse. Because the filing system at the file server is not integrated into the local filing sytem at the client, there is no help from the system in keeping track of files. Further, only because most active files reside on a particular user's local disc (inaccessible to everyone else) do consistency problems not arise with two users taking a copy of a file from the file server, updating it and flushing the resulting file back to the file server.

As discs become more unwieldy, the alternative approach is better. Instead of using the local disc for the primary file system and tacking the network on the side, the network filing system is used as the primary filing system and the local disc acts as a cache. This has an immediate advantage to the client since a simple client operating system easily inherits all the power and guarantees of the network filing system. Since the client filing system is not trusted by the network filing system, great care is needed in using this cache if any performance gain is going to accrue. There are two clear-cut cases.

The first is to cache read-only files, such as compiler and system executable code files, dictionaries and so on. Of course, occasionally a compiler is updated. However, this is not an update to the executable code file but rather a change in which file is used by default. The old release is usually kept around somewhere since compiler updates are regularly less invisible than users might wish. Releasing a new compiler is thus binding a new internal filename to the old textname rather than updating the file to produce a new version. Since read-only files cannot be altered, there are no consistency worries about whether the network copy is in step with the local copy.

The second clear-cut case for using local discs is for unshared temporary writeable files, such as compiler workfiles or outswapped virtual memory segments. Because these files are not shareable, there is little need for security or consistency control, and often no need even for a textname. This is especially true of outswapped virtual memory which may well be handled by a special mechanism outside of any filing system. A particular case of temporary data is the data accessed in a transaction. The locking strategy ensures that a cached copy of the data read will remain accurate. Data updated by the transaction will need to be written through to the network filing system before it completes, but intermediate alterations need not be.. Once a transaction has committed and a new one started, the timestamps of cached files will need to be checked to ensure that they are still up to date.

There is almost certainly a requirement for local discs to be fairly cheap, since they are very numerous. The prime candidates are therefore floppy discs or small Winchester discs of about ten megabytes.

Unfortunately, most of the cheap local discs have just the wrong characteristics for this. There is a large volume of read-only files and so we want the local discs to be large; they tend to be small. The temporary files, almost by definition, can make extensive use of high bandwidth from client machine to its local disc; they tend to be slow. Experience with WFS and with our early filestore demonstrates that access to a local disc is often slower than access to a file server over a network, since the file server usually has faster discs and, perhaps, a significant amount of caching to increase performance. If the local disc is a floppy disc then the network file server is faster by an order of magnitude.

The characteristics required of a local disc for holding temporary files is that it should be fast. A ten megabyte fast fixed head disc is likely to increase performance of a client

system much more than a much larger slow disc. An extra megabyte of memory might increase performance even more so. Already, the cost of dynamic memory chips has fallen to a level where a megabyte of memory is of comparable cost to a ten megabyte Winchester disc drive.

The characteristic required of a local disc for holding read-only files is that it should be large. Read-only optical disc transports are very cheap, largely due to their development for the domestic television market. Three or four hundred pounds for an optical disc transport is cheaper than even the pack for an eighty megabyte drive. A single optical disc, costing only a few pounds, can hold about 3500 megabytes of data [LAU80]. This is more than sufficient to hold all the read-only files of even a large organisation. In many cases it will be sufficient to hold a snapshot of all the files held on nearby file servers of the network filing system. The actual discs are sufficiently cheap that they could be replaced every few months with an up-to-date issue, although the cost of writing large numbers of nearly full discs is unclear.

Using a local read-only optical disc like this creates a rather inverted filing system where the network file server acts as a cache for the active files, whilst the inactive files are all held locally. Nevertheless, a lot of file accesses are to rarely changed files and so will be local. Precisely how optical discs are best used, taking economic and performance issues into account, will need to wait until they are more widely available. Nevertheless, their enormous capacity and, to a lesser extent, the enormous bandwidth achieved by loading a disc, makes optical discs attractive as cheap local storage devices.

The client operating system mangement of read-only files is not too complex. The client operating system needs to maintain a table (probably a large hash-table partially held on disc) of which internal filenames correspond to locally held files and where on

the local disc they are to be found. Accessing a file then proceeds as normal by resolving the textname to get an internal filename. If this internal filename occurs in the table then the local copy is used; if not, then the network filing system will have to be used. Any read-only file can be added to the local disc (if it is writeable) and any read-only file on the local disc can be discarded. Of course, as with any cache, making wrong decisions on these points will lead to a loss of performance but not to incorrect operation.

If the client operating system can be trusted, for example, if it is in read-only memory or is otherwise as secure as the file servers, then the client machine may be able to run the file server code as a protected process. It would need to provide external service to requests received from the network to meet our design criteria, and to be able to transmit requests out into the network. In practice, these guarantees are only likely for a conventional terminal system attached to the network, with a lot of high performance local discs. Almost all access from the terminal system will be to its own discs and so its performance and that of other computers on the network is likely to be improved by this technique. It is, however, an important precondition that the system be able to provide a suitably protected process environment in which the file server code can run without compromising security or consistency.

For an untrusted client operating system we are left with only being able to cache read-only files and temporary writeable files. We cannot keep the current version (from the point of view of the network filing system) on the local disc since there is no guarantee that it will ever become accessible from elsewhere or even that it will not be destroyed by an unreliable client system. One way of keeping more files locally is to keep file versions which are seen as current by the client system, but are seen as part of an incomplete transaction (and hence locked) by the servers of the network filing system. This approach is not without its

drawbacks. Because so much is cached locally, if the client or server should crash then it is unacceptable merely to abort the transaction and start again; an hour of work at a terminal could be lost. If the client should crash then it must be able to pick up the pieces of the transaction and restart from a checkpoint. If the server should crash, then without any persistent locks which hold over a crash, locked files will become unlocked and so liable to updating. To check this requires the client to compare all the timestamps of cached files with the current versions in the network filing system, relocking them as it reconstructs the transaction. It is not possible to avoid pushing the problem through to the user or to a running application program if this recovery phase should discover that the data has been updated in the meantime. This violates one of our overriding design criteria. More complex approaches, such as that taken by DFS [STU80], lead to a need for long term locks which, in turn, leads to a need for breaking locks under certain circumstances. Otherwise an errant client could lock a file for ever. It is similarly impossible to avoid this being pushed through to the application program. The chief problem with long term locks is this difficulty of breaking locks since it is difficult to distinguish between a client recovering successfully from a crash and requiring the locks to be held, and a client recovering unsuccessfully and requiring the locks to be broken without being able to enumerate them. The problem is further complicated if multiple clients, each with a local disc, are cooperating and sharing locks.

The way in which cached copies of writeable files can be held at untrusted clients remains an open problem. However, performance at a client can be improved fairly easily by adding a suitably fast or large local disc to hold temporary or read-only data.

Any system design has to demonstrate that it can be implemented. Further, when performance is a design criterion, it has to demonstrate that the desirable functionality has not been achieved at the expense of performance. This demonstration requires a particular implementation to be described in convincing detail. For the detail to be convincing, the contentious areas at least do actually have to be implemented. Only building a real system is a convincing demonstration that there are no hidden flaws or problems with the design.

The implementation to be described is intended to demonstrate this point, that the system is efficiently implementable. It is not a complete product; this would require several times as much resource to construct [BRO75] and have a large hardware requirement. The implementation is in terms of multiple processes on a terminal system using large files as virtual discs and the inter-process communication scheme as a network. Unreliablility can, however, be injected at suitable points to investigate the effect of crashes, network failure, loss of synchrony of clocks and so forth. As little dependence as possible has been made on the host operating system; the environment in which the file server code runs has no advantages which could not be provided by a tiny operating system in a dedicated processor. Each process on the host system runs code for either a file server or a client system, one host process corresponding to one processor in a real implementation.

The implementation is, however, reasonably realistic. The file servers (processes) cooperate through the network (inter-process message system) to provide a network filing system on discs (files). They offer concurrent service to multiple clients, handle crash recovery and have suitable caching to increase performance.

The code of a file server is unexceptional. It is written as a single Pascal program [JEN74] with some underlying assembly code procedures. The assembly code procedures implement the virtual machine in which the program runs. Since the program is actually running as a process on an outer host operating system, much of the virtual machine is concerned with recasting requests for the host system. For example, a request to read a disc page which the server would regard as going to the disc device handler is recast as an operating system call to read a page from a file. The program is concurrent within the process on the host operating system. Some of the assembly code procedures are thus concerned with controlling this concurrency, switching the stack of the Pascal program, handling external events and so on. The concurrency is managed by monitors [HOA74,LAM80] although the lack of compiler support means that entry and exit calls must be programmed explicitly. External events, such as pages arriving from the file or messages arriving at the process, are recast as monitor signals.

