



This work is protected by copyright and other intellectual property rights and duplication or sale of all or part is not permitted, except that material may be duplicated by you for research, private study, criticism/review or educational purposes. Electronic or print copies are for your own personal, non-commercial use and shall not be passed to any other individual. No quotation may be published without proper acknowledgement. For any other use, or to quote extensively from the work, permission must be obtained from the copyright holder/s.

RELIABLE FILE STORAGE
IN A DISTRIBUTED COMPUTING SYSTEM

by

Kenneth Lunn

A thesis presented in support of an application
for the degree of Doctor of Philosophy in the
University of Keele

March 1982

1.0	ABSTRACT	2
2.0	ACKNOWLEDGEMENTS	4

CHAPTER 1 INTRODUCTION

1.1	THE PROBLEM	1-1
1.2	THE APPLICATION AREA	1-4
1.3	A SOLUTION TO THE PROBLEM OF RELIABLE FILE STORAGE	1-5
1.4	CONCLUSION	1-9

CHAPTER 2 RELIABILITY THEORY

2.1	INTRODUCTION	2-1
2.2	SYSTEMS AND THEIR FAILURES	2-2
2.2.1	Systems	2-2
2.2.2	Specification	2-3
2.2.3	Reliability	2-4
2.2.4	Errors, Faults And Failures	2-5
2.2.5	Fault Avoidance	2-6
2.2.6	Fault Tolerance	2-7
2.2.7	Fault Tolerance Techniques	2-8
2.2.7.1	Protective Redundancy	2-8
2.2.7.2	Error Detection	2-9
2.2.7.3	Fault Treatment	2-11
2.2.7.4	Damage Assessment	2-13
2.2.7.5	Error Recovery	2-14
2.2.7.6	Component Dependency	2-17
2.2.7.6.1	Independent Components	2-17
2.2.7.6.2	Dependent Components	2-18
2.2.7.6.3	Conclusion	2-20
2.3	CONCLUSION	2-20

CHAPTER 3 LOCAL AREA NETWORKS

3.1	INTRODUCTION	3-1
3.2	DEFINITIONS	3-3
3.2.1	Communications	3-3
3.2.1.1	The Cambridge Ring	3-4
3.2.2	Protocols	3-6
3.2.3	Homogeneous And Heterogeneous Systems	3-7
3.2.4	Autonomy	3-8
3.2.5	Servers And Clients	3-10
3.3	PROS AND CONS OF LOCAL AREA NETWORK SYSTEMS	3-11
3.4	SURVEY OF EXISTING LOCAL AREA NETWORK COMPUTER SYSTEMS	3-15
3.4.1	Cambridge Model Distributed System	3-15
3.4.2	Xerox Ethernet	3-18
3.4.3	Apollo Domain	3-20
3.4.4	Z - Net	3-21
3.4.5	Dec Net	3-21
3.4.6	Unix Satellite Processor System	3-22
3.4.7	Other Distributed Unix Systems	3-23

3.5	CONCLUSION	3-24
-----	----------------------	------

CHAPTER 4 FILE-STORES

4.1	DEFINITIONS	4-1
4.1.1	File-stores, File-server And Files	4-1
4.1.2	Database	4-2
4.1.3	Naming	4-3
4.1.4	Protection	4-6
4.1.5	Mutual Exclusion	4-6
4.1.6	Deadlock	4-8
4.1.7	Consistency	4-9
4.1.8	Atomicity And Transactions	4-10
4.2	ISSUES IN FILE-STORE DESIGN	4-11
4.2.1	Data Placement	4-12
4.2.2	Consistency	4-13
4.2.3	Shared Access	4-14
4.2.4	Shared Update	4-15
4.2.5	Naming	4-16
4.3	A SURVEY OF A NUMBER OF EXISTING FILE-STORES	4-18
4.3.1	Unix	4-18
4.3.2	GEC OS4000	4-21
4.3.3	OS 360	4-22
4.3.4	Xerox DFS	4-23
4.3.5	Apollo Domain	4-26
4.3.6	George III	4-27
4.3.7	Other Unix-like Distributed File-stores	4-29

CHAPTER 5 REFINEMENT OF THE PROBLEM

CHAPTER 6 KUDOS

6.1	THE KEELE UNIVERSITY DISTRIBUTED OPERATING SYSTEM	6-1
6.2	AIMS OF KUDOS	6-2
6.2.1	General Objectives	6-3
6.2.2	Objectives Within Our Application	6-4
6.2.3	Experience	6-6
6.3	KUDOS STRUCTURE	6-10
6.3.1	The KUDOS Message Passing Scheme	6-11
6.3.2	Client-Server Architecture	6-16
6.3.3	Public/Private Domain Architecture	6-16
6.3.4	The Conceptual Structure	6-18
6.3.4.1	The Personal Autonomous Work Station	6-22
6.3.4.2	The Multi-Access Shared System	6-23
6.3.4.3	The Public System Interface	6-23
6.3.5	Object Naming, Addressing And Location	6-23
6.3.5.1	Resources	6-24
6.3.5.2	Addressing	6-25
6.3.5.3	Processing	6-25
6.3.5.4	Resource Location	6-26
6.3.5.5	Some Effects Of Node Crashes And Start-Ups	6-29
6.3.5.6	Alternative Schemes	6-30

6.4	KUDOS IMPLEMENTATION TO DATE	6-32
6.4.1	File-store	6-32
6.4.2	UCSD Filer Interface	6-32
6.5	EXPERIENCE AND CONCLUSIONS	6-33
6.5.1	Hardware	6-33
6.5.2	Software	6-35
6.6	OVERALL CONCLUSIONS	6-40

CHAPTER 7 KUDOS FILE-STORE - DESCRIPTION

7.1	THE KUDOS FILE-SERVER	7-8
7.1.1	Introduction	7-8
7.1.2	File-server Primitives	7-10
7.1.2.1	Create And Delete	7-12
7.1.2.2	Expand And Shrink	7-12
7.1.2.3	Read And Write	7-13
7.1.2.4	Failures	7-13
7.1.2.5	Desirable Extensions To The Primitives	7-14
7.1.3	File-server Protection	7-17
7.1.4	File-server Reliability	7-19
7.2	THE DIRECTORY SYSTEM	7-22
7.2.1	General Description	7-22
7.2.1.1	Hierarchy	7-22
7.2.1.2	The Active File-store	7-23
7.2.1.3	Associated Volumes	7-24
7.2.1.4	Mount/dismount	7-27
7.2.1.5	Directory Resolving	7-29
7.2.1.6	Timestamping	7-31
7.2.1.7	Atomicity	7-32
7.2.2	File Protection	7-35
7.2.2.1	System Defined Capabilities	7-35
7.2.2.2	User Defined Capabilities	7-36
7.2.2.3	Locking	7-37
7.2.3	Directory System Primitives	7-40
7.2.3.1	Activate_subdirectory	7-42
7.2.3.2	Read_associated_volumes	7-44
7.2.3.3	Create_subdirectory And Delete_subdirectory	7-44
7.2.3.4	Insert_file	7-45
7.2.3.5	Find_file	7-45
7.2.3.6	Delete_file	7-46
7.2.3.7	Read_lock, Write_lock, Refresh_lock And End_lock	7-46
7.2.3.8	List	7-47
7.2.3.9	New_read_permit And New_write_permit	7-48
7.3	CLIENT VIEW OF FILE-STORE	7-49
7.3.1	File-server/directory Relationship With Client	7-49
7.3.2	Client Error Detection/Recovery	7-51
7.3.3	Deadlock Detection/recovery	7-52
7.3.4	Example Of Use Of File-store	7-53

CHAPTER 8 KUDOS FILE-STORE - ASSESSMENT

8.1	THE DESIGN	8-1
-----	----------------------	-----

8.1.1	Structure And Methodology	8-2
8.1.2	Reliability	8-10
8.1.2.1	Elementary Reliability Calculations	8-12
8.1.2.2	Access	8-19
8.1.2.3	Prevention Of Loss	8-19
8.1.2.4	Other Aspects Of Reliability Theory	8-20
8.1.3	Performance	8-24
8.1.3.1	File-server	8-24
8.1.3.2	Directory System	8-25
8.2	NEW IDEAS	8-28
8.3	FUTURE RESEARCH	8-35

CHAPTER 9 SUMMARY AND CONCLUSIONS

9.1	SUMMARY	9-1
9.2	ACHIEVEMENTS	9-3
9.2.1	Personal	9-3
9.2.2	Local	9-5
9.2.3	Global	9-6

CHAPTER 10 REFERENCES

1.0 ABSTRACT

This thesis considers the problem of how to construct a file-store which is reliable in terms of high accessibility of data and low likelihood of data loss. In particular the problem is discussed in the context of a local area network computer architecture. A novel approach is taken to the implementation of the naming network for files, which controls multiple copy redundancy. The naming network is a single hierarchy which incorporates redundancy in the access paths to files as well as in the files themselves, thus improving accessibility as well as reducing likelihood of file loss. A prototype file-store was developed and implemented; to facilitate this the author had to develop a simple distributed operating system which evolved as an interesting research project in its own right. A distributed name-server algorithm was developed, and interesting insight gained into the design of local area network computer systems. The property of local area networks important to the file-store is scope for dynamic redundancy; the file-store is constructed out of a number of independent file-servers. Inconsistencies between multiple copies of a file are resolved automatically. Levels of redundancy and data location are controlled through the naming network, allowing replication of files to any degree thought necessary, bounded by the number of independent storage volumes in the system. Deadlock avoidance and automatic reconfiguration on hardware

component failure are included. Some simple combinatorial mathematics is included to highlight the reliability of multiple independent copies of a file; the need for a more quantitative approach is indicated.

2.0 ACKNOWLEDGEMENTS

The work presented in this thesis was undertaken within the Department of Computer Science at the University of Keele. About the time of my arrival a project commenced on distributed file-stores sponsored by the Science and Engineering Research Council, and it is within this project that the work evolved.

I am grateful to the Department of Computer Science, firstly for accepting me to undertake a Ph.D.; despite having spent two years as a Cobol applications programmer, they divined some promise in me which I hope I have fulfilled. The department is a small, friendly one, and I was made to feel part of the family far more than might have been the case in a larger, more established department.

I must also thank the undergraduates whom I had the honour to teach. In teaching them I had to close large gaps in my own knowledge. It was interesting to note the variety of students who take computer science at Keele, no doubt a consequence of the philosophy of the University.

The distributed file-store project was headed by Keith Bennett, who also acted as my supervisor. I must thank him for his encouragement and guidance and collaboration on a number of research papers. The other members of the project, namely Pearl Brereton, Paul Singleton, Philip Peake and Pete Truman, also deserve thanks.

The Science and Engineering Research Council is to be thanked for financing both myself as a research student and the distributed file-store project.

Finally, I must thank my wife Susan, who agreed that I should embark on such a venture, accepted the financial sacrifices, and provided plenty of encouragement with few complaints. No marriage is thoroughly tested until it has suffered a Ph.D. and the consequent thesis.

CHAPTER 1

INTRODUCTION

1.1 THE PROBLEM

This thesis considers the problem of how to construct a file-store which is reliable in terms of high accessibility of data and low likelihood of data loss. In particular, the problem is discussed in the context of a local area network architecture.

The contribution of this thesis is seen to be largely practical; a solution to the stated problem has been proposed and implemented as a prototype. The algorithms for this solution will be presented, together with extensive discussion of the implementation.

A number of solutions to file-store reliability have already been tried, and will be discussed later. The solution presented here differs from most in that it provides a user-oriented naming scheme with user-oriented protection mechanisms. Multiple copy redundancy is controlled through the naming network and includes redundancy of access paths. Tailoring of redundancy for

files and the location of files is also controlled through the naming network.

We shall try throughout to distinguish between file-store and file-server. The former we see as providing a high level of functionality, the latter as a repository for files and as such only a component part of a file-store. For example, a file-store may associate user-arbitrary names with files whereas a file-server may only refer to files by a system generated name. In our solution, the file-store will also administer multiple copies of files. A fuller distinction between file-store and file-server will appear later.

The desirability of a reliable file-store needs little justification. Improving technology has meant that data loss through hardware failure is relatively rare. On the other hand, storage devices are capable of holding more and more data so that a single hardware failure can lose vast quantities of data. Anyone who has lost even a day's work through data loss knows how inconvenient and time-consuming recovery can be. Complete loss of large quantities of data, to commercial organisations in particular, can be disastrous.

Consequently, most organisations and/or individuals take care to secure data, largely by copying crucial data sets, usually to magnetic tape. A former employer of the author duplicated all critical files and kept punched cards

(the primary data entry) for several days after use. Such practices are common.

The problems with such ad hoc duplication of data are that it requires a large amount of organisational effort, it is prone to oversight, and involves extensive work to recover from a failure. More automatic means of data replication and recovery would be of significant use. Some systems already provide this in a limited way, for example, the George III system described later. A flexible response is also desirable, providing different degrees of reliability for different files according to their importance.

What is meant by reliability will be discussed early on in this thesis. Intuitively, reliability is the property of not going wrong too often. Unfortunately, systems which never go wrong are part of an unrealisable utopia. We shall have to content ourselves with minimising the likelihood of failure, restricting the consequences of failure, and hiding the effects of failure. A successful system might be one that, although it fails, is never seen to fail, or which never has a greatly upsetting failure. To a large extent, appearances are all important.

1.2 THE APPLICATION AREA

A local area network can be loosely defined as a collection of inter-connected computers within a small area of (say) one square mile. These computers can cooperate on common tasks and typically service the general needs of an organisation such as a university or a commercial enterprise.

In terms of cost, local area networks have become viable only over the last few years. This is partly because the diminishing cost of computers has made possible experimentation and techniques which previously would have been outrageously extravagant, and partly because of the relatively recent development of appropriate, low-cost communication systems. It is interesting to reflect how economics can significantly change emphasis in both research and development.

Local area networks have provided a significant amount of research material. A too rigorous definition of a local area network would exclude many useful research projects. Existing local area networks range from ad hoc interconnections of existing resources on a number of university campuses, to fully-fledged integrated computer systems such as the Cambridge Model Distributed System described later. Research topics cover such areas as communications, system reliability, software design, file storage, and concurrency.

It is the belief of the author that the most fruitful research will stem from integrated, purpose-built systems rather than ad hoc systems. One can then view a local area network as an alternative form of computer system architecture. We shall later discuss the features of this sort of architecture, in particular those most appropriate to the problem at hand.

The choice of local area network as an implementation vehicle for the solution to the stated problem is justified in two ways. Firstly, local area networks are a worthy area for research. This is an "art for art's sake" definition, but perfectly admissible. Secondly, there are a number of features of local area networks appropriate to the stated problem.

Briefly, the appropriate features of local area networks are (without definition as yet) scope for redundancy, independence of components, and potential for concurrency. Other features of local area networks, such as incremental growth and the ability to absorb new applications, are important but not related to the problem. We shall explore local area networks more fully later.

1.3 A SOLUTION TO THE PROBLEM OF RELIABLE FILE STORAGE

The *raison d'être* of this thesis is a set of algorithms which provide a solution to the stated problem. Related to this is a distributed operating system developed by the

author called KUDOS (Keele University Distributed Operating System). The KUDOS file-store encompasses the file-store algorithms and has been implemented as a prototype.

KUDOS itself provided some useful research. A novel algorithm for resource location was developed and implemented, and will be described later. This was published as [LUNN3]. Also a great deal of time and heartache were spent on communications software, which can largely serve as an example of how not to approach communications. When reading the description of that experience the reader might do well to pity the poor tyro.

In retrospect, KUDOS was little more than a necessary diversion. An implementation vehicle was required for the algorithms. Had a suitable vehicle been readily available then it would have been exploited. However, hardware and software resources, at the time, in the computer science department at Keele, were severely limited. Researchers might well be advised to beg, borrow and steal whatever they can to make life easier; developing tools can be time-consuming and leads one to mistakes and problems already solved by others. A readily available operating system would have allowed more time for developing and experimenting with the file-store algorithms.

Actual implementation effort was split almost evenly between KUDOS and the KUDOS file-store. The completion of the prototype took twelve months. The design and

development of the algorithms spanned two years, spreading into the development of the prototype. Early versions of the file-store algorithms appeared in [LUNN4].

The final set of algorithms provide a number of important features. These are:

1. Single global hierarchical naming scheme. The naming mechanism appears to the user as a single hierarchy similar to that provided by Unix and many other operating systems. There is no notion of location embedded in the name of a file, unlike many such file-stores on distributed systems, for example the Apollo system. A user names a file in the same way from any part of the system.
2. Controlled multiple copy redundancy. It is possible to replicate a file on all or any subset of the set of volumes in the system. If a copy falls behind because its storage volume is inaccessible, it is brought up to date before it is accessed by reference to other copies in the system. The algorithms presented can respond flexibly to requirements by minimising the likelihood that an out of date file is accessed or by removing that possibility under normal progress of the system (including the response to system failures). Non-critical files can be stored as one-copy, as are uncommitted files during update.

3. Neat solution to data placement. Through the hierarchy of directories, though not embedded in the name of a file, the placement of copies of a file is controlled. This is by a mechanism called "associated volumes" which will be described fully later. This scheme also provides another benefit in that a file can be accessed if any volume containing a copy of that file is online. This is because each volume contains a copy of all paths to all files held by it.

4. Deadlock avoidance. It was decided to duck the issue of deadlock detection and to adopt a policy of deadlock avoidance. This is perhaps a better alternative in a system where no centralised control exists, but we shall not argue the case. The mechanism chosen is based on timeout of locks, and relies to some extent on the reasonable behaviour of software using the file-store.

5. Automatic reconfiguration. If a node in the system fails, the file-store will reconfigure itself to provide a continued service. Any software dependent on a failed node for a particular transaction may, however, have to back out and retry in the new configuration.

6. Limited checkpointing and recovery. By adopting a policy of careful replacement, process failure should not leave a file in a partially complete state.
7. File protection. A system of keys, generated by the system and by the user prevent illegal access to data stored in the file-store, and implement a data privacy scheme for users.

1.4 CONCLUSION

The work presented in this thesis is fairly straightforward. The problem of reliable file storage is cited, a solution proposed and a prototype implemented. Along the way, some useful results were derived, both directly and indirectly relevant to the problem. Useful experiences were gained, and the work provided insight into organisational problems of research, as well as the more abstract nature of it.

We shall begin the main body of the thesis by exploring the terms of reference. Pertinent aspects of reliability theory will be presented and set in context. An appraisal of current developments in local area networks will be made with the intention of extracting appropriate characteristics for the solution to the stated problem. An appreciation of current file-stores will be included, with a discussion of existing solutions to file-store reliability. We shall then

refine the stated problem in the light of the definitions made.

The next step will be to examine KUDOS. This is a distributed operating system based on a network-transparent means of communication, in the sense that communication between any two processes is the same irrespective of whether or not they are on the same node. Some bitter experiences with communications will be related. A novel algorithm for resource location will be presented. It is important, however, to view KUDOS as an implementation vehicle for the file-store.

The file-store will then be described with the algorithms which provide the aforesaid features. Some experiences with the implementation will be related. There will be some assessment of the file-store itself in terms of its structure, reliability and performance; these will have to be largely a priori. Finally, some examination of research stemming from this thesis, and possible alternative methods of approach, will be considered.

CHAPTER 2

RELIABILITY THEORY

2.1 INTRODUCTION

We shall consider the topic of reliability theory as users rather than contributors, although a practical application of the theory forms the basis of this thesis. The most substantial body of work on reliability theory, at least in the United Kingdom, stems from Newcastle University. Representative of their work is [ANDE1], [ANDE2], [RAND1] and [RAND2], and this chapter will draw heavily from these sources. We shall consider some aspects presented in [SH00], particularly on component dependency.

Reliability is an important parameter of good design. It is the function of reliability theory to highlight the major principles for achieving reliability, and to provide a number of general policies, structures and techniques for constructing reliable systems. To a large extent, reliability theory is an application of intuitively obvious principles discovered (and probably repeatedly rediscovered) by all good designers; this is commensurate with the view that science is the rigorous application of common sense.

We shall commence this humble guided tour of reliability theory by trying to define what we mean by reliability. To a large extent we must rely on intuition, at least in general terms. Engineering principles usually only take on a precise meaning in a specific context. We shall then discuss failures in a general way, and some principles and techniques for minimising their consequences.

Throughout, we shall try to illustrate, with examples where appropriate, which principles and techniques are most pertinent to the problem of reliable file storage, and in particular those which have been drawn upon in constructing the KUDOS file-store.

2.2 SYSTEMS AND THEIR FAILURES

2.2.1 Systems

We shall consider a system to be a set of components together with their inter-relationships and their connections with the outside world. This definition is by no means rigorous, but is useful and intuitively obvious. A system itself might well be a component of another system, and likewise a component of a system might be seen as a system in itself decomposable into its own components.

A useful example, familiar to most people, is a car. A car can be seen to be constructed of a number of components such as engine, body shell, wheels, transmission. An engine can be seen itself as a system, comprised of pistons,

cylinders, crankshaft, ignition and so on. A car itself might be seen as a component of a taxi service.

Likewise, a file-store can be seen as a system composed of file-server, directory scheme, recovery algorithms. The file-store is a component of a computer system, and a file-server can be seen as a system in its own right consisting of disc-servers, data-maps and so on.

It is important to clarify the boundaries of a system. For example, a lord might consider a chauffeur as a constituent component of a car. Lesser mortals could not afford such a definition, but might implicitly assume certain components such as a heated rear window though a car might be defined without them.

2.2.2 Specification

The specification of a system is a set of statements concerning the behaviour and external states of a system. These may include a set of rules regarding the transition from one external state to another as a result of external stimuli, or constraints on the external states. For example, the specification of a car might include the number of passenger seats, luggage capacity, maximum speed and fuel consumption.

The subject of system specification is of great importance, and there is much research on how to specify systems rigorously (eg. [JONES]). Ambiguous or incomplete specification of a system is a major failing too commonly experienced by designers. However, to specify exhaustively any large system is probably infeasible in practice, at least with the tools currently available and with the time constraints usually imposed. Specifications, therefore, usually include implicit assumptions commonly associated with the type of system.

The internal state of a system is the aggregate of all the external states of its constituent components. The external state is an abstraction of the internal state. Thus, in passing from one external state to another a system may well pass through a number of internal states. A specification does not directly stipulate the internal states of a system, nor indeed the components of a system. Two functionally identical systems may well have different internal structure and behaviour despite satisfying the same specification. Intuitively then, a specification states what a system does and not how it does it.

2.2.3 Reliability

The reliability of a system is a measure of the success with which it conforms to specification and can be quite arbitrary. For example, one measure might be the average time to deviation from the specification (mean time to

failure), or perhaps the number of times it deviates from specification over a fixed period. It is important to realise that the reliability of a system cannot be discussed without reference to the specification.