In a genuine implementation on a dedicated processor, these assembly code procedures would form a small multi-process operating system including device handlers. The functions which this operating system would need to provide are process switching (spawn a new process, enter a monitor, leave a monitor, wait for an event, signal an event), access to the disc (read a page, write a page), access to the network (await a message, send a message) and access to a clock (what is the time, wait for a prescribed time). Functions such as caching disc pages or scheduling transfers to the disc and the network are handled in the file server code itself.

The implementation splits into two parts. The textnaming is almost completely independent of everything else. This is to enable directories to be readable and to enable the textnaming to be handled by separate servers or even client systems. There is no direct correspondence between the textname servers and the file servers. If the processor bandwidth of the textname server were a

problem then multiple textname servers could be provided for a file server. If the processor bandwidth of the textname server were very underused and there were multiple file servers nearby, then the one textname server could handle the textnaming for them all. Some client systems could handle their own textnaming while others could use the textname servers. On the other hand, there is no assumption that the textname server must be in a separate processor. The file server and the textname server could be in the same processor.

## Representation of files

One of the key decisions in any filing system is how files are to be represented. The consistency scheme requires that individual pages in a file can be updated without requiring to rewrite the entire file. This means that some sort of indirection scheme with index pages is needed. To cope with large files a multilevel index is necessary; however, most files are small and so a multilevel index is not usually needed. It considerably simplifies implementation if the organisation of the indices does not depend on the number of levels required. For this reason a scheme based on that of version 7 of Unix [THO78] is used.

Disc addresses are 31 bits and disc pages are 512 bytes. Page zero on the disc is never one of the pages in a file and so a disc address of zero is conventionally used to represent an absent disc address. A file is handled using eleven disc addresses. The first eight are just the disc addresses of the first eight pages of the file; in most cases this will be sufficient for the whole file since most files are less than 4K long. For a file longer than 4K bytes the ninth address is the address of a page containing the addresses of the next 128 pages of the file. For files longer than

68K bytes the tenth address is the address of a page containing 128 addresses of index pages containing between them the addresses of the next 8192 pages of the file. If this is insufficient then the eleventh address contains the address of a triple indirection page. The maximum file size is thus about a gigabyte. Associated with each disc address in the scheme is a bit called the "moved" bit. A disc address together with its moved bit is thus conveniently 32 bits. The moved bit is used to indicate whether or not the page marked has been updated from the immediately preceding version of the file, and hence moved to a new place on the disc.

All the administrative data for the file is held in a file header page. Headers are kept in a special area of the disc and so there is no confusion between a file header and a data page from a file. The administrative data consists of the internal file number (64 bit integer), the eleven disc addresses already mentioned, the timestamp, the number of inbound textname references the file has from influential volumes and the type of file (redundant, normal or read-only). The internal filename contains the holding server in certain bits but read-only files have a server number of zero. The header also contains the commit state and the internal file number of the master file; these are used to manage commitment of atomic transactions. The header number of the previous version of a file is held in each header. This is a hint rather than reliable information - if the header turns out to contain a previous version of the same file then it contains the immediately preceding version. If it contains something else then the immediately preceding version has been digested. The rest of the page (which, in practice, will be largely unused for most files) contains pairs of keys and their associated rights of file access, and file attributes together with their associated values.

When a file is updated, a new header is always created. Most of the data it contains will, of course, remain unchanged from the previous version. Additionally, for a normal file, each updated page is also relocated. By this is meant that the data for a page

of the new version is never written to the same address as the old page. It does not mean that subsequent updates to the new page before the version is instantiated are written to new pages since a crash will cause the new version to be discarded in any case. This strategy is also used for updated index pages. The moved bit associated with each disc address indicates whether that disc address is a page in the previous version of the file (unset) or whether the page was updated from the previous version and so has been moved to a new site (set).

Ignoring caching, which delays creation of a file header for a new version, updating a file proceeds as follows. Similar approaches have been used within databases [CHA77,LOR77,CHA81] but with the simplification that old versions are not retained. A new header is allocated and a copy of the old header copied into it with the header number of the previous version suitably assigned and with the moved bits of the eleven disc addresses cleared. When a page is updated, a new page is allocated and filled with the new data. The disc address of the old page is altered to that of the new and the moved bit is set. This, in turn, might require altering an index page which is similarly modified by allocating a new page, updating the old disc address and setting the moved bit. This is repeated through the appropriate number of indirections until one of the eleven disc addresses in the header has been altered. If the page is multiply updated during the creation of a new version, detectable by the moved bit being set, then the page can be updated in situ without allocating further new pages. This is particularly important for index pages, which are often multiply upadated. When the update is complete, the old header still maps out the old version and the new header the new (see figure 2). The moved bits indicate which pages are part of the new version but not the old.

The moved bits are also used when discarding new versions rather than instantiating them, and when digesting old versions. To discard a new uninstantiated version, the eleven disc addresses and

associated index pages are scanned, and pages corresponding to addresses with the moved bit set are returned for subsequent reallocation. Finally, the header is returned for reallocation. For a large file with only a small update, this scan does not require the entire tree of index pages to be scanned since only a moved index page can possibly lead on to further moved pages.

Digesting an old version of a file requires both the old header and the new to be scanned. Whenever a new page was allocated to the new version then the corresponding page in the old version can be discarded. Note that this technique only works to discard the oldest existing version of the file. To discard an old version but leave older versions would require the moved bits from the discarded version to be propagated through to the succeeding version, a rather complex operation.

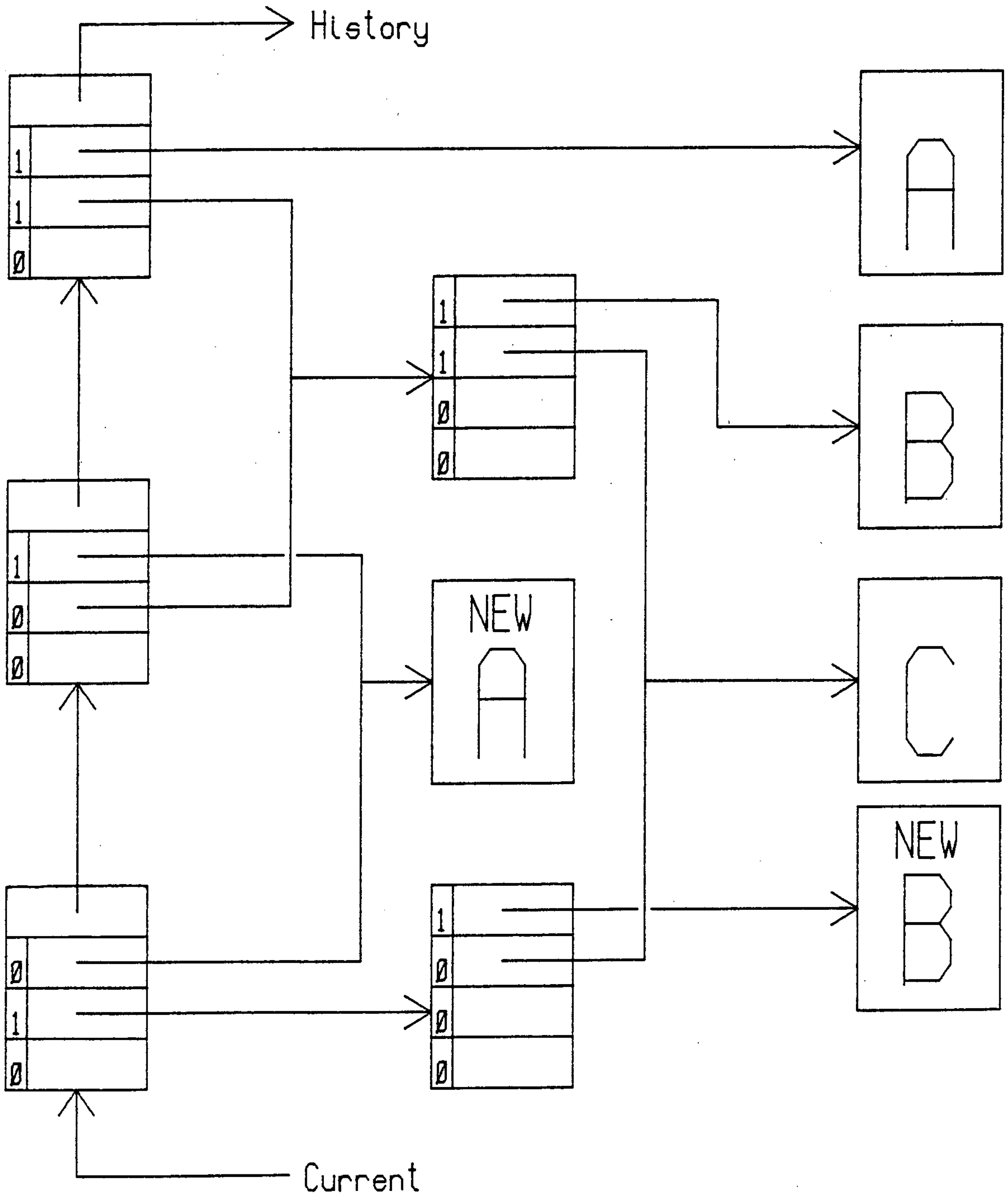
A file has a number of attributes. These are just pairs of 32 bit integers and are mostly uninterpreted by the file servers. Certain low numbered attributes are those maintained by the file server, such as the timestamp of the version or the file size. These, of course, may not be altered by clients.