The reliability of a system may well involve a number of measures, depending on the type of deviation from specification. For example, one measure of a file-store's reliability might be the frequency with which it loses files (that is, it deviates from the specification which states that it stores files), and another measure might be the frequency with which stored files are inaccessible (that is, it deviates from the specification that it retrieves files placed therein). The reliability of any system is likely to be a multi-valued measure.

2.2.4 Errors, Faults And Failures

We term the internal state of a system an erroneous state when that state is such that further processing by the normal algorithms of the system will cause a deviation from the specification (that is a failure). The term error is used to designate that part of a system which is incorrect. A fault is a mechanical or algorithmic cause of an error.

For example, a car might be in an erroneous state if fuel stops flowing to the carburettor. Continued use of the car will result in the engine ceasing to propel the car, thus causing a failure. The fault might be a break in a

fuel line or malfunction of the fuel pump.

Similarly, a file-store might be in an erroneous state if an area of a disc becomes corrupted. Future access to a file stored using that area of a disc will cause a failure. The fault might be a disintegrating disc surface or a malfunction in the disc controller.

A fault might be permanent, meaning that the system will continue to malfunction, or transient, meaning that the system will function correctly in the future. Communication systems are prone to transient errors, where messages might be garbled due to the effect of the environment, say a lightning strike or a power fluctuation.

2.2.5 Fault Avoidance

One method of reducing the likelihood of a failure is to choose components with a low likelihood of failure. No special methods are taken to deal with faults within the system, so that a fault will eventually cause a failure. Handling of a fault is outside the bounds of the system.

For the designer, fault avoidance is the simplest method of preventing failure. If one can incorporate components with low likelihood of failure at reasonable cost, then that is the most sensible approach. Designers should incorporate fault avoidance as extensively as possible. Fault avoidance is the most widely recognised method of providing reliable software; program proving

techniques are just one way of detecting and removing faults from software. However, complex systems with a large number of components will fail eventually, and the more components, the greater the chance of failure.

2.2.6 Fault Tolerance

Once fault avoidance becomes inadequate as a means of achieving reliability, the designer must incorporate additional components and abnormal algorithms to ensure that occurrences of erroneous states do not result in system failure. Thus the system is tolerant of faults. Fault tolerance includes an overhead in terms of cost, occasionally in terms of performance, and in terms of complexity. However, when the results of a system failure can be disastrous (eg a life-support system in a hospital), or when the number of components is high (thus increasing the likelihood of a component failure), then fault tolerance is a must to increase the reliability of a system. A fault tolerant design can produce a system which is more reliable than its constituent components.

Fault tolerant systems differ with respect to their behaviour in the presence of a fault. In some cases the aim is to continue to provide the full performance and functional capabilities of the system. In other cases only degraded performance or reduced functional capabilities are provided until the fault is removed; such systems are described as having a fail-soft capability.

2.2.7 Fault Tolerance Techniques

2.2.7.1 Protective Redundancy -

The additional components and algorithms which provide fault tolerance constitute protective redundancy. They are redundant in that they contribute nothing to the system during the normal functioning of other components. In an error-free system they have no effect on the internal or external activity, except in so far as they monitor the internal states for errors. However, when an erroneous state occurs for which recovery has been provided, then the so-called redundant components and algorithms come into play.

Two classifications of redundancy can be made, namely masking redundancy and dynamic redundancy. Masking redundancy masks or hides the effect of a fault in a component; as far as the environment of the component is concerned, the component works perfectly, despite internal faults, at least while the masking redundancy is effective. Static redundancy is a form of masking redundancy where all components remain in use in the same fixed relationship whether or not any errors are detected. An example is triple modular redundancy where three identical components are run in parallel and their outputs compared; the three components act as one, and the output is the result of a majority vote on the three sets of outputs.

Dynamic redundancy involves a component providing implicit or explicit indications among its outputs as to whether or not they are erroneous. The internal redundancy of a component is complemented by external redundancy which provides for recovery. Thus a component actually behaves wrongly, but its miscreant behaviour is noted and acted upon.

The KUDOS file-store employs redundancy techniques by keeping multiple copies of files. If one copy of a file is lost, corrupt or unavailable, the algorithms of KUDOS hide this fact. A variant on protective redundancy is used, where if a node in the local area network fails then another node takes on the critical functions of the failed node. The node which takes over might be fulfilling a function already. Redundancy here lies in the capacity of components to take on extra work, though performance may suffer slightly as a result.

2.2.7.2 Error Detection -

Error detection enables system failures to be prevented by recognising when they are about to occur. Ideally, error detection mechanisms should be based only on the specification of the system and be independent of the system itself; otherwise there is the possibility that a single fault could affect both the system and the check, thus preventing error-detection. In practice, however, one is forced to make do with much less rigorous checking than

this.

Ideally, checks on the function of a system will be made on its results immediately before they leave the system. Such checks are based on the specification of the system performance. We call these acceptability checks. Often, establishing acceptability involves replication and checking the results obtained (eg triple modular redundancy). A different kind of acceptability check is to process the results of the system and determine the inputs, then check these derived inputs against the actual inputs (eg multiplying a set of factors to determine the number which they were factored from). However, it is difficult to envisage just how reversal checks could be widely applied since a given result might be caused by a large set of inputs (eg a system which determines whether or not an integer is even).

Complete acceptability checking takes no account of the design of the system. Internal checks are often the only sensible alternative, and these do require some faith in the internal structure of the system. Checks are made on the behaviour of components, and a failure in a component causes some form of recovery to take place before the erroneous state propagates a system failure. The major example of an internal check in KUDOS involves checking the timestamps associated with duplicates of a file; a file with a timestamp earlier than the other copies is in an erroneous state and is corrected by replacement with a copy of the

duplicate with the latest timestamp.

A form of internal check is diagnostic checking. A component is checked periodically for satisfactory performance. Between checks it is assumed that the component is behaving correctly. The checks should approximate to or exceed the demands made in normal use. The trouble with such schemes is that errors might go undetected for a long period while damage spreads throughout the system and beyond. A diagnostic check is used in KUDOS to prompt reconfiguration if a directory-server fails; the directory-server is polled periodically to ensure it still functions - a failed poll generates a new directory-server.

2.2.7.3 Fault Treatment -

A detected error is only a symptom of the fault which caused it. An error could be caused by any one of a variety of faults. The task of locating and removing a fault can therefore be very complex.

One strategy is to ignore the fault and continue to provide a service despite its continued presence, having dealt with any damage it might have caused. Continued usage of a faulty component, though, may only make sense if the fault is effectively transient, for example caused by a rare combination of inputs.

When it is decided to avoid the fault during future operation of the system, it is first necessary to locate it. The search strategy will inevitably be influenced by the internal structure of the system. This can be difficult if the fault caused violation of the intended inter-relationships between components. One solution might be to perform diagnostic checks on all suspect components if an error is found.

Given that a component is known to be faulty, various strategies are possible. Replacement strategies are those in which a previously idle component is directly substituted for the faulty component. Reconfiguration strategies arrange for some or all of the responsibilities of a faulty component to be taken over by the other components which are already in use by the system. Reconfiguration strategies are adopted in a number of places in KUDOS.

Such strategies can further be classified as manual, dynamic or spontaneous. In the first category the system takes no part in the strategy, which in the case of hardware may involve recabling. Dynamic strategies react to external stimuli (ie. the system is informed that a fault has occurred), and use provisions the system contains for reorganising future activity. Spontaneous replacement and reconfiguration are sometimes referred to as self-repair strategies.

KUDOS aims at providing spontaneous reconfiguration, though a failure may result in the temporary, recoverable disruption of user processing.

2.2.7.4 Damage Assessment -

Damage assessment can be based on a priori reasoning, or can involve the system in activity to determine the extent of the damage. Either way, this can involve reliance on the system structure to determine what the system might have done wrongly.

Atomic actions provide a simple method of damage assessment. An atomic action is a set of activities which can be considered logically as a single action, with no information flow between that set of activities and the rest of the system until the completion of the atomic action. This appears in databases in the form of transactions (or logical transactions). The system is dependent on the success of the atomic action as a whole, and consequently atomic actions form a useful notional boundary within which to carry out error detection and recovery. By definition, a fault within an atomic action can have no effect outside the atomic action until completion of the atomic action. The atomic action in which a fault is detected is damaged and must be recovered. However, this approach assumes that atomic actions are well-defined. Unplanned information flow would completely invalidate this approach.

In practice damage assessment is closely involved with error recovery and dealing with faults, and is usually rather uncertain and incomplete. Effort spent in trying to prevent the spread of damage, by careful definition and monitoring of interfaces between components, is well worthwhile.

2.2.7.5 Error Recovery -

Backward error recovery involves first of all backing up one or more processes in a system to a previous state, which is hoped to be error free, before attempting to continue further with the operation of the system or the subsystem. Recovery points are provided, giving a means whereby the state of a process can be recorded and if necessary reinstated.

Atomic actions are a useful tool for enabling backward error recovery. If the state of the system, or those parts changed by the atomic action, are stored before commencement of the atomic action, the state of the system can be restored to that before the start of the atomic action if the atomic action fails. At the end of such a recoverable atomic transaction a decision must be made on whether to commit the atomic action, or to back out if the atomic action included a fault.

Forward error recovery, on the other hand, attempts to make use of the erroneous state to recover the system. It depends to a large extent on being able to identify the fault, or at least its consequences. Generalised techniques for backward error recovery are quite feasible, but forward error recovery must, it seems, be designed as integral parts of the system it serves.

Verhofstad [VERH] lists a number of recovery techniques for databases. These are obviously applicable outside the topic of databases and some are worthy of mention here:

1. Incremental dumping. Incremental dumping involves the copying of updated files into archival storage (usually tape) after a job has finished or at regular intervals. It creates checkpoints for updated files. Backup copies of files can be restored after a crash. The George III operating system, described later, adopts such a policy.
2. Audit trail. An audit trail records sequences of actions on files. It can be used to restore files to their state prior to a crash or to back out particular processes. An audit trail provides the means to back out a process whereas incremental dumping merely provides the means to restore files to previously consistent states.

3. Differential files. A file can consist of two parts: the main file which is unchanged, and the differential file which records all alterations requested for the main file. The main files are regularly merged with the differential files, thereby emptying the differential files.
4. Multiple copies. More than one copy of each file is held. The different copies are identical except during update. Comparison can be done to select an up to date version. This technique provides crash resistance; loss of a copy of a file does not mean loss of the file itself. KUDOS adopts a multiple copy policy for files stored under the directory system. Before accessing a file, all available files are considered, and out of date copies are brought up to date before access is allowed.
5. Careful replacement. The principle of the careful replacement scheme avoids updating any part of the data structure in place. Altered parts are put in a copy of the original; the original is deleted only after the alteration is complete and has been certified. The difference between this and other methods is that two copies exist only during update. The technique is used to provide crash resistance since the original will always be available in case a crash occurs during update. This policy is adopted by KUDOS for updating files.

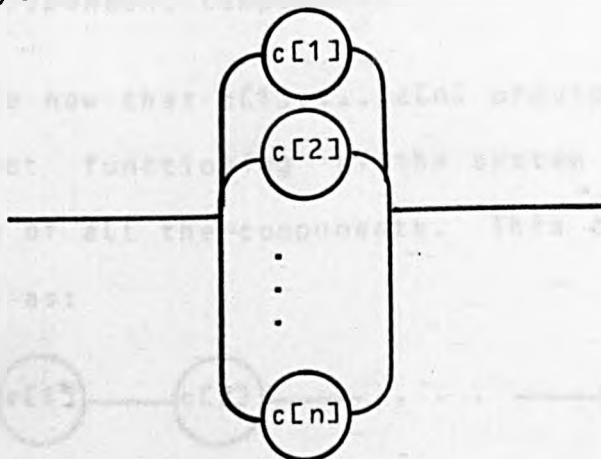
Files stored under the KUDOS directory system are read-only, and to update a file one must take a copy, update that, and proffer it back to the directory system as a complete replacement. Thus the unit of transaction in KUDOS is a complete file. KUDOS need not worry about user processes which do not complete.