The current version of a file is thus represented by a header. This header is located from the atlas, indexed by the 64 bit internal filename, although caching effects often cut out the need for this lookup. Previous versions of a file are found from the current version by following the chain from one version to the immediately preceding version. This assumes that old versions will be rarely accessed.

Figure 2 : Simplified diagram of three versions of a file

The number of disc addresses in each page has been reduced for clarity; most file header information is omitted.

The file starts as ABC; then A is updated to A'BC; then B is updated to form the current version A'B'C.



Representation of active files

Active files are files which are (or have recently been) taking part in a transaction. They need not have been updated to be active. Each active file is represented by an in-store structure called a lock record.

A lock record contains the internal file name, the addresses of the current and the previous file headers and the eleven disc addresses together with their moved bits, the type of file, the file size and one of the file keys with its associated rights. This is merely a cache of part of the data held in the file header, with concessions to the creation of a new version, at which time a file has two relevant file headers.

As their name implies, lock records are also used to lock the files that they describe. An attempt to use a lock record for a conflicting operation causes a process to wait. In the lock record are a reference count and a lock state. The reference count gives the number of readers for a shared lock. The lock state gives the state which the current version (the new if the file is being updated, the old otherwise) has reached. This can be one of unlocked, historic, archaic, reading, writing, deleting.

Unlocked indicates that the file is not in use and that the lock record is merely an up-to-date cache of some of the file header data. This data is normally sufficient to activate the file without requiring to fetch the file header from the disc and is certainly sufficient to locate the correct header. The performance of the server is improved by having enough lock records to cache most of the files which are being worked with by users. Further, the atlas associating internal filenames and header addresses is

only updated when an unlocked lock record is required for a file for which no lock record already exists. This is considerably more rare than the moving of the current header when each version is instantiated. A file just updated is a likely candidate to be updated again in the near future. Lock records are reused on a least recently used basis; when a lock record is needed the one which has been unlocked for the longest is taken. If necessary, the atlas is updated at this moment.

A state of historic indicates that an old version of the file is being accessed but, incidentally, this happens to be the current version too. Archaic indicates that an old version of the file is being accessed, but that it is genuinely old. No locking is implied since an old version can never change (merely be superseded by new versions). Historic lock records can be cloned off lock records for reading and vice versa. If a lock record for writing is cloned off a historic lock then the historic lock is downgraded to archaic since it will probably (this uncertainty does not matter) become superseded. Historic access to a redundant file is permitted, although archaic access is not. This is because there are no multiple versions of redundant files and so two different versions of a redundant file cannot be simultaneously accessible. Marking a file redundant is an efficiency consideration
and so it should only be done if this will not be a restriction. Compiler object code files, for example, may not be edited in place if they are redundant, but this is unlikely to be a noticeable restriction.

A state of reading indicates that the current version of the file is being read and is locked against competing updaters superseding it by creating a new version.

A state of writing indicates that the current version of the file is a tentative one created but not yet instantiated. It is locked against any access by other than the creator.

A state of deleted indicates that the new tentative version is a tombstone for the file, but that it has not yet been instantiated. It is locked against all access by anyone. Only old versions of a deleted file can be accessed, and even they may well have been digested. If the file is redundant then instead of instantiating the tombstone the entire file is destroyed and its pages returned for subsequent reallocation.

There are naturally quite a lot of commands associated with the management of files. These will be described briefly. There are additional commands for administrating the file server which are not described. These commands are privileged and relate to functions such as stopping the file server or initialising a virgin disc. Table 4 contains a summary of the commands available to clients.

Firstly, there is a file create command. This includes the file type: one of normal, redundant or read-only. The command is not privileged, although the file will vanish, if it is not given a textname, when the transaction creating it commits. Creating a file returns the internal filename of the file. It is not an idempotent operation, although the restriction that a file must be textnamed means that any accidentally created files will be short lived. A read-only file is actually a write-once file. The transaction which creates it can also write it. Once that transaction completes it can never alter again.

A file can be opened for access in three ways. These are for reading, for writing and historically. Historic access requires the provision of a timestamp, although a timestamp of zero is conventionally taken to mean the latest version (but with no locking implied). Opening a file returns a channel number, a small integer associated with the transaction in which the file is opened. There is no close operation. Files are closed when the transaction which opened them completes. As well as the internal filename for the desired file, a key must be provided. This key

must grant appropriate access to the file. For genuine historic access, this means the appropriate key at the time in the past that the version existed.

Having opened a file it can be accessed by a read page command and, for a file open for writing, by a write page command. The details of these commands interact quite closely with the implementation of the network. In the implemented system the read page command specifies a port number on the network to where the data should be delivered. The write page command replies with a port number at which the data will be accepted. The last page in a file may be short. For a write page command, the number of bytes is provided in the command; for a read page command, in the reply.

Attributes are handled by two commands. Add attribute gives a new pair of 32 bit integers. The first is the number defining the attribute and the second is its value. By convention, a value of zero causes the attribute to be removed. Attributes with values less than ten are reserved and may not be altered in this way. Get attribute specifies an attribute and returns the associated value. A missing attribute returns zero so, to some extent, an attribute can be hidden by using an unlikely attribute number. Low numbered attributes return file server information such as the file size and timestamp.

The size of a file is the high water mark of the writes to the file, taking into account a possibly short last page. This can be altered by a set length command to either extend or truncate the file.

There is no explicit delete command. Files are deleted by unbinding their textnames, which will either cause their reference count to fall to zero or may cause them to be garbage collected eventually.

A file may be moved from one server to another. This command, which is necessary for sensible archiving if for no other reason, is not currently implemented. No special care is taken at the file level to keep track of where a file has gone. Once moved, the current version becomes historic. Apart from only being accessible historically, it is a candidate for digestion, unless it is read-only, although even then it is not necessarily destroyed immediately. It may be sensible for this command to be privileged. Unfortunately, the problems which it may engender are difficult to deduce from an experimental implementation.

There are also two commands to update reference counts (intended for use by the site implementing the textnaming). The two commands increment and decrement these counts. If they should fall to zero then the file is deleted. For a normal file this means creating a tombstone current version so that old versions remain accessible for a time.

The garbage collector, which is not implemented, will need additional commands to obtain the internal filename of every existing file. By definition, it cannot find them all from the textname graph. Most of the garbage collector's operations, however, are interfaced to the textnaming scheme and not the underlying file scheme.

Table 4: Summary of commands

| | | |
|---|---|---|
| createFile | tno key type | => fno |
| openRead | tno fno key | => channel |
| openWrite | tno fno key | => channel |
| openHistoric | tno fno key time | => channel |
| readPage | tno channel pno port | => bytes (+data at port) |
| writePage | tno channel pno bytes | => port (+data to port) |
| setLength | tno channel bytes | => ok |
| addAttribute | tno fno key att val | => ok |
| getAttribute | tno fno key att | => val |
| incRefCount | tno fno key | => ok |
| decRefCount | tno fno key | => ok |
| | | |
| start | | => tno |
| commit | tno | => ok |
| abort | tno | => ok |
| unjam | tno | => ok |
| | | |
| addKey | tno fno key key right | => ok |

| | | |
|---|---|---|
| att | is | attribute number |
| bytes | is | number of bytes in block |
| channel | is | opened file number |
| fno | is | internal file number |
| key | is | key for file access |
| pno | is | page number within file |
| port | is | network address and subaddress |
| right | is | granted rights |
| tno | is | transaction number |
| type | is | normal, redundant or read-only |
| val | is | attribute value |

Layout on the disc

The disc is divided up by physical disc address into four areas. The bounds of these areas are recorded in a special page, the title page, which occurs as both the first and the last page of the disc. This page is only ever written when the disc is first handcrafted as an empty disc containing no files. The title page also contains the server number, which forms part of each internal filename.

There is also a subtitle page which is occasionally written during normal running. This is held in the second and third pages on the disc. Normally, these pages are identical but the contents of either are always acceptable. If the first is unreadable then the second can be used. The subtitle page records a number of different pieces of information. Firstly, it records whether the holding server is running or not. When a server starts up, crash recovery is necessary if the subtitle page states that the server was running when it stopped. Otherwise, the disc is up to date and no crash recovery is needed. If the subtitle page states that the server is stopped then it also records one or two pieces of information necessary for a smooth restart. The main use of the subtitle page is to record a list of other servers which may need to be consulted during crash recovery.

The first main area on the disc is an atlas of file headers. This is just a large hashtable associating internal filenames with header addresses. In normal running it is not kept up to date due to caching effects and so it needs to be reconstructed during crash recovery.

The next area are the headers themeselves. Among the other information in a header is a bit which states whether or not it is free for allocation. Some of the free headers are used to hold a list of all the free headers so that header allocation can take place quickly. This free list is not maintained up to date on disc and needs to be reconstructed during crash recovery.

Next is a bitmap of the user area, indicating which pages are free and which are allocated to files. This is not kept up to date either and needs reconstruction after a crash.

The last area is the user area itself, consisting of most of the disc. Data and index pages of files are all allocated from this area, using the information in the bitmap. There are no restrictions on what patterns of bits can occur in these pages.

Figure 3 : Layout of pages on the disc

| |
|---|
| TITLE PAGE (copy 1) |
| SUBTITLE PAGE (copy 1) |
| SUBTITLE PAGE (copy 2) |
| ATLAS (locates file headers) |
| FILE HEADERS (maps out files in user area) |
| BITMAP (of user area) |
| USER AREA (data pages from files) |
| TITLE PAGE (copy 2) |

Atomic transactions

All file access is performed within the cocoon of an atomic transaction. Each atomic transaction is managed using data held in an in-store transaction record. This holds a list of pointers to lock records held by the transaction, the transaction number and some data concerned with transaction commitment.

When a file is opened within a transaction then it is activated and the lock record is added to the list of those held by the transaction. Sometimes the lock will already be held and in this case it is merely upgraded if necessary. In both cases, this may involve waiting for competing transactions to complete and to release their locks on the file.

A transaction number is a 32 bit integer and contains the identity of the server which started it and issued the number. This server is called the master server, and any other servers participating in the transaction are called slaves. When another server first receives a request from a transaction, it sends a message to the master server, extracted from the transaction number, adding itself as a slave and it receives a reply either that this has been done or that the transaction does not exist. Each server maintains an on-disc list of all the servers with which it has any dealings. This must guarantee to include all such servers although it is not a strict requirement that it includes no others.

Finally, the transaction commits or aborts. This will normally be at the request of the client but servers may abort transactions to break deadlocks or due to crashes. The commit request is only accepted by the master server.

In the normal case, no other servers will be involved and so commitment is fairly simple. If no files were updated in the transaction then all the locks held are released. Otherwise, one of the updated files is selected as the master file for the transaction. The clock is read to give a timestamp for the transaction. Any other files updated in the transaction are flushed out onto the disc with their headers recording the internal filename of the master file and the timestamp. The data and index pages for the master file are flushed out onto the disc. Finally, the header for the master file is flushed to disc marked as its own master and with its stateword set to "committed". All locks are released. The transaction has now committed. Aborting a single server transaction merely consists of freeing any pages provisionally allocated to the nascent versions. In this case, the lock records for new versions are completely destroyed, so that they will be reconstructed from the file header if the file is accessed again.

Note that the headers for new versions are normally only created at commitment, rather than, as described earlier, when the new version first starts to come into existence. Only if some item of data not cached in the lock records is updated will the new header need to be created earlier. This is done with its state set to "volatile". Thus, in the common single server case, no extra disc transfers are normally involved in atomic commitment.

A transaction involving multiple servers proceeds as follows. First, the master server selects a file held there for which a new version will be instantiated. If it holds none then it creates a file specially for the purpose (this file will eventually vanish). This selected file is the global master file. The clock at the master server is read to get a timestamp for all the new versions being instantiated. First, the master server performs the single server commit already described, except that the file state is set to "ensured" rather than to "committed". Then a message is sent to every participating server informing them of the internal filename

of the global master and of the timestamp.

Each slave server selects a file for which a new version will be instantiated as the local master. They then execute the single server commit scheme already described except that the header for the local master file is not flushed out as its own master, but with a state "ensured" and with the global master file as the master file; the identity of the master server is also recorded in the header. At the completion of this stage every server has all the new versions channelled through a single version, the local master, at that site; and all the local master files are channelled through a single version at the master server, the global master. If any server should fail to reach this stage, then the transaction is aborted, although if the master server should crash, it will not be aborted until crash recovery is completed.

Now the transaction actually reaches the point of no return. The master server rewrites the header of the master file with the state set to "committing". From now on, the transaction is guaranteed to commit whatever happens. The preceding interaction exists to guarantee that all new versions of files held at slave servers are undamaged. The remaining part of the commitment procedure decouples the slaves so that they will later be able to autonomously decide which versions have been instantiated.

The master server sends each slave server a message to commit their part of the transaction. Each slave server rewrites (to a new site) the header for the local master server with state set to "committed" and as its own master. It acknowledges the success (it cannot fail but merely take longer than expected) back to the master server.

Finally, the master server rewrites the header for the global master file with state "committed". This logically unnecessary step means that crash recovery will not need to attempt to retransmit all the commit messages to the involved servers.

A multiple server abort is simple, provided that the transaction has not passed the "ensured" stage after the first round of messages. Up to this point, an abort consists of sending each server a message to abort the transaction. If the server has crashed, it will abort it in any case and so there is no problem. The difficult case arises if a slave ensures the transaction, the slave crashes and then the master needs to abort the transaction (perhaps because another slave had already crashed). Now, the file needs to go into suspended animation. The new version cannot be discarded since it is the only place that the transaction's aborting is recorded. The new version thus remains suspended and locked awaiting the return of the crashed slave. This might prove to be so inconvenient as to justify a special mechanism to deal with it. On the other hand, it is very unlikely.

## Crash recovery

During crash recovery, two different sets of information are needed. Firstly, which servers are involved in transactions of indeterminate state and secondly, which file versions are involved.

The servers attached to the recovering server at the time that it crashed are known from the on-disc list already mentioned. This is used as the list of all slave servers in any rediscovered transactions where the recovering server is the master. It would be more correct to record an individual list in each global master file, but crashes are hopefully rare and multiple crashes where some servers remain crashed for a long time, rarer. Thus, in practise, the optimisation would merely save some message traffic.

First, a message is sent to all connected servers announcing that crash recovery is taking place. These connected servers abort any transactions involving the crashed server which have not reached the point of no return, since they can now never commit. Transactions which are past this point will be rediscovered automatically.

All the file headers are rediscovered from a complete scan of the area of the disc containing them. As file versions are discovered, the atlas associating internal filenames and header addresses is reconstructed. Apart from headers which are not in use, the headers are dealt with as follows.

A header with state "volatile" is part of a transaction in progress at the time of the crash. It can safely be destroyed. It is possible that a message may arrive to commit the transaction containing these versions. The transaction number will be unrecognised and so the request will be rejected.

A header with state "committed" is an instantiated version, as are any versions reached by following the chain of previous versions.

A header with a master file held at the crashed server (but not its own master) is in the same state as its master. If a later version of the master file has already been instantiated, then the older version definitely committed even though it may now have been digested. It may require several passes over the remaining file headers to decide this since a header may be discovered before that of its master file. With enough memory, these multiple passes can be reduced in size and perhaps eliminated. The recovery time is, however, dominated by the time required to perform the pass over all the headers.

A header with state "ensured" is part of a transaction being committed or aborted around the time of the crash. Dealing with these headers is complicated by the fact that they may have already been superseded by another header with state "committing" or "committed". In either of these cases the ensured header is ignored. Consider the case of the slave server first. The global master file, held at another server, will decide whether the transaction committed or aborted. An enquiry is sent to the master server to find out what to do. The global master file can be in one of two states.

It can be "ensured" itself, indicating that the commitment in progress has still not completed. The transaction is aborted, taking care to avoid the race between a recovering slave causing the transaction to abort and the master server finally deciding to commit. It is possible, at a considerable increase in complexity, to keep open the option to commit at this point; this requires reconstructing at least part of the transaction record at the slave server.

The global master can be "committing", indicating that the transaction is past the point of no return. The rediscovered version at the slave should be instantiated.

When a master server rediscovers an "ensured" header it has to handle things differently. The transaction will abort; indeed, it may already have done so at some slave servers. A message is sent to all slaves to abort the transaction. The transaction number is unknown and so this has to be done by specifying the global master file instead. Slave servers may already have aborted (perhaps they crashed too) and so a message that the global master file is unknown is acceptable. Until all these messages have got through successfully, the file cannot be unlocked and the version discarded.

A header with state "committing" can only be discovered at a master server. As with ensured headers, it may have been superseded by a "committed" header and in this case it is ignored. If not, then all the slaves must be informed or reinformed that the transaction must be committed. Again, the transaction number is unknown and so this has to be done by specifying the global master file instead. If the slave server denies knowledge of the global master file then it must already have committed it. The file could be unlocked if a slave server should prove inaccessible. This is not done due to the additional complexity it would introduce; the file remains locked until all slaves have been successfully told to commit.