2.2.7.6 Component Dependency -

A most critical factor in considering reliability of a system is the component interdependency. A valuable discussion of this is found in [SH00]. If the reliability of individual components is known then the reliability of a system of such components can be calculated.

2.2.7.6.1 Independent Components -

Suppose $c[1], \dots, c[n]$ are used to provide a system such that correct functioning only depends on the correct functioning of any one component. This can be represented graphically:



Suppose $p[i]$ is the probability that $c[i]$ will succeed. Then the system S will succeed with probability:

$$P = 1 - (1-p[1])(1-p[2])\dots(1-p[n])$$

If $c[1], \dots, c[n]$ are identical, each with probability p of success then

$$P = 1 - (1-p)^n$$

and $P \rightarrow 1$ as $n \rightarrow \text{infinity}$.

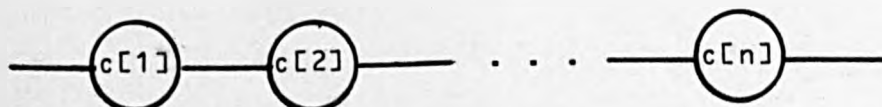
Some figures might help to illustrate this:

$p \backslash n$	1	2	3	4	5
.5	.5	.75	.875	.9375	.96875
.8	.8	.96	.992	.9984	.99968
.9	.9	.99	.999	.9999	.99999
.99	.9999	.999999	.99999999	.9999999999	
.999	.999	.999999	.999999999		

Thus if a component is 99% reliable, then dependence on one of two components produces a system which is 99.99% reliable.

2.2.7.6.2 Dependent Components -

Suppose now that $c[1], \dots, c[n]$ provide a system such that correct functioning of the system depends on correct functioning of all the components. This can be represented graphically as:



Then

$$P = p[1] p[2] \dots p[n]$$

If $c[1], \dots, c[n]$ are identical then

$$P = p^n$$

and $P \rightarrow 0$ as $n \rightarrow \text{infinity}$ for $p < 1$ (ie $c[1], \dots, c[n]$ not totally reliable).

Some figures to illustrate this are:

p\n	1	2	3	4	5
.5	.5	.25	.125	.0625	.03125
.8	.8	.64	.512	.4096	.32768
.9	.9	.81	.729	.6561	.59049
.99	.99	.9801	.9703	.9606	.9510
.999	.999	.998	.997	.996	.995

It is possible, by considering groups of components, as a single component, to derive the reliability of any system whose individual components have known reliability. The calculation is complicated when the reliability of different components is expressed differently. Reliability of components is often expressed as mean time to failure rather than a simple probability of success at any one time; typically a component will function correctly until a fault appears then continue to malfunction until repaired.

2.2.7.6.3 Conclusion -

The lesson to be learnt here is that independence of components is to be sought to improve reliability. Thus, in KUDOS, files are replicated on different volumes, preferably on different nodes. Important too in KUDOS is the fact that access to a copy of a file depends only on the volume which contains that copy. Reliance on the existence of one of two components is remarkably better than just relying on the existence of one component, whilst reliance on the simultaneous existence of two components is remarkably worse than relying on just one of the components.

2.3 CONCLUSION

We have surveyed very briefly certain aspects of reliability theory. In particular, we have discussed the principles and techniques which guided the design of KUDOS, especially independence of components, dynamic redundancy, automatic reconfiguration and careful replacement strategies. We have given a broad definition of reliability, and we shall use this definition later to state precisely what we mean by reliability in the KUDOS file-store. We shall discuss later how reliable file storage has been attempted elsewhere and how KUDOS differs from these.

CHAPTER 3

LOCAL AREA NETWORKS

3.1 INTRODUCTION

Local area networks are very much a fashionable subject for computer science research. A number of universities in the United Kingdom have implemented local area networks for both pragmatic reasons, providing a service to the general user community, and as vehicles for research.

In terms of functionality, local area networks offer little that is new. It is difficult to conceive of an application which has an overriding necessity for a local area network. However, certain architectural features of a local area network based computer system are very attractive. They also introduce a new set of problems, or at least problems in a new context, particularly in terms of process synchronisation and communication.

A local area network can be loosely defined as a collection of inter-connected computers within a small geographic area of (say) one square mile. The computers within a local area network can cooperate on common tasks,

and typically they service the needs of an organisation such as a university or commercial enterprise.

This definition, however, is barely adequate. The central feature of most local area networks is a communication system where nodes can exchange messages at high speed, typically with point-to-point transfer rates in the range ten kilobytes a second to one megabyte a second. The assumption that local area networks contain such a high speed communications system is widespread, and commonly the term "local area network" is taken to mean the communication system itself.

The term "local area network" stems from the fact that such high speed communication systems have not been devised for a network spread over a wide geographic area. Current wide area networks have much more cumbersome and slow means of passing messages, restricting their potential. No doubt we shall see the distinction between wide and local area networks diminishing as techniques for implementing communications improve. We might then simply refer to networks and avoid the distinction.

There are a diverse range of implementations of local area network based computer systems, from ad hoc inter-connections of existing computer services to purpose-built computer systems. A number of such systems will be described later. After defining certain terms we shall discuss the general features of local area networks.

LOCAL AREA NETWORKS

There will then follow a brief survey of a number of existing systems, and finally an examination of the pertinent features of local area networks regarding the design of a reliable file-store.

The choice of a local area network as an implementation vehicle was implicit in the context within which the research for this thesis commenced. It is contended, however, that the problem and the solution are somewhat more general. For example, the notion of associated directories described later could well be applied profitably to a stand-alone system based on single or multiple processors in order to control redundancy and improve accessibility of files.

3.2 DEFINITIONS

3.2.1 Communications

The central feature of a local area network is the communication system. This is usually a high bandwidth system such as the Cambridge Digital Communication Ring [WILK1], or Ethernet [ALMES], which allow direct communication between any two computers on the local area network system. A variety of such communication systems is reviewed in [PENN]. However, some local area network systems have developed differently, such as DECNET [SELIG] which is a message-switched system (a message between two computers may be routed through a number of intermediate

computers), or a number of advertised micro-computer local area networks which are all linked to and communicate via a single shared disc.

We shall restrict ourselves to considering high-bandwidth systems such as the Cambridge ring or Ethernet. To illustrate the area of communications we shall briefly describe the Cambridge ring. This will be useful later, since KUDOS was implemented using the Cambridge ring and some discussion of its performance will take place.

3.2.1.1 The Cambridge Ring -

The Cambridge ring differs from most other local area network communication systems in its method of transferring data. Usually a wire is dedicated to a single transmission between two nodes for as long as that transmission takes, up to some upper bound; a transmission consists of a sequence of bytes of data sandwiched between control information pertinent to the network; nodes wishing to transmit must contend for the wire, and this can be done by passing a token from node to node giving access permission, or by listening until the wire is silent. Ethernet is a prime example of such a network which listens for a silent wire.

The Cambridge ring adopts a much different policy. A train of objects called packets moves cyclically from node to node. A packet can either be empty or contain 16 bits of data. A further 22 bits are used in each packet for control

information. One bit indicates whether a packet is full or empty; if the packet is full 8 bits are used to indicate the sending node and 8 bits to indicate the receiving node. When a packet is used to send data it returns to the transmitter with an indication that the receiver either accepted or rejected the packet, or whether the receiver was busy, or whether the receiver did not exist and the packet was ignored - this uses 2 further bits. Three other bits are used for parity checking and hardware level packet administration. Thus 38 bits are used to transmit each 16 bits of data. At 10 megabits a second transmission rates this means that a packet passes a node every 4 microseconds approximately.

The transmitter when it has data ready must wait for an empty packet, fill in the source, destination and data bits and mark the packet as full. The transmitter waits for the packet to return and checks whether it was accepted, rejected, ignored or the receiver was busy. If a packet was marked rejected or busy then the transmitter can inform the sending process or automatically retry. If automatic retry is used, the retry is delayed by the transmitter by a period twice the time it takes a packet to get round the ring; with third and subsequent retries this delay is increased to sixteen times the time it takes a packet to get round the ring. This prevents repeated retries saturating the ring.

A receiver can accept any packet sent to it, or choose to listen to only one particular node by setting a source select register. If the source select register is set to a different node from that of the sender of a received packet, then that packet is rejected. The source select register is useful for block transmissions, allowing a process to receive data from one source at a time, rather than have a number of blocks interleaved. If the data register has not been emptied at the receiver when another packet arrives, that packet is marked as busy.

3.2.2 Protocols

Most communication systems provide an interface which is less than ideal. The Cambridge Ring in particular is of little direct use to an application program. The assembly of a stream of packets for transmission, followed by collection and collation of such a stream of packets to receive a message would be tedious and error prone.

To overcome this, processes usually communicate using protocols which are a set of routines which simplify the use of the communication service and add extra features. For example, a basic block protocol [JOHNS] is used on the Cambridge Ring to handle the disassembly of a block into packets for transmission across the network and reassembly into blocks on receipt. It also prevents two blocks colliding at a receiver and causing a garbled block to be constituted out of two sensible blocks. A block is checked,

using a check sum appended to a transmitted block, to reduce the likelihood of a received block containing an erroneous packet. Thus the basic block protocol permits processes to communicate using whole blocks of information and tries to ensure that blocks are received as transmitted without communication faults causing errors.

The ability to transmit blocks, however, is still very basic. More protocols can be implemented to provide a higher level of service for processes, for example the Byte Stream Protocol for the Cambridge Ring [JOHNS]. Such a service might provide the ability to create and maintain a communication route between any two processes in a network, and ensure that messages arrive at a process in the same sequence as they are sent, and that no message is lost.

3.2.3 Homogeneous And Heterogeneous Systems

A homogeneous system is one in which all nodes are of the same architecture. However, individual nodes may have different peripherals attached and be programmed for specialised functions. An advantage of homogeneous systems is that potentially a program can run on any node, thus reducing the programming effort and possibly permitting a process to move from node to node during the course of execution.

A heterogeneous system is one in which the nodes are of a number of different architectures. This is likely to evolve within an organisation which already possesses a number of independent processors. A local area network is a development which allows these independent computers to provide a more general service with easier sharing of vital resources. For example, in a university the owner of a small 8-bit micro-computer might wish to use the facilities of the central mainframe to store large quantities of data and for printing.

A homogeneous system is more likely to evolve when a local area network system is designed from scratch. However, there is a strong argument that a local area network should be able to absorb other types of computer, say for a specialist application. Heterogeneity involves more careful interfacing of nodes, since the internal structure of a node cannot be implicitly assumed. A number of manufacturers are offering homogeneous local area network systems as an alternative to mainframe-based time-sharing systems, for example the Apollo Domain system [APOLLO].

3.2.4 Autonomy

An autonomous node is one which can function independently of the rest of the local area network. A node can be autonomous in varying degrees; for example a single node might be able to provide a user with processing and data storage, but rely on another node to provide printing

services. Autonomy is an important consideration for a reliable system. If a node is highly dependent on another node, a failure in either node can effectively cause the first node to fail.