The details of the recovery algorithm are as follows. The goal of the algorithm is to either accept or reject each file header. Accepting a header will happen if and only if that header is a version to be instantiated. Rejecting a header will happen if and only if that header is a version contained in a transaction which has aborted (or will definitely abort). Firstly, the atlas is emptied since there are, as yet, no accepted files on the volume. An in-store bitmap is intitialised to record which file headers remain to be judged. Whenever a file is accepted, it is inserted into the atlas. All its old versions are also accepted, but these are not inserted into the atlas since the atlas only records the most recent version of a file.

Table 5: Summary of file header states

| State | Own master | Master file local | Master file remote |
|---|---|---|---|
| Volatile | . . . Header is part of incomplete transaction . . . | | |
| Ensured | Global master of completing transaction which will now abort. | Non-master file of transaction which may have committed or may abort. | Local master file for multiple server transaction which may have committed or may abort. |
| Committing | Global master file for multiple server transaction which will commit. | . . . . never occurs . . . . . . . | |
| Committed | Local master file for a long since committed transaction. | . . . . never occurs . . . . . . . | |

The recovery algorithm is multiple pass. It could be reduced to a single incredibly complex pass with a sufficiently large main memory to hold a huge data structure. Some of the passes could be combined with only modest storage requirements. Recovery happens rarely so it seems judicious to place emphasis on getting it correct rather than fast. Only the first pass scans all the headers, and most of the other passes will normally be small.

Phase one consists of a scan to handle the simplest cases, which are the files which can be judged by their own headers alone. Each header which remains to be judged is examined. If it is unreadable then it is certainly not the header of a valid version and so it is rejected. If it is not in use then it is rejected (actually, in this case it can merely be ignored). If it is "volatile" then it is rejected. If it is "committed" then it is accepted, and so are any older versions of the file still in existence. Decisions about headers which are not in these classes are deferred to later passes.

Phase two performs two tasks. It decides about each file which has its master file locally at the recovering server. And it removes all superseded file headers left as debris from transactions which successfully committed. Each header which remains to be judged is rescanned. Firstly, if it is its own master, or the master is at a remote server, then the internal filename of the file itself is looked up in the atlas. If a version at least as recent as the header in question has already been accepted then the header has been superseded and so it is rejected. The second (and only other) case is where the header has a local master. The internal filename of the local master is looked up in the atlas. If a version at least as recent as the header in question exists then the file is accepted (together with its previous versions). This phase is repeated until a pass of it causes no new files to be accepted, since long chains of dependencies on old versions can occur.

Phase three finds all local and global master files participating in transactions which were actually involved in committing at the time of the crash. There are three types of these files. Files with a remote master, files which are their own master and are "ensured" and files which are "committing". The headers remaining are scanned once again and for each header which falls into one of these three classes, in-store data structures are created. A lock record is created for the new version exclusively

locking the file, and a transaction record is created which possesses this exclusive lock, with the file as the local master for the transaction. If a header for the same version is found in both the "committing" and "ensured" state, then the "ensured" header is rejected since it has been superseded by the "committing" one.

Phase four finds all files dependent on those found in phase three. The headers remaining to be judged are scanned once again. The headers being looked for are those with a local master present in the in-store data structure constructed during phase three. A lock record is created for the new version exclusively locking the file, and this lock record is appended to those held by the transaction with the appropriate local master.

Phase five reconstructs the header free list. All the headers still remaining to be judged are rejected. They can only be new versions of files involved in transactions being flushed to disc prior to commitment proper commencing. All the rejected headers are inserted into the list of file headers available for allocation.

Phase six reconstructs the user area bitmap of pages available for allocation. Each accepted file header, and each header corresponding to in-store lock records, is scanned and all the pages comprising it are marked as not available. This pass makes use of the moved bits in the headers and indices to avoid scanning all the index pages of every version.

At this point, the file server can open for service. The disc and the in-store data structures are pretty much as they were at the time of the crash. The fate of some files remains to be decided, but they are all locked. Phase seven decides about these remaining files. For each transaction for which the recovering server is the master, a message is sent to every server attached at the time of the crash. If the global master is "ensured" then this

message tells the slaves to abort, if it is "committing" it tells them to commit. The original transaction number has been lost and so this has to be done in terms of the internal filename of the global master file. When replies have been received from all slaves then the transaction is completed in the normal way, and all the locks are released. If the recovering server is a slave then it sends a message to the master to enquire whether to commit or abort the transaction. According to the reply, the new versions are accepted or rejected in the usual way, and all the locks are released. If any servers are inaccessible during phase seven then some files may remain locked for inconveniently long periods.

Note that all the complexity of the consistency scheme occurs during crash recovery. In normal running, the situation is very much simpler and has almost no overhead for the single server commit, and little for the multiple server commit.

There are four commands associated directly with transaction management, although all other commands also form part of a transaction and so include a transaction number. The four commands are start transaction, commit transaction, abort transaction and unblock transaction. There are also some server to server commands such as add server, commit slave portion or enquire state of global master. These are not described since they are not themselves complex and are fairly predetermined by the approach to committing transactions.

Start transaction merely returns a transaction number. It allocates a transaction record and generates a unique transaction number.

Commit and abort transaction do just that. Abort transaction is one of two commands which will be accepted immediately when the transaction is busy. All other commands will be rejected if there is an outstanding uncompleted command for the transaction at that server. After a commit or abort transaction command has been

received, the transaction number becomes invalid.

Unblock transaction is the other command which is always acceptable. If there is no outstanding command being processed for the transaction at that server then it has no effect. Otherwise, if the uncompleted command is waiting for a locked file to become unlocked then it gives up waiting. The uncompleted command fails with no effect. If the uncompleted command is actually progressing, it is merely marked as unstoppable. It will fail if it ever needs to wait for a file lock. Unblock transaction thus guarantees to complete an outstanding command in a reasonably short time, perhaps because the user became bored and pressed an interrupt key.

## Security

The security is managed by keys, uninterpreted 32 bit integers. In a more complete implementation there would need to be a recommended way of turning a text string into such a key, so that all client systems uniformly convert passwords to keys.

As already mentioned, the file header contains a number of pairs of keys and their associated access rights. This restricted implementation only has three rights: to read the file, to write the file and to issue further keys for the file. Other rights which might be sensible are to append data to the file, to delete the file, to alter file attributes, and to move the file to another server. All operations like this are regarded as reading or writing in the current implementation, according to whether the operation implies any alteration to the file (and hence the allocation of a new header).

Each lock record contains one key and its associated rights. This is the key last successfully used to access the file, on the reasonable assumption that this will be the most likely key to be used on the next access. This is all part of the desire that simple access to the file should entail neither looking in the atlas nor fetching the header from the disc.

There is just one command for manipulating keys, which updates (or adds) the rights associated with a key. If the key does not exist then it is added, otherwise the rights associated with the key are updated. If the associated rights are none, then the key is removed. There is no command to discover the keys associated with a given file.

## Textnaming

As already mentioned, the textnaming scheme is designed to be sufficiently independent of the underlying file implementation that it can be moved to a separate server, although there is no requirement that this be done.

Textname directories are just files. Additionally, an attribute is set to discriminate between directories and other files. Naturally, directories are normal files rather than redundant files. This means that their implementation is considerably simplified. Complex structures can be used within directories without worry that a crash may corrupt the structures or cause the structures to get out of step with the files that they describe.

That the textnaming scheme is implementable does not seem to be contentious. Almost every aspect of it has been used before, although not at the same time. Pathname naming schemes have been

used in many filing systems such as Unix [RIT72]. Both the CAP filing system [NEE77] and the Cambridge file server [DIO80] allow these to become sufficiently complex to require occasional garbage collection [BIR78,GAR80]. Binding of component names to other pathnames has been used in CAL [LAM76] and in Multics [BEN72]. Allowing objects other than files to be preserved in the textname scheme has been used in Unix [RIT72] (for devices and mounted volumes) and in an extension to Unix [RAS80] where non-leaf objects can be named which are capable of handling resolution of the remaining part of the pathname. Naming across volumes has usually been barred due to the problems of consistency. With an underlying consistency scheme this is no longer a problem.

The textnaming scheme has not been implemented. It is implementable, but to investigate how convenient it is in practice would require a genuine multiple server implementation with real users, rather than an experimental prototype. A simple textnaming scheme has been implemented merely to keep track of files when testing other parts of the file server implementation since 64 bit numbers are inconvenient to remember.

If the textnaming is implemented in a client system, then the commands which it makes available to application programs are not in the province of the network filing system. The commands which are described are those which could be implemented by a separate textname server.

There are four main commands available to manipulate the textnaming scheme. Additionally there will need to be a number of commands to handle volumes which influence each other, add new server names and interface to the garbage collector. These latter commands are not considered in detail and, to some extent, will depend on factors outside of the network filing system (such as whether there are name servers on the network). In all these commands, when an internal filename is a parameter then a key must also be provided which grants suitable access.

The first command is that to create a directory. This takes a key and returns the internal file number of a new empty directory. This should be preserved in the textname graph before the creating transaction commits or it will be rather short lived. A directory is just a normal file with a special attribute, but it makes for a more convenient interface to the textname scheme to distinguish directories immediately they are created.

Next there is an add binding command. This could be split up into a number of different commands since the varying format could prove embarassing in a strongly typed language. This takes three basic parameters, although the third is subdivided. The three basic parameters are the internal filename of the directory in which the binding will be preserved, the component name to be bound and the object to be bound. The object to be bound consists of a type code and the defining data, either a string or 64 bit integer. If this object is a file (or a directory) then this data consists of a 64 bit integer, the internal filename. If this object is pathname then it consists of a string. The only complex case is that of adding a binding to a file. In this case, the reference count for the file may need to be incremented, and this requires finding the file. Normally it will be at the creating server, which can be extracted from the internal filename, but otherwise all the servers under the influence of the server holding the directory may need to be searched. If it is not found here but all the influenced servers were available, then the binding can simply be added for it does not affect preservation of the file. If the file could be on an offline influenced volume then the binding must be rejected, since it does have implications for the preservation of the file. These two cases should really be distinguished more explicitly, perhaps by separating the two cases where the binding should and should not have implications for preservation of the file. It is assumed that directories do not drift away from their creating server.

The next command is the remove binding command. It takes the internal filename of a directory, and a component name. The binding is removed from the directory. For a file, an attempt is made to decrement the reference count but it is not an error if this should fail (perhaps because the relevant volume is a normally offline archive).

Finally, there is the resolve command. This takes as parameters a pathname and the internal filename of a directory in which to commence resolution; zero conventionally means starting from the global context. The name is resolved to produce the object it names. The result is thus the 64 bit internal filename and also the remaining unresolved part of the pathname, if any.

## Assessment

The prototype implementation handles almost all of the data transfer primitives, all of the consistency and all of the security. The textnaming scheme has not been implemented.

Apart from textnaming, the few unimplemented areas are as follows. Files may not be moved between servers automatically (keeping the same internal filename). There is no automatic detection of deadlocked transactions, since there are no timeouts associated with waiting for locked files. It is possible to abandon waiting by command, however. There is no automatic digestion of old versions of files. Only single page transfers are implemented; this is partly due to the inter-process communication simulating the network, which cannot cope with arbitrary length messages.

The consistency scheme seems to have as low an overhead as predicted. This depends crucially, though, on the mix of single and multiple server transactions, which is not something which can be predicted accurately from a prototype file system.

We began by refining the notion of a network filing system and deriving some criteria with which to assess such a system. It is appropriate to return to these criteria and reassess the proposed solutions.

### The independence criterion

The first criterion stated that the existence of the network filing system should have no implications for application programs. It should continue to be possible to write portable programs which are independent of whether they are run using a network filing system or a coresident filing system.

This criterion is not difficult to satisfy. Run-time systems for high-level languages have rather stereotyped notions of files. This is because great care has already been taken to avoid operating system and filing system dependencies. If they are assumed to be textnamed, then the textname is just a character string of system dependent form. Files may be read and written sequentially, perhaps also randomly and perhaps by being mapped into the virtual memory space of the program.

None of these requirements is hard to satisfy and the network filing system proposed certainly does this. The only dubious area is that of data representation within files if different client systems are all to be able to access the same file (at a level higher than raw bytes). This requires agreement amongst the designers of client operating systems about how such files are represented.

The efficiency criterion

The next criterion was that access to the network filing system should be about as fast as access to a coresident filing system implemented on a local disc. This criterion is difficult to assess without a genuine implementation to actually measure performance. However, we can examine the solutions proposed for excessive disc traffic or excessive computation requirement. Either of these would indicate probable performance problems.

The data transfer primitives access individual pages from files. In itself this is an efficient operation. The consistency scheme, however, requires a file to be indirected through indices rather than be constructed from a small number of contiguous extents of disc. Large files further require multiple levels of index. This means that a disc page cache in each server is essential if accessing a page is not always to involve several disc transfers. The implementation has such a cache.

Each page written to a file goes to a new place on disc. This means that locating and allocating unused pages must be very fast. Because of disc head latency, these pages should not be too widely scattered or otherwise the common operation of reading a file sequentially will be unnecessarily slow. The implementation uses a bitmap with a pointer to the last bit allocated. Since only single pages are allocated, the bitmaps never need extensive searching and so pages can be allocated quickly.

The textnaming scheme clearly requires a number of disc transfers to resolve a long pathname. However, most file textnames are single components resolved in a known directory such as the user's working directory or a directory of executable programs. The ability to concisely name such a directory with its internal filename means that the resolution of the directory textname need not be repeated.

The consistency scheme is efficient for the single server case. Hopefully, this will be the rule rather than the exception. The multiple server case requires two rounds of messages and some extra disc transfers. It does, however, seem to be faster than any other scheme proposed for commitment of multiple server transactions. This is partly because the time taken to perform digestion of old files is not counted as an overhead since it is not synchronised with client activity at all.

Local discs are not used efficiently. Much of the traffic between a client and the filing system will be creating versions of files which will subsequently be updated further before they are ever accessed from another client. However, the existence of read-only files simplifies the caching of many frequently used files.

Overall, the system proposed seems to be fairly efficient. There is little overhead on a single server transaction, and access to pages in files should be no less efficient than it is on Unix, where it is only seen as a problem by the largest database programs.

## The coherence criterion

The next criterion was that the file servers should conspire to present the illusion of a single filing system and that the facilities available should not vary with file location.

The data transfer primitives are weak in this area. Each client will get the same data upon accessing a file. Whether this is seen at the client as a representation of the same information depends on agreement amonst the designers of client operating systems about how data, especially text files, should be represented as bytes.

The scheme for naming files is coherent at both the textnaming level and the internal naming level, apart from the minor restriction connected with influenced volumes which is necessary to make the rediscovery of lost files tractable. Whether the full generality of the naming scheme is delivered to users by client systems may vary. For example, a client system may have no command to create multiple textnames for a file. However, it will still be able to access existing files through multiple textnames created in some other way.

The security scheme is uniform across the network at the level of keys. Users should have (or not have) access to files independently of which client system they are using. At the very least, this requires agreement amonst client systems about how a character string password is transformed into a key. Changing of passwords is very poorly handled. To improve this requires authority servers of some sort.

Making the consistency guarantees uniform across the network considerably complicates the implementation, especially in the area of crash recovery. In compensation, other areas of the implementation, such as the textnaming, are much easier to implement because of the strong consistency guarantees.

The uniformity of the network filing system proposed is good, but relies on a certain amount of cooperation from client operating systems. There is, unfortunately, no certainty that pre-existing client systems will be able to live with the standards which are needed. Further, there is no certainty that the primitives which the network filing system provides to clients will be sufficient for sophisticated client operating systems in the future.

## The autonomy criterion

The autonomy criterion insists that the file servers comprising the network filing system are largely independent of each other. In particular, there should be no assumption about all the servers simultaneously being in any state.

Only the textnaming and the consistency schemes are not completely autonomous. The textnaming scheme only requires coordination when creating and removing textnames; at these points, reference counts to files need to be udpated. Even this could be avoided by only destroying files when they are detected as lost after a garbage collection of the textnaming graph.

The consistency scheme requires close cooperation only when committing (or aborting) a multiple server atomic transaction. It is in the nature of such a transaction that the file servers cannot operate independently at such a point.

The autonomy of the file servers is thus good. They only cooperate when explicitly requested to by clients, and even then the cooperation is as minimal as possible.

## The ignorance criterion

The design of the network filing system should not be predicated on clients' possessing extensive resources. However, the design should not hinder the use of such resources should the client, in fact, possess them. The network filing system has to remain ignorant of the facilities at clients.

The data transfer primitives assume that clients can buffer a few pages from files since files can only be accessed starting on a page boundary. With a very large page size this could prove an embarassment. The implementation uses a page size of 512 bytes, so

buffering the current page of eight files requires only 4 kilobytes of store.

There is an underlying assumption that client operating systems will be designed to exploit, rather than obstruct, the uniformity that the network filing system seeks to provide. If a client operating system does not cooperate in this way, it does not compromise the filing system but rather seriously reduces the ability to share files amongst different client systems. This assumption seems reasonable. A client system that stores files or uses security keys in an idiosyncratic way will not only hinder their export to other systems, but also hinder files being imported. To an extent, client systems have to compete for users in a network environment. If they are seen to be unreliable, or to make poor use of the facilities available over the network, then users may choose to run other systems. This fact should militate against client systems being too uncooperative.

## The distribution criterion

There are many clients and few servers. This leads to the last criterion, that tasks should be implemented in clients if doing so would not compromise the guarantees of the network filing system.

The data transfer primitives are in terms of raw data; any transformation of this data is done in client machines. The file servers have no concept of records, character codes and so on.