Many local area network architectures depend greatly on single components, for example the monitor on the Cambridge Ring, repeaters on a loop network, single name-servers. A sensible architecture should provide scope for high autonomy of nodes, reducing or removing dependence on any single component. Where dependence on a single component is inevitable, techniques should be used for increasing the reliability of that component.

An autonomous node might depend to some extent on the services of other nodes on the local area network, but might be free to determine how to exploit those services depending on the state of the local area network. Autonomy, therefore, we understand as an imprecise, intuitive term. A common way of describing an autonomous node is to say it is loosely linked with the system.

A major design principle for KUDOS was autonomy of components, allowing scope for reconfiguration of the system in the event of a failure.

3.2.5 Servers And Clients

An important notion in local area networks is that of a server. There is scope for dedicating a single node to a single function. Such a node is often called a server. Examples are compiler-servers, file-servers, printer-servers.

Most local area networks utilise servers in some way. Within KUDOS the idea of a server is used more generally to mean a process dedicated to a particular function. Thus a single node could handle a number of servers. Examples of this in KUDOS are directory-servers and file-servers, which can coexist on the same node.

The notion of a server also extends to the notion of client-server systems. A client is any process which utilises a server. A client of one server might well be a server itself. For example, in KUDOS a directory-server is a client of the file-servers in the system.

Within a system the interface between clients and servers is an important consideration. In particular, we shall be concerned with the level of dependence a client has on a server. For example, what happens if the server fails. Equally important is the dependence of a server on a client. For example, how would a file-server cope with an "open" file if the client fails or inadvertently omits to close the file.

3.3 PROS AND CONS OF LOCAL AREA NETWORK SYSTEMS

It is useful to consider local area networks as a form of computer system architecture rather than an ad hoc collection of distinct computer systems. If local area networks are to be considered as anything other than an academic diversion, they must be viewed as an alternative method for solving problems for particular applications.

Systems based on a single cpu have had considerable success over the last two decades. Many such systems are well developed and have been programmed to meet the needs of a great number of applications. The problems and limitations are well understood. Moreover, the achievements of VLSI technology mean that single cpu systems can be mass-produced at very low cost.

Arguably, a single cpu system will provide a more than adequate solution to most applications. What features, then, does a local area network based system provide which make them worthy of consideration.

Firstly, there is the capacity for expansion. Notionally at least, a local area network can be extended easily with the minimum of fuss and effort. If processing power is in short supply, one simply adds another node. In a similar manner one ought to be able to add peripheral and storage devices.

However, ease of expansion is not a natural consequence of a local area network. There is the opportunity for allowing easy expansion in a local area network, but the designer of a local area network based system must be fully aware of the need for ease of expansion and the pitfalls to avoid. In particular, the binding of objects to processes needs particular care. More will be said about this later, but for example if a program has embedded in it fixed network addresses then alteration of the network can become difficult, requiring changes to programs if objects are moved. A typical means of avoiding this tight binding of objects to programs is a name-server such as that described in the Cambridge Model Distributed System (see later). KUDOS uses a scheme called resource directories.

Expansion has other hidden constraints, in particular the communications system. Whatever communication system is used, saturation is inevitable if expansion continues unabated. It is important to consider the behaviour of the communication system under heavy demand. A designer would do well to consider the communications as a limited resource. For example, if broadcasting is frequently used, say to locate resources, then the number of broadcast messages is likely to grow rapidly with the size of the network, and could become a problem very quickly.

A second advantage of local area network systems is the performance benefit of providing cpu's on a per-user basis, especially where there is a demand for interactive applications, and more especially if the interactive work involves graphic displays. A problem of time-sharing systems is the dramatically variable response to an individual user depending on the loading of the system. A personal computer, integrated with a local area network, ought to be able to satisfy most user's processing requirements.

On the other hand, the use of personal computers introduces the problem of data sharing and inter-user communication. These problems are by no means insurmountable, but require a different solution to the ones traditionally used in single cpu systems.

Cost is frequently advertised as a benefit of local area networks. However, in the present topsy-turvy world of computer prices, it is difficult to argue for or against local area networks as a least expensive solution. For a given application, a single cpu based system might well offer the cheapest solution. A local area network, however, might offer longer term benefits since upgrading the system could well be done by simply adding new nodes rather than replacing the whole system, which might well be cheaper and probably easier.

A significant motivation for the development of a local area network in an established computer environment is the ability to draw together existing computer resources to provide a more or less unified service. University campuses are a significant application area in this respect. Individual departments often have their own specialised computer equipment as well as access to a centralised shared resource. Linking individual departments and the central resource provides a number of benefits. For example, the central resource might provide expensive peripherals such as graph-plotters which are outside the budget of individual departments. There might also be benefits in departments accessing each others resources for inter-disciplinary activities, or on a more mundane level for administration purposes.

Finally, local area networks provide scope for designing reliable systems. However, it must be said that local area networks also provide scope for designing potentially very unreliable systems. Within a local area network there are a large number of components. The inter-dependence of these components determines the likely effect of a component failure.

On the one hand, local area networks provide scope for dynamic redundancy. For example, if a printer-server goes offline a process might be able to use another printer-server elsewhere on the network. More pertinently, if a file-server is offline, it may be possible for a

process to locate another copy of a file on a different file-server.

On the other hand, heavy dependence on a component or set of components can seriously impede the functioning of the system if such a component fails. For example, if a single name-server is provided and all objects are located through that name-server, then the loss of that name-server could stop all processing in the system.

We see then that local area networks provide a designer with certain potential benefits on performance, cost, expansion and integration of services. A number of approaches to local area network system design are already manifest, and some of these will be discussed in the next section.

3.4 SURVEY OF EXISTING LOCAL AREA NETWORK COMPUTER SYSTEMS

3.4.1 Cambridge Model Distributed System

This system, described in [WILK1] and [HERB], takes the view that processors and other resources should be banked together and remote users allocated resources from a pool. It is seen as an alternative to the "computer in every office" approach. Justification for this approach is given as ease of maintenance, convenience (computers can be noisy, bulky and hot), economics (the system can have fewer computers than people), and flexibility (specialised machines can be shared among many users).

On logging in to the system a user is allocated a dedicated processor from the pool. Two critical components of the system are the Resource Manager and the Name Server. The Resource Manager deals with allocation of processors to clients, and with management controls. The Name Server provides a mapping from names to network addresses. Both these components are critical to the functioning of Cambridge model, and are implemented on Z80A micro-computers.

Other servers on the net are a file-server, a printing-server, a boot-server for bootstrapping an allocated processor and a time-server. An editor-server has been considered, though editing is usually done by an allocated processor. Access to the network is through a terminal concentrator, which can connect up to four terminals to the network.

The Cambridge model is based on the Cambridge Digital Communication Ring, described above. Data is sent one 16-bit word at a time. A number of protocols are used to provide block transfer (Basic Block Protocol - BBP), fast control messages (Single Shot Protocol - SSP), and a transport level service with virtual circuits (Byte Stream Protocol - BSP) [JOHNS].

A number of commercial products have stemmed from the Cambridge model and from the Cambridge Digital Communication Ring - e.g. Logica's Polynet [LOGICA]. However, the view

that computers are unpleasant physical objects is rather pessimistic; miniaturisation has reduced heat dissipation, and the need for fans.

Furthermore, the demand for graphics as a widespread tool will make the barrier of a network between terminal and processor too restrictive. On cost we should soon see powerful systems available very cheaply.

On reliability, the Cambridge model relies on a statically located Resource Manager and Name Server. Failure of these cause system failure. The Cambridge Digital Communication Ring also raises concern - it is dependent on a single station, called the monitor, which maintains the packet structure, and on each repeater, and a break in the loop will bring down the ring.

Some concern does exist about the ring as a fast means of communication (see [BRER]). Some simulation work was undertaken by the author early on in the KUDOS project (see [LUNN1]). Only 40% of the bandwidth of the ring is available for data, the rest being used by the ring logic for addressing and error detection. Although a node-to-node transfer rate of 250 kilobytes per second is possible on a 10 megabit ring, contention and protocols can reduce this in practice to the order of 10 kilobytes, and on an interrupt-driven interface rates of around 3 kilobytes can be experienced. More discussion on the Cambridge Ring will follow, since this has been used as the communication system

for KUDOS.

However, given that it is a very practicable system, we are likely to see variations on this scheme gaining popularity. A compromise on the personal computer, with some users having processors located at their own site, would make such a scheme very attractive. Some assurances on the reliability of various components, either by redesign or guarantees on critical components, would make this a very useful system.

3.4.2 Xerox Ethernet

The Xerox Palo Alto Research Centre have a number of local area network research projects based on an Ethernet. These have culminated in the Xerox Star [SMITH] computer system which is an interesting and novel product aimed at the automated office market. The Ethernet is a Carrier Sense Multi Access network, connecting nodes on an unrooted tree. Ethernet appears neater than the Cambridge Ring since there is no dependence on a monitor and whole block transfers can take place without irritating disassembly/assembly into sequences of words; thus almost the entire bandwidth of the network can be used. Much more is known about the behaviour of Ethernet under heavy loading [ALMES] than the Cambridge Ring.

A number of projects have exploited Ethernet. For example, Violet [GIFF1] and its associated file-store [STURG]. A client-server approach is taken. A client is a process which acts, either directly or indirectly for a user. A server is a process which acts (provides a service) for a client. In this way a server can itself be a client. The basic communication is for a client to send a request to a server, and for the server to send a result. A client can either issue a request and wait for the result, or issue a number of requests and accept results as they return (not necessarily in the same order as the requests).

A client "knows" about the structure of the network and server, in the sense that it must be programmed to exploit the architecture of the system. Decentralised control in this way means that individual computers can be specialised for different tasks, and the impact of malfunctions is reduced.

Here we have a more flexible approach. Particular applications can exploit the network as they wish. Some services, such as a file-store (see [GIFF1]), are provided for public use. More rigid systems can, if so desired, be built on top of it. It appears to recognise what could be a useful guideline to distributed computer system design: provide a good communication mechanism and a few services then let the users exploit this as they see fit.

Here we have high autonomy of individual nodes, in contrast with the Cambridge model where there is high interdependency of nodes with certain critical nodes such as the name-server determining the correct functioning of the whole system. KUDOS seeks to emulate the high autonomy and the client-server mechanism evident in the Xerox projects.

3.4.3 Apollo Domain

Apollo Computers [APOLLO] have introduced a network based system called the Apollo Domain Architecture. The network is a token passing ring system. Nodes are based on a Motorola 68000 micro-processor, with up to 1 megabyte of main memory, integral Winchester disc, optional peripherals such as printers, and a high resolution display system.

Nodes are highly autonomous, but can interact. For example it is possible to access the file-store of a remote node by specifying its full network-wide name. Objects, such as files or peripherals, have a 96 bit system-wide address, and can be accessed from anywhere on the network. Presently it is not clear from the documentation how this powerful system will be exploited, and it appears as a linking together of high powered personal computers, rather than a fully integrated system design.

The basic inspiration for the software is Unix, with other features such as multiple virtual terminals (see [LANT]). In all, this is a very ambitious project,

especially considering the short timescales involved.

3.4.4 Z - Net

Z-net [BENHAM] is a network of highly autonomous nodes based on Z80A 8-bit micro-processor. Each node has a single simple, non-distributed, single-user operating system. A number of servers are provided on the network. The network is described as "Ethernet-like".

Apparently the small size of the nodes caused software development problems. This problem occurred in developing KUDOS. It is arguable that the overheads of interfacing an 8-bit micro-computer to a local area network system are too high, especially if that micro-computer is to provide a general service. This problem should diminish with 16-bit micro-computers as nodes, and Zilog are experimenting with the Z8000 16-bit micro-processor. Nevertheless, Z-net has appeared on the market.

3.4.5 Dec Net

Dec Net is a family of packet-switched networks produced by Digital Equipment Corporation. It is remarkable in that it is message-switched. It seems unlikely, however, that message-switching will have significant impact on local area network development, considering the availability of digital communications systems such as Ethernet and Cambridge Ring. The point to be made here, however, is that

the communication mechanism should not dictate the local area network system design. It is a principle difficult to adhere to, but KUDOS could well have developed on top of a message-switched communication system.

3.4.6 Unix Satellite Processor System

This network described in [LYCK], is a star network based on the DEC PDP-11 series. A central processor runs a version of Unix and satellite processors are linked to the central processor via a collection of serial lines. Satellite processors do not have their own operating systems, but rely entirely on the central processor.

The interface between satellite processor and central processor is at the level of system call. A program running in the satellite processor which issues a system call has that call trapped and routed to the central processor. In this way a program on a satellite processor has access to the services of a Unix operating system.

One use of this system is to develop stand-alone systems. More relevant to this thesis, it can also be used to provide a user with the real-time capabilities of a dedicated mini, together with the level of service of a Unix time-share system. Another possibility, discussed in [LYCK], is that a powerful satellite processor might be used by the central processor for running compute-bound programs, leaving the central processor to service system calls.

Such a system, one might argue, is not "truly distributed". However, variations on this scheme are likely to evolve, meeting the needs of certain styles of organisation. A satellite processor may be implemented as a capable of stand-alone performance, reducing the dependence on the central processor. Moreover, the central processor is an obvious bottle-neck and critical component, so that improvement of reliability and performance can be concentrated there.

One important point, most recognizable here, is the need for a communications system and set of services as the core of a network system. Whether these are provided in a central processor or as a number of services on a network is arbitrary.

3.4.7 Other Distributed Unix Systems

Coinciding with the production of this thesis, a number of distributed Unix systems have appeared. The author is also aware of as yet unpublished work at York University [TOML] and Strathclyde University on distributed Unix. The existence of such distributed developments is an interesting indication of the maleability of the Unix operating system. The clear definition and good judgement in the choice of the system calls, and the implementation of all but a small part of the operating system in a high level language have encouraged the use and adaptation of Unix.

Cocanet [LAWR] modifies the Unix kernel to trap service requests for remote resources. Inter-process communication is modified to facilitate use of a network. Minimal changes have been made to the Unix kernel interface so that major changes to existing Unix software are unnecessary.

The Newcastle Connection [BROWN] adopts a slightly different approach. A layer (the Connection Layer) is placed between the kernel and the user software which filters system calls and redirects requests to remote resources. The user software sees an apparently normal single-system kernel, and the kernel itself needs no special modification, apart from a driver for the communication network. "Network awareness" is restricted to the connection layer.

The distributed Unix described in [LUDER] is unusual in that it connects a number of individual Unix systems to a pool of Unix-derivative file-servers through a circuit-switched communication system. This differs from the above approaches in that the user-processors have no direct way of communicating apart from through shared file-servers.

3.5 CONCLUSION

This brief discussion and survey has indicated a variety of approaches to local area network design. The KUDOS approach stems most directly from the Cambridge Model

Distributed System and the Xerox approaches.

It is the author's belief that a greater diversity will appear over the next few years. "Local area network" seems to imply, at present, a basic communication system such as the Cambridge Ring or Ethernet. However, it is important not to discount such approaches as the Unix Satellite Processor System. Cheap digital circuit-switched telephone exchanges might also have a significant impact.

Whatever the underlying communication system, local area networks will play an important role in future computer systems. We shall finish this section by indicating a number of guidelines which underlie the design of KUDOS.

Naively we might expect a local area network system to provide the performance per-user of a dedicated mini, the level of service and sharing of a large time-sharing system, greatly improved reliability, incremental growth, and flexibility. These are the expressed aims of the Apollo Domain Architecture [APOLLO].

On performance, the route which seems most profitable is the personal computer, located near to the user. The Cambridge System is based on the view that computers should be grouped together away from users, but if a high level of service is expected by a user, for example high-resolution graphics, then placing a network between user and computer is a retrogressive step. Even a high-bandwidth communication system is unlikely to cope with the data flow

between terminal and main processor for interactive graphics. It may be necessary to place certain specialised and expensive computers remotely, but there are distinct advantages in placing the routine services and user interface as close to the user as possible.

Essentially, if a resource is cheap and there is no functional justification for sharing, then that resource should be provided on a per-user basis. Time-shared systems evolved around the fact that processing power was an expensive resource. That is no longer the case.

On service, we need to provide a group of users with a pool of resources, say a shared file-store and peripherals such as printers, plotters and tertiary storage. Some of these are too expensive to provide and maintain on a per-user basis; a file-store must be shared if it is to contain other than purely personal data. How they are provided is arbitrary: in the Cambridge System it is via a pool of independent servers across a network; in the Unix satellite system it is provided by exploiting the services of a central time-sharing operating system.

Reliability is a crucial issue in a local area network. In some local area networks the number of critical components is high. For example, the Cambridge system relies on the Ring monitor, the resource manager, the name-server, and the boot-server. Failure of any one of these seriously degrades or incapacitates the system.

Reducing dependence on critical components should be a crucial aim in local area network design. We want high autonomy of nodes. The personal computer approach goes some way to achieving this.

Incremental growth should be a natural consequence of local area network architecture; adding nodes should be a simple and relatively inexpensive operation up to some clearly defined limit. This can, however, be crippled by bottlenecks on the communication system and on various servers. Much more information is required on the comparative performance of communications systems such as Ethernet and Cambridge Ring. It is not enough to say that there is plenty bandwidth available. If networks are to grow larger, it is necessary to know at what point will a decay in service occur.

Local area network systems must be designed with incremental growth in mind. Thus demand on performance-critical aspects of the system must grow linearly with the size of the network. For example, if broadcasting to all nodes is a commonly used technique, then the processing needed to handle broadcast messages grows proportionally to the square of the number of nodes. Thus broadcasting should be avoided for all but small systems, or in rare and extreme cases such as error recovery.

Applicability of a local area network seems to imply a heterogeneous system. The architecture of a node should be chosen to fit an application, rather than to pander to network constraints. However, it may be sensible to choose the same architecture for a number of similar servers. This would save on programming effort and help to attain standard interfaces.

The KUDOS design attempts to provide these properties. Alternative designs may give a different emphasis. For example, the Cambridge model aimed to minimise cost and ease maintenance of hardware. The properties of a system should be largely determined by the original design aims. The design aims of KUDOS will be presented in chapter 6.

CHAPTER 4

FILE-STORES

In any computer system secondary storage performs a key function, as a repository for data, as an extension of main memory, and as a means of inter-user communication. The user-view, how it performs, how reliable it is, are critical to the success not only of the file-store, but of the whole system.

We shall begin this chapter by defining certain terms and then discussing certain issues in file-store design. We shall then examine a number of existing file-stores and conclude with a discussion of existing approaches to reliable file storage.

4.1 DEFINITIONS

4.1.1 File-stores, File-server And Files

We take this opportunity to define "file-store" for the rest of this thesis. A file-store will be considered as a repository for data, providing a mnemonic (user-arbitrary) naming scheme for files, with some means of protecting data

from unauthorised update and/or interrogation, plus some method of allowing clients to ensure consistent update of data. A file will be taken to be a user-arbitrary sequence of bytes which is stored by the file-store under a given name, and not interpreted by the file-store.

Further, a file-server will be taken to mean a repository where files can be stored, and which provides an index address for files contained in it (e.g. i-nodes in Unix, or UID's in Xerox DFS - see later). This address may contain authorisation information. A directory-server will provide a mapping from user-arbitrary mnemonics to the file-server index, and provide data protection, perhaps using any authorisation facilities provided by the file-server.

4.1.2 Database

A database implies a much more complex and explicit relationship between collections of data than a file-store. A database interprets the data to a large extent, whereas a file-store imposes relatively few constraints on the data content of a file.

Three general structures for database have gained wide acceptance, namely hierarchical, network and relational. The former two structures involve explicit pointers, allowing a user of the database to navigate around the data. Relational databases contain no such explicit pointers, but

the structure of data held is restricted and operations on the database involve reference to actual data content.

This thesis does refer to work on databases. To a large extent the problems of distributed database are similar, but require different approaches. In a sense, the designer of a file-store has much less to worry about.

It was decided at the beginning of the Keele distributed file-store project not to consider the problems of databases as such, except in so far as they were relevant to file-stores. This was possibly a wise decision since it narrowed the aims of the project so that they could be tackled with the limited resources available. However, it may be worthwhile extending some of the results of the research presented here to databases. More will be said about this later in the thesis.

4.1.3 Naming

The naming of files in a file-store is an issue of great importance. It is desirable to give individual users a flexible and adaptable naming scheme. This section introduces some basic concepts and definitions, and in a later section we shall examine some of the issues related to a file-store. A useful source is [SALT].

Firstly, we take a name to be an identifier, such as a character string or integer, used to refer to an object. Clearly an object can have many names; this is useful since an object can be referred to in many ways; for example it is often beneficial for the system software to use a short integer identifier (ie name) for an object, whilst a user would find such a name difficult to remember. It is also possible for a name to refer to many objects, for example a name may refer to a file which has multiple copies, though this can lead to ambiguities which must be resolved in practice.

By a context we take to mean a mapping from a set of names to a set of objects. A common example is a directory in a file-store which provides a mapping from file names to files. A sensible restriction on a context is that a name maps to only one object, and we shall assume this. It might, however, be sensible to permit two or more names to refer to the same object in the same context.

A context might well map a name to another context. For example, a directory in a file-store may name other directories, often referred to as sub-directories. In this case we have a naming network. This allows reference to an object indirectly via a path name. A path name is a sequence of names, where all but the last name is a name of a context; the object referred to by a path name is the object named by the last name in the path-name in the context derived by removing the last name from the initial

path-name. This is illustrated by the Unix file-store (a description of which will follow later), where `/usr/fs/ken/tmp1` refers to the file `tmp1` in the directory `/usr/fs/ken`.

If there is a particular context in a naming network from which all objects referred to by contexts in the naming network can be named by a path-name then that context is called a root. Unix has a single root, but more than one root is conceivable; indeed any context which refers to a root is by definition itself a root. If a root exists then a naming network can itself be considered a context where the names are path-names stemming from the root, and the objects mapped are those mapped by the path-names.

By an address we take to mean a name which is system generated. Intuitively, an address refers to the location of an object, but this is not always the case. An example of an address in Unix is an i-node number, which refers to a file on a disc. The i-node number itself is not sufficient to locate the data in a file and it is necessary to use the i-list as a mapping from i-node to blocks of data on the disc. The i-node number is not usually referred to directly by a user, but is obtained from a directory which maps names to i-node numbers.

In practice the implementation of a context, such as a directory in a file-store, may not provide a direct mapping from name to object, but from name to address (which is but

another name). That address may have to be interpreted in another context in order to access the object. For example, Unix directories map names to i-nodes, which in turn map to files.

Finally, there is widespread use of the idea of a unique identifier. A unique identifier is a name in a single global context which names all objects in the system. Its advantage is that it provides an unambiguous way of identifying any object in the system. Typically a unique identifier is a fixed length integer or bit-string with a range of values chosen to be large enough to exceed the number of objects ever likely to be created by the system.