The only other area where this criterion has much impact is in the implementation of the file textnaming scheme. Curiously, being able to implement the textnaming scheme in the clients depends on having a good security scheme and on having resistance against a crashing client system. Such a scheme means that textnaming can be implemented in client systems, in separate servers specially for

the purpose or in the file server machines themselves. Further, a mixture of all these approaches could be used since any synchronisation required in the textnaming scheme is provided by the locking of the transaction mechanism.

It is unfortunate that there does not seem to be any simple scheme whereby the current copy of a file can be held on a local disc at a client system without the need to write it through. Simple solutions seem to have one of two faults. If the file server acknowledges the copy at the client as being the current version, then the strong consistency guarantees are violated if the client system should misbehave and lose the copy. Alternatively, if the file server does not acknowledge the copy at the client as current but rather as a tentative new version, then there is a need for long term locks. These are difficult to handle over crashes since it may be necessary to break them without enumerating them (for example, if a client system has a hardware failure which takes days to fix).

## Summary

There was no guarantee when the original criteria were outlined in chapter 2 that they would be simultaneously satisfiable. The prototype implementation has shown that, in the main, they are.

Further assessment of the network filing system proposed requires a real implementation to be done. There are two areas where a prototype forms an inadequate basis for assessment. The first is performance. It is not possible to do more than estimate the deliverable bandwidth between a client system and a file. The other area which is hard to assess is the ease of use. Only by constructing real client systems with real users will the powers and weaknesses of the network filing system be laid bare.

# Chapter 10: Conclusions

Much of this thesis has been concerned with the compromises necessary to keep the servers of the network filing system as autonomous as possible, without breaking down the user's view of the filing system as a coherent whole. In particular, we started from a belief that it is undesirable, and probably also intractable, to perform any operation on the entire filing system. This has lead to an emphasis on preservation of consistency.

Coresident and single server filing systems traditionally avoid the problems of consistency by two approaches. Firstly, they artificially restrict any consistency guarantees to the filing system structures themselves (level one consistency); they then further restrict even this consistency by strictly partitioning the filing systems into areas (usually disc volumes, but sometimes smaller partitions) which only need to be self-consistent rather than mutually consistent. The second avoidance tactic is to have a program which can regenerate consistency from a damaged filing system. Since each partition only needs to be self-consistent, this regeneration is tractable.

Our original criteria that the facilities and guarantees should not vary with file location mean that the network filing system cannot be statically partitioned into independent areas. This means that regeneration of global consistency is not tractable. In any case, it is a design criterion that all servers need never be running simultanously, so no consistency regeneration program could guarantee to scan the entire filing system anyway. Thus the only way to ensure consistency is never to lose it in the first place. This is, of course, not directly attainable and much of chapter 7 was concerned with precisely what consistency means and how it can be preserved.

Once global consistency is guaranteed, it is much easier to implement other facets of the network filing system, and to implement client systems. Choosing among different file structures in an environment without consistency guarantees is often influenced more by what is not dangerous than by what would really be most appropriate.

Most other schemes for the preservation of consistency have too large an overhead to be considered for a filing system. They do not trade cheapness in normal running for an expensive crash recovery. The new scheme proposed in chapters 7 and 9 has a much lower overhead. To some extent, this is because it trades time for space. Although fast, it uses up disc space. This disc space can be recovered quickly, if necessary, but the recovery time has not been included as an overhead of the scheme. It is assumed that file digestion will take place at off-peak times such as the middle of the night. In a very active filing system, this might require large amounts of disc space to act as a buffer during the day.

Summarising, consistency in a network filing system is so important that a proper scheme for controlling it is a prerequisite for implementation of the other parts of the filing system (unless each part implements its own consistency scheme independently).

The security schemes of other network filing systems have been shown to be lacking. Capability based schemes tie together naming and security at the lowest level, and it seems to be impossible to divorce them in the view of the filing system seen by client systems. Given a capability based scheme it is impossible to implement a textnaming scheme in which some files can be textnamed but not accessed, unless client systems are assumed to be secure and well-behaved.

Access list security schemes seem unsuited to a highly distributed environment since they have the inherently global concept of a username and a way of validating identity. There

seems to be no easy way to implement these global concepts on a local basis. To some extent, the keys of the system proposed are just validated usernames with different powers; the fact that a user can have several means that it is not necessary to reveal a powerful one in order to be permitted access to a file.

The textnaming scheme is more general than in other filing systems. To some extent this is because the proposed scheme would be too vulnerable to crashes without the strong consistency guarantees which are lacking in most filing systems. Unfortunately, the desire to reuse file pages impacts on the naming scheme, and special rules are required to handle lost files. In turn, this leads to further restrictions to make the garbage collection tractable and increase its locality. The textnaming can be implemented in client machines, but their freedom to implement other than the standard scheme is restricted by the insistence that every file have a standard textname. This is an improvement over other network filing systems, where the textnaming is either implemented in the file servers or else must be implemented in the client systems by a trusted program. In both cases, it is the lack of a proper security scheme which makes the textnaming problematic.

The consistency, security and textnaming schemes are not independent. As we have seen, implementation of a textnaming scheme of any complexity requires consistency guarantees if directories are not going to be corrupted and files lost. To be able to devolve the textnaming to client systems requires a good security scheme since security must not be devolved at the same time. The existence of the network means that consistency has to be properly controlled since a restart of the whole network is not an acceptable way to cope with the crash of a single site. This makes it possible to construct a filing system superior to existing ones (including coresident filing systems).

Further research

The main failure of the design proposed, where further research is needed, is in the use of local discs for holding current versions of files, without compromising the strong guarantees of the filing system. The overhead of not being able to do this can be large, especially for the vast majority of files for which attempted concurrent access is unlikely. In any sort of design cycle, whether designing integrated circuits, newspaper pages or programs, files are accessed only by their creator before they are superseded by a new version. It is unnecessary to despatch this version to a file server if it is not going to be accessed.

Atomic transactions, as described in chapter 7, are completely independent. However, there is nothing to prevent a client program from starting more than one transaction. These cannot share exclusive locks since the filing system assumes that they are competing against each other, rather than working towards a common goal. If the client system lets the transactions share such locks, so that one transaction is allowed to see the data of the other, then all the consistency guarantees are void. If a transaction commits after accessing data by routes unknown to the filing system, then the strong guarantees about being able to see a consistent view of a set of files can be invalid. Clearly, the serialisability of transactions is too strong a condition under these circumstances, but suitable weaker conditions are elusive.

The design of a network filing system is one example of a general problem. How can a global abstraction be constructed from local implementations? As well as provision of services on networks, there is some commonality with parallelism in computation, where a large program is executed by a number of small processors [ABE78], and with large designs (such as space-craft or integrated circuits) where a number of largely independent designers work together to achieve a goal. There is possibly some general theory about how to partition such a task among the maximum

number of largely autonomous agents. If there are too few, then the task is excessively centralised at some of them; too many, and the autonomy is lost by the excessive communication required to synchronise them. Many of the criteria derived in chapter 2 are concerned with attempting to increase both the functional distribution and the autonomy. It is an open question whether there are any universal results about how to assess this balance.

## Conclusions

It is possible to implement a network filing system on a large inhomogeneous open network in such a way that the facilities are globally available independent of location, but without sacrificing the efficiency which comes from largely independent local implementations. This can be done without the requirement to trust client systems, and without assuming that file servers will never crash, and without requiring unacceptably complex client systems. The basis for such an implementation has to be the effective control of the consistency of files.

## References

[ABE78] H. Abelson
"Towards a Theory of Local and Global in Computation"
Theoretical Computer Science 6/1, 1978.

[ALE71] Aleph Zero
"Computer Recreations"
Software Practice & Experience 1/2, 1971.

[AST76] M. Astrahan
"System R: Relational Approach to Data Management"
A.C.M. Trans. on Database Systems 1/2, 1976.

[BAR67] D. Barron et al.
"File Handling at Cambridge University"
Proc. A.F.I.P.S. Spring J.C.C., 1967.

[BAR78] J. Barnett
"A Highly Reliable File System which Supports Multiprocessing"
Software Practice & Experience 8/8, 1978.

[BEL79] Bell Laboratories
"UNIX Programmer's Manual"
Bell Telephone Laboratories, 1979.

[BEN72] A. Benoussan, C. Clingen & R. Daley
"The Multics Virtual Memory: Concepts and Design"
Comm. A.C.M. 15/4, 1972.

[BER79] P. Bernstein, D. Shipman & W. Wong
"Formal Aspects of Serialisability in Database Concurrency Control"
I.E.E.E. Trans. on Software Engineering
5/5, 1979.

[BIR78] A. Birrell & R. Needham
"An Asynchronous Garbage Collector for the CAP Filing System"
A.C.M. SIGOPS Review 12/2, 1978.

[BIR79] A. Birrell & R. Needham
"A Universal File Server"
I.E.E.E Trans. on Software Engineering, SE-6/5, 1980.

[BOB72] D. Bobrow et al.
"Tenex - a paged timesharing system for PDP10"
Comm. A.C.M. 15/3, 1972.