4.1.4 Protection

In a system which is shared among a number of users, the issue of data protection is important. Individuals should be limited to what data they can access and how. This is to prevent reading of confidential data, to prevent inadvertent update of another persons data, or in extreme circumstances to prevent malicious update of another persons data.

4.1.5 Mutual Exclusion

In a system where a number of processes update shared data concurrently, there is a requirement that updates proceed in a sensible manner. The classical example used to

highlight the need for mutual exclusion is the bank balance update. Suppose two processes A and B update a bank balance X by adding a and b respectively. There are a number of ways this might happen, for example:

- 1) A reads X
 A adds a to X
 A writes X
 B reads X
 B adds b to X
 B writes X
- 2) B reads X
 B adds b to X
 B writes X
 A reads X
 A adds a to X
 A writes X
- 3) A reads X
 B reads X
 A adds a to X
 B adds b to X
 A writes X
 B writes X
- 4) B reads X
 A reads X
 B adds b to X
 A adds a to X
 B writes X
 A writes X

The resulting value of X for these four instances is:-

- 1) $X+a+b$
- 2) $X+b+a$
- 3) $X+b$
- 4) $X+a$

Clearly the first two are correct, the latter two incorrect.

To ensure the correct processing of data it is necessary to ensure that the actions of A and B exclude each other in time, at least during the update of X. In [BRIN1] and [BRIN2] we find a number of mechanisms for mutual

exclusion discussed.

The most appropriate form of implementing mutual exclusion on files is some form of locking mechanism. If a file is locked then only the issuer of a lock can access that file.

It is useful to discriminate between read and write access. A read lock permits inspection but no update of a file. A write lock permits update of a file. It is common to allow a file to have a number of read locks or just one write lock but not both.

Some systems permit locking of part of a file only. This is necessary for database applications. We take the view (perhaps naively) that locking of whole files is adequate for non-database applications.

4.1.6 Deadlock

Locking presents the problem of deadlock. Two clients are said to be deadlocked if each is waiting for action by the other in order to proceed. For example, two clients may wish to lock files A and B. If simultaneously client 1 locks A, client 2 locks B, then client 1 tries to lock B and client 2 tries to lock A, neither can proceed unless the other relinquishes its first lock. More complex forms of deadlock can occur, involving a number of clients waiting for each other.

A system must either prevent deadlock, or detect deadlock and recover from it. In a distributed system where knowledge of the complete state of the system is difficult to obtain, deadlock detection is difficult. Deadlock prevention is therefore the most attractive option.

[BRIN2] discusses a number of deadlock prevention schemes. For example, if all resources are ordered hierarchically, and a client is restricted to locking only those resources higher up the hierarchy than those it has already locked, and if all clients guarantee to release locks within a finite time, then it can be proved that clients will not deadlock. However, such an ordering of resources is difficult to enforce on a file-store, more especially a distributed file-store. Some form of deadlock prevention or detection must, however, be provided if locking of files is permitted.

4.1.7 Consistency

A file-store (or database) is said to be consistent if it satisfies a set of conditions called consistency constraints. These constraints are arbitrarily chosen, but intuitively state that the file-store (or database) behaves sensibly. Some examples of consistency constraints will be given later. Consistency constraints can be considered part of the specification of a file-store.

It is often the case that a file-store (or database) must pass through inconsistent states. For example, a transfer of money between accounts may leave a ledger file inconsistent between the debiting of one account and the crediting of the other.

To overcome this, operations on a database are usually grouped into "logical transactions" which are similar to atomic actions with the added property that they transform the database from one consistent state to another.

4.1.8 Atomicity And Transactions

Atomicity is taken to mean the property that an operation succeeds completely, or has no effect at all. In practice, this is difficult to achieve. For example, a disc write may not succeed, but may write incorrect data through some electrical fault. Techniques are required to ensure that either an action completes successfully or that any alteration made by the action is undone.

A transaction is a group of operations which, taken together, are atomic; ie the whole group of operations succeed or no effect is apparent. More properties are often associated with transactions, especially in databases, such as the provision of mutual exclusion and preservation of consistency constraints.

The point at which a transaction completes successfully and relinquishes the ability to roll back and restore the states it changed is called the commit point. Before commit, the effects of the transaction, its internal state and partial results should be invisible to the rest of the system; this may mean that other transactions may have to wait for the resources used by the uncommitted transaction. After commit, the results of a transaction are eventually made visible and resources freed for use by other transactions.

Transactions may be nested; a transaction may consist of a number of concurrent sub-transactions. One technique for implementation is two-phase commit, which allows a group of transactions to operate as one. A coordinating process, to commit the group of transactions, issues a "ready to commit" request to all transactions in the group. Each transaction replies with a ready, or with failed; ready transactions enter a ready state in which no more operations can be carried out. If the coordinating process receives a "ready" reply from all the transactions then it issues a commit request to all transactions; otherwise it issues an abort to all transactions.

4.2 ISSUES IN FILE-STORE DESIGN

4.2.1 Data Placement

A serious problem in local area networks is where to place data and what algorithms to use. A number of issues arise, and the designer must choose a scheme appropriately.

Firstly, on performance, it is intuitively obvious that the "nearer" data is to a process the faster that process can access that data. However, physical proximity is probably not an adequate definition of nearness. On a local area network which is fast enough, accessing a file on a remote node may be little slower than accessing the file on a local node. Certainly, there should be no difference in accessing two files on two similar remote nodes, at least with a communication system such as Cambridge Ring or Ethernet. Where the local area network provides significant delay either on latency (ie time to service an operation) or transmission rate, then a distinction between local and remote is useful.

An aspect of performance, not considered in this thesis, is loading on a node which contains a file-store. If two copies of a file exist, it is clearly advisable to access the file via the file-server with the least loading.

On reliability, under a multiple copy scheme, it is necessary to place files in order to minimise likelihood of loss and to maximise the availability. There is clearly no sense in placing two copies of a file on the same volume if it is possible to split them across two volumes. If volumes

are on distinct nodes then all the better.

4.2.2 Consistency

There are a number of consistency constraints which one might place on a file-store in a local area network:

1. Any two copies of the same file shall appear to be identical to the user;
2. Any process which does not successfully complete an operation on a file shall have no effect on the contents of that file;
3. Between user generated operations on a file, the contents of that file shall not appear to change;
4. Two independent processes operating on the same file concurrently will have the same effect as if one process had completed operations on a file before the other commenced.

The first constraint is clearly vital. At worst, a user process ought to be informed if two copies of a file are not identical, but then the process is left with a difficult decision. The second constraint implies some form of recoverable atomic action or transaction. The third point might well be an integrity constraint (ie the system models the real world). The fourth point again implies some form of atomic action or transaction.

The second and fourth constraint sound to be of overriding importance. It is interesting to note that a number of file-stores do not guarantee either, especially Unix. Preserving consistency in a file-store may often involve some form of transaction; KUDOS provides this in a very limited way for operations on a single file.

4.2.3 Shared Access

Users need to share data storage for reasons of economy and consistency. Users may not be able to afford private storage for all their personal data, and a number of users may wish to interrogate and operate on the same file. Sharing a file-store, however, leaves two important issues to be resolved, namely the protection of data from illegal access or update, and the control of concurrent access so that the result of multiple operations is sensible.

The question of data protection is application dependent. Technology will probably lead to a situation where personal data is physically held separate from shared data. After all, if the cost of personal storage is low enough why put personal data in a shared file-store - shared file-stores should be for shared data only. Meantime, however, some sharing of storage for personal data is to be expected.

The problem of confidential shared data is much more difficult. Some form of access restriction is necessary for this. For highly confidential data, some form of encryption may be necessary, but this should probably be a function of client software, not the file-store.

Protection should not go so far as to restrict the services of the system. Moreover, elaborate protection mechanisms offer a challenge to certain types of user. A simple, but effective scheme is desirable, flagrant misuse being prevented or penalised outside the system. Protection against accidental rather than malicious damage is more important and easier to implement effectively.

4.2.4 Shared Update

The constraint that two independent processes operating on the same file concurrently will have the same effect as if one process had completed operations on a file before the other commenced, requires some method of ensuring mutual exclusion. The usual mechanism used in file-stores and databases is a lock. Before a process accesses a file, it must lock that file, and on completion of the activities involving that file, unlock it.

One problem with locks is that of deadlock, discussed earlier. Another problem with locks is how to deal with a process which does not relinquish a lock either by oversight or because that process fails. With a single controlling

executive the problem is easily solved. However, in a loosely-coupled system, the detection of such a problem is not easy and must rest to a large extent with the server of the file (or indeed any other resource) which is locked.

4.2.5 Naming

The naming scheme of a file-store is largely for the benefit of the users of the system. The naming is typically mnemonic, allowing users to give names which indicate the content and function of individual files. Ideally filenames should be arbitrary strings of characters, though practical limitations usually exist, and some unhelpful restrictions are often imposed such as a very short name length or a particular structure to a name.

Apart from allowing individual users to retrieve files, the naming scheme should allow users to share files. This means that any file can be accessed (via a global context) by any user on the system. Unix achieves this by allowing any user to use the path-name of any file in the system. Note, however, that being able to name a file does not necessarily imply the ability to access it. OS4000 (see later) provides a cumbersome mechanism for file sharing via a shared context called P00L; to access another user's file the other user must catalogue that file in P00L, or catalogue a directory referring to that file in P00L. The OS4000 policy seems to be designed to provide file protection through limitations on naming as well as by more

normal methods of password and user-oriented protection; this turns out in practice to be a severe limitation.

A useful feature of a file-store is the ability to switch contexts. For example, if user 1 wishes to access the files of user 2, it is useful if user 1 can use the same names (ie the same context) as user 2. Unix provides this as a change directory command. OS4000 has no such mechanism, and sharing of files is consequently tedious.

Two possible ways of implementing a file-store are discussed by [SALT]. The first technique effectively requires access to a file for read and write using the file name each time. The second technique, called "direct access", is to use the name of a file to obtain an address, and that address is used for accessing the contents of the file. The choice of technique depends on a number of issues in the design of the system.

The direct access method is most common, and is the one chosen by KUDOS. The arguments in favour of this were mainly performance, and it provides a hidden benefit that scratch files need not be given a user-oriented name at all. In KUDOS there is added complexity in that a file may exist as a number of copies. Accessing a file via its name each time might have certain benefits, and we shall explore these later when we assess KUDOS.

4.3 A SURVEY OF A NUMBER OF EXISTING FILE-STORES

4.3.1 Unix

Unix [RITCH] has deservedly gained popularity as a time-sharing operating system. It has been cited as a standard operating system for the new generation of 16-bit micro systems. Much of its popularity derives from its file-store.

Unix provides a hierarchical file-store visible to all users. A directory can contain files or other directories. All directories, apart from a special directory called root, live in a parent directory. Root is the ancestor of all directories, and a path can be traced to any file or directory from root.

Files are named by specifying the full path name of the file. For example, `"/fs/ken/diary"` refers to the file "diary" in the directory "ken" which is a subdirectory of "fs", which in turn is a subdirectory of "/" (i.e. root). In this way any file in the system can be named.

To ease the tedium of specifying full path-name the concept of a "current directory" is used. For example, if a current directory is `"/fs/ken"` then to refer to `"/fs/ken/diary"` it is only necessary to specify "diary" and the current directory will be prefixed by default.

All files in the Unix system are held in this file-store. By convention certain directories are set aside for system files. Individual users are typically given a subtree for their own data storage. Protection of files is implemented by a system of "ownership" based on the registered users of the system.

A mechanism for mounting and dismounting subtrees associated with volumes provides removable file systems. A removable file system can be mounted anywhere in the system. Effectively, mounting replaces a leaf of the tree with a whole subtree.