[BRO75] F. Brooks
"The Mythical Man-month"
Addison-Wesley, 1975.

[CHA77] M. Challis
"Database Consistency and Integrity in a Multi-user Environment"
in Databases: Improving Usability and Responsiveness, ed. B. Schneiderman, Academic Press 1978.

[CHA81] M. Challis
"Version Management"
in Pergammon Infotech State of the Art Report on Databases, ed. M. Atkinson, to appear December 1981.

[CHU41] A. Church
"The Calculi of Lambda-conversion"
Annals of Mathematical Studies, Princeton
University Press, 1941.

[COL72] A. Colin
"A Design for a Modular Filestore"
Department of Computer Science, Strathclyde
University FEP 8/72, 1972.

[COP80] G. Copeland
"What if Mass Storage Were Free?"
5th Workshop on Computer Architecture for
Non-numeric Processing, Pacific Grove CA, 1980.

[CYP78] R. Cypser
"Communications Architecture for Distributed Systems"
Addison-Wesley, 1978.

[DEC78] Digital Equipment Corporation
"VAX/VMS Summary Description"
Digital Equipment Corporation, 1978.

[DEC80] Digital Equipment Corporation, Intel Corporation
& Xerox Corporation
"The Ethernet"
Xerox Corporation, 1980.

[DEL80] C. Dellar
"Removing Backing Store Administration from the
CAP Operating System"
A.C.M. SIGOPS Review 14/4, 1980.

[DEW77] H. Dewar, V. Eachus, K. Humphry & P. McLellan
"The Filestore"
Department of Computer Science, Edinburgh
University, 1977.

[DEW80] H. Dewar
"ISYS80 - reference manual"
Department of Computer Science, Edinburgh
University, 1980.

[DIF76] W. Diffie & M. Hellman
"Multiuser Cryptographic Techniques"
Proc. A.F.I.P.S., 1976.

[DIO80] J. Dion
"The Cambridge File Server"
A.C.M. SIGOPS Review 14/4, 1980.

[ESW76] K. Eswaren, J. Gray, R. Lorie & I. Traiger
"The Notion of Consistency and Predicate Locks
in a Database System"
Comm. A.C.M. 19/11, 1976.

[GAR80] N. Garnett and R. Needham
"An Asynchronous Garbage Collector for the
Cambridge File Server"
A.C.M. SIGOPS Review 14/4, 1980.

[GRA77] J. Gray
"Notes on Database Operating Systems"
in Operating Systems: an Advanced Course
Springer-Verlag 1978.

[HOA74] C. Hoare
"Monitors: an Operating System Structuring Concept"
Comm. A.C.M. 17/10, 1974

[ISR78] J. Israel, J. Mitchell & H. Sturgis
"Separating Data from Function in a Distributed File System"
Proc. 2nd International Colloq. on Operating Systems, IRIA, 1978.

[JEN74] K. Jensen & N. Wirth
"Pascal User Manual and Report"
Springer-Verlag, 1974.

[LAM74] B. Lampson
"An Open Operating System for a Single User Machine"
in Aspects Theoriques et Pratiques des Systemes d'Exploitation, Roquencourt, 1974.

[LAM76] B. Lampson & H. Sturgis
"Reflections on an Operating System Design"
Comm. A.C.M. 19/5, 1976.

[LAM78] L. Lamport
"Time, Clocks, and the Ordering of Events in a Distributed System"
Comm. A.C.M. 21/7, 1978.

[LAM79] B. Lampson & H. Sturgis
"Crash Recovery in a Distributed Storage System"
Comm. A.C.M. to appear, 1979.

[LAM80] B. Lampson & D. Redell
"Experience with Processes and Monitors in Mesa"
Comm. A.C.M. 23/2, 1980.

[LAM81] B. Lampson
Private Communication, 1981.

[LAU80] L. Laub
"Optical Mass Storage Technology"
Proc. 1980 Workshop on Computer Architecture for Non-numeric processing, Pacific Grove CA.

[LOR77] R. Lorie
"Physical Integrity in a Large Segmented Database"
A.C.M. Trans. on Database Systems 2/1, 1977.

[MCL78] P. McLellan
"LEGOS - user reference manual"
Department of Computer Science, Edinburgh University CSR-49-79, 1978.

[MCQ77] J. McQuillan & D. Walden
"The ARPA Network Design Decisions"
Computer Networks 1, 1977.

[MET76] R. Metcalfe & D. Boggs
"Ethernet - a Distributed Packet Switching for Local Computer Networks"
Comm. A.C.M. 19/7, 1976.

[NBS78] National Bureau of Standards
"Data Encryption Standard"
Federal Information Processing Standards Publication 46, 1977.

[NEE77] R. Needham & A. Birrell
"The CAP Filing System"
Proc 6th Conf. on Operating System Principles, Purdue University, 1977.

[NEE78] R. Needham & M. Schroeder
"Using Encryption for Authentication in Large Computer Networks"
Xerox CSL-78-4, 1978.

[NPL77] National Physical Laboratory
"The Central Filestore"
National Physical Laboratory, Doc S3, 1977.

[PAX79] W. Paxton
"A Client-based Transaction System to Maintain Data Integrity"
Proc. 7th Symposium on Operating System Principles, Pacific Grove CA, 1979.

[RAS80] R. Rashid
"An Inter-Process Communication Facility for Unix"
Department of Computer Science, Carnegie-Mellon University, 1980.

[RED80] See after Z

[REE79] D. Reed
"Implementing Atomic Transactions on Decentralised Data"
7th Symposium on Operating Principles, Pacific Grove, CA, 1979.

[RIE77] D. Ries & M. Stonebraker
"Effects of Locking Granularity in a Database Management System"
A.C.M. Trans. on Database Systems 2/2, 1977.

[RIE79] D. Ries & M. Stonebraker
"Locking Granularity Revisited"
A.C.M. Trans. on Database Systems 4/2, 1979.

[RIT74] D. Ritchie & K. Thompson
"The UNIX Timesharing System"
Comm. A.C.M. 17/7, 1974.

[RIT78] D. Ritchie
"A Retrospective"
Bell System Technical Journal 57/6, 1978.

[RIV80] R. Rivest
"A Description of a Singe-chip Implementation of the RSA Cipher"
Lambda 1/3, 1980.

[SAL77] J. Saltzer
"Naming and Binding of Objects"
in Operating Systems: an Advanced Course
Springer-Verlag 1978.

[SCH77] B-M. Schueler
"Update Reconsidered"
Proc I.F.I.P. Working Conference on Database Management Systems, Nice, 1977.

[SEV76] D. Severance & S. Lohman
"Differential Files: their Application to the Maintenance of Large Databases"
A.C.M. Trans. on Database Systems 1/3, 1976.

[SKE80] D. Skeen & M. Stonebraker
"A Formal Model of Crash Recovery in a Distributed System"
I.E.E.E. Trans. on Software Engineering, to appear, 1980.

[SKE81] D. Skeen
"Nonblocking Commit Protocols"
A.C.M. SIGMOD International Conference on
Management of Data, Ann Arbor MI, 1981.

[STO76] M. Stonebraker, E. Wong, P. Kreps & G. Held
"The Design and Implementation of Ingres"
A.C.M. Trans. on Database Systems 1/3, 1976.

[STU80] H. Sturgis, J. Mitchell and J. Israel
"Issues in the Design and Use of a Distributed
File System"
A.C.M. SIGOPS Review 14/3, 1980.

[SWI79] D. Swinehart, G. McDaniel and D. Boggs
"WFS: a Simple Centralised File System for a
Distributed Environment"
Proc. 7th Symposium on Operating System
Principles, Pacific Grove CA, 1979.

[TAY79] D. Taylor, D. Morgan & J. Black
"Redundancy in Data Structures: Improving Software
Fault Tolerance"
Department of Computer Science, University of
Waterloo CS-79-34, 1979.

[TEI77] W. Teitelman
"A Display Oriented Programmer's Assistant"
Xerox CSL 77-3, 1977.

[THO73] R. Thomas
"A Resource Sharing Executive for the Arpanet"
Proc. A.F.I.P.S. 42, 1973.

[THO76] R. Thomas
"A Solution to the Update Problem for Multiple Copy Databases Which Use Distributed Control"
B.B.N. Report 3340, July 1976.

[THO78] K. Thompson
"UNIX Implementation"
Bell System Technical Journal 57/6, 1978.

[WEC80] S. Wecker
"DNA: the Digital Network Architecture"
I.E.E.E. Trans. on Communications 28, 1980.

[WIL72] M. Wilkes
"On Preserving the Integrity of Databases"
Computer J. 15/3, 1972.

[WIL79] M. Wilkes and D. Wheeler
"The Cambridge Digital Communications Ring"
Proc. Local Area Communications Network Symposium, Boston, 1979.

[RED80] D. Redell et al
"Pilot: an operating system for a Personal Computer"
Comm. A.C.M. 23/2, 1980.