A file entry in a directory is actually a pointer to an object called an i-node. The i-node holds protection/ownership information and a map of data blocks on disc. More than one directory entry can point to an i-node, so that an actual file can have more than one name; this is called "linking". This facility is very useful for sharing data, but can cause problems with dangling or lost pointers. For example if a file is accidentally lost through some system failure, the i-node could be reallocated to a different file. The name which originally pointed to that i-node now points to a different file. The i-node knows of only one link to it, but there may be many incorrect ones.

Unix is very popular with users because of the simple structure of the file-store. It is easy to understand, and not obstructive. Files are considered as sequences of bytes

by the system and any record structure must be interpreted at a higher level; this overcomes the structure clashes common with a number of other systems which insist that files have a record structure. Files are random access; system calls exist not only for sequential read/write but also to move the pointer to anywhere in the file for the next read/write.

The file-store also accommodates peripherals, denoted as special files. Thus an application or system program can write to a peripheral as easily as to a file. This is particularly useful in situations such as diverting output to a terminal rather than a file, or vice versa - one program can do both tasks, and the program does not know exactly where its output is going or its input is coming from. Certain natural restrictions do exist, such as lack of random access on a magnetic tape, or reading from a printer.

Directories themselves are classed as files and can be read as any other file. This means that system software does not need special calls to interpret the contents of a directory. It is sometimes useful for applications software to examine directories too.

Apart from the file-store, Unix has a number of attractive features, such as pipes, fork, shell, and a wide variety of utilities and software packages which form a very versatile system. Emulating its features on a local area

network would be an admirable aim. However, some of the features such as fork do exploit the architecture of a single processor system, and therefore would have no direct translation.

4.3.2 GEC OS4000

This operating system [GEC] also has a hierarchical file-store. However the whole file-store is not accessible to all users, only to a privileged user called the super-user. A normal user has direct access only to his own subtree. Indirect access to other subtrees can be provided by a mechanism of context pointers and referencing. A context pointer is a name for a file or directory which is dereferenced to provide the full name in the file-store. Each user has a private set of context pointers. A file or directory name can be a reference, which means it is dereferenced to another file/directory name. Initially a user is provided with a context pointer to the command directory and a shared data area (called POOL), which contains references to files which users have placed there to enable shared access.

There is no "change directory" capability, so that all objects in a users file-store need to be referred to by a full name starting at the users directory. Files are held in a record format, causing certain structure clashes for application programs.

Although potentially as attractive as Unix, the awkward naming and added complexity of context pointers and references to access public and shared data make this file-store tedious to use and severely restricts the activities of a user.

4.3.3 OS 360

The IBM OS360 file-store is remarkable principally because of how difficult it is to use. There are no contexts for naming -- a file must have its full name specified. To manipulate files a huge number of parameters are required such as volume, size (in cylinders or tracks), access method (shared, exclusive, read only, sequential, random), record format (fixed, variable), blocking factors, and a host of other parameters if anything out of the ordinary is required.

However, OS 360 is not an interactive system, and the file-store is usually static. Under a batch system, users are often remote from the system or have procedures set up for them by technical staff. The file-store is designed with efficiency in mind - data is stored in large blocks placed as contiguously as possible. Since under a batch system most files are accessed sequentially from beginning to end, and throughput is important, this significantly improves efficiency.

This is not a file-store to be emulated, especially under an interactive system. However, it is an example of one which did meet the needs of a certain class of users, and which was designed with the overriding objective of efficiency. It is included here because the author suffered long at its mercies, and wishes to warn others of the tedium of such a file-store.

4.3.4 Xerox DFS

The Xerox Distributed file-store [G1FF1] is particularly relevant to this thesis. It is based on an Ethernet local area network, and has a number of users. With some adaptation many of the algorithms in this thesis could be implemented on this system.

It is based on the client-server principle. The basic communication is for a client to send a request to a server, and for the server to send a result. A client can either issue a request and wait for the result, or issue a number of requests and accept results as they return (not necessarily in the same order as the requests).

The DFS is implemented as a number of servers. Files are identified by a unique identifier (UID) which is effectively a long integer. Clients communicate directly with the servers which contain the files they wish to access.

Atomic transactions and recovery are implemented using a mechanism called "intention lists". Locks and updated records are stored in the intention list, and the state of the system outside the intention list is unchanged until the intention list is committed. On commit the contents of the intention list are applied. If a failure occurs before commit the intention list can be scrapped and the transaction has no effect. After the intention list has committed, and before the contents applied, a failure can be recovered by applying that intention list again. This is an alternative to updating the disc and rolling back if a transaction fails. It is essentially a form of careful replacement only of individual disc pages, rather than whole files.

A scheme of "stable storage", where a disc page is written on two parts of a disc to improve recovery under a crash, is used to increase reliability of crucial data such as intention lists. By writing the same data to two different surfaces, the likelihood of data being destroyed through faults on a disc are minimised. Also, by writing two pages, the likelihood that one succeeds is higher.

Deadlock prevention is through a locking mechanism which can be broken. If a client wishes to lock a record which is already locked, then the intention list of the first client is marked so that it cannot commit, and the second client now has the lock. This avoids certain forms of deadly embrace, but consistency must be ensured by the

client, and a client can conflict with another client so that neither can proceed.

No directory scheme is implicit in the DFS. A directory server has been implemented as a client of the DFS to give a mnemonic to UID mapping. This is an optional service and a client may use its own mechanisms for locating files.

There is no concept of ownership for a file - knowing a UID permits access to it. How a UID is determined is not described. File operations are apparently only at disc page level - a reasonable requirement, but not as neat as the Unix sequence of bytes. How files are stored on disc is not described.

The DFS is a basic but very flexible file-store. More application-oriented structures can be implemented by clients to the DFS (such as the directory server). The atomic transaction property permits writing of a database system as a client of the DFS. Atomic transactions in a file-server are also important in file-store applications, as we shall later discuss.

The directory scheme detailed in this thesis could well run as a client to this DFS. The client-server concept has in fact been used. This file-server is far advanced of the crude file-server used in KUDOS, and its properties of atomicity would have been a boon to the file-store in a number of ways. The KUDOS file-store is, however, more

user-oriented and leans more towards Unix in its external appearance.

4.3.5 Apollo Domain

The file-store on the Apollo Domain system is hierarchical, and similar to Unix. Each node has a hierarchical file-store, and files are named within a node in the same manner as Unix. There is a network-wide root directory which contains the root directories for each of the nodes. The network-wide root is replicated on all nodes. The full network-wide name of a file is preceded by two slashes and the name of the node where it is held. A system of current directories is implemented as in Unix to ease naming.

For example, if the current directory is `//compsci1/smith` (ie the directory smith on the node compsci1), then a user on the node compsci1 can use the names `//compsci1/smith/prog/sort`, `/smith/prog/sort`, and `prog/sort` to refer to the same file. The first is the network-wide name, the second the local name, and the third the name in the current context.

The notion of linking varies from that of Unix. Apollo provides a link by storing the full name of a file for second and subsequent links; Unix just provides a pointer to the i-node for the linked file. This should avoid the problem of dangling pointers which can occur under Unix if

an i-node is lost.

4.3.6 George III

One of the earliest hierarchical file-stores is supported by the George III operating system on the early range of ICL mainframes [ICL]. It names all files in the system, both disc and magnetic tape files. A user need not know where a file is stored and a file may be moved by the system without the user knowing.

There are three types of file in George III. Peripheral files are strings of records with formats appropriate to the peripheral being simulated, say a card reader or a line printer; such a file may be kept on disc or magnetic tape. Direct access files have a format appropriate for storage on disc or drum only, reflecting the random access capabilities of such devices. Magnetic tape files also have a special format, corresponding to the storage medium.

This is contrary to the Unix philosophy that a file should be device independent. Otherwise access to files through the job control language can become long-winded and prone to semantic errors. On an interactive system in particular it is tedious to have to over specify the details of files in order to gain access. It also causes problems for the programmer who needs to consider the types of files being used.

The George III hierarchy is a rooted tree with a "master directory" at the root from which all files and directories in the system can be traced. A directory can contain files or directories up to a limit of 64. Each directory has associated with it a unique user name, and that user name can be used to name the directory instead of the full path-name. A dictionary of users is kept, and this forms the basis of the accounting system for storage. A current directory scheme is used to ease naming.

The binding of a unique user name to a directory, and the limit on a directory size seems an unfortunate restriction. However, the George III system is batch oriented, and as such is unlikely to have a large number of small files per user.

Directories contain access information for files based on user names. Access to a file in a directory can be restricted or prevented for other named users of the system.

A back-up system runs an incremental dumper at regular intervals which copies recently updated files from direct access devices to magnetic tape. This means that a failure of the direct access system can be recovered by a roll-back to a previously accepted state. The scheme is also used to free direct access devices when they are becoming full. Files which are deleted from direct access devices can be retrieved from magnetic store on demand. This allows the file-store to contain more data than can be held on direct

access devices.

This appears to be a significant attempt at providing a large scale file-store on a commercial operating system with built-in redundancy to allow recovery from system failures. Compared with OS360, this is a much more attractive file-store.

4.3.7 Other Unix-like Distributed File-stores

A number of distributed Unix-like systems, or extensions of Unix, appeared at the end of the author's involvement in KUDOS. They are discussed here for comparison. Notably, the approaches have been different.

The LOCUS project [POPEK] closely parallels the KUDOS project, especially in the provision of low level network transparency. It too provides a hierarchical file-store with replication. The operating system and file-store are based on existing Unix software.

The file-store is broken into groups, and the files within a group are replicated at all sites associated with that group. Synchronisation of file usage by many users is through a "Current Synchronisation Site" which is designated as being responsible for coordinating access to all files within that group. All open calls for a file must involve a message to the Current Synchronisation Site irrespective of whether or not the file is stored at that site. Any site which has a copy of the file can support the open request.

Once a file is updated at one site, the Current Synchronisation Site is informed and the other sites which contain copies of this file are brought up to date. Each copy of a file has a version vector associated with it. This version vector contains an integer for each copy of the file. When a file is updated the entry in its version vector for itself is incremented. [POPEK] claims that by comparing version vectors inconsistencies can be resolved when, for example, a site has been offline during an update.

Certain conflicts can arise when say a file is replicated three times, then one copy is offline and the other two updated followed by the latter two being offline and the former being updated. For the sake of availability, such possibilities have been allowed. [POPEK] claims that such conflicts are likely to be rare, since actual sharing of files is low; this seems to be ignoring the problem.

Considering the similarity between LOCUS and KUDOS, it would have been worthwhile to make a full comparison. However, the timing of the two projects makes this infeasible for the purposes of this thesis.

Cocanet [LAWR] and the Newcastle Connection [BROWN] both provide a network-wide hierarchical file-store. These two hierarchies are similar to each other and to the Apollo Domain, and are direct derivatives of the Unix hierarchical file-store.

In the Newcastle Connection it is possible to effectively mount the tree of one node as a subtree in the tree of another node. Thus the root of one system can appear as a subdirectory in another system, building a large naming network out of a number of naming networks. This is a radically different approach from KUDOS which provides a single naming network by overlaying naming networks in a way to be described.

Neither Cocanet nor the Newcastle Connection, as yet, supports multiple-copy redundancy of files, though no doubt such extensions are being considered. This, and the approach to naming are the significant differences between the external appearance of the KUDOS file-store and these two file-stores.

CHAPTER 5

REFINEMENT OF THE PROBLEM

In chapter 1 we stated the basis for the research presented in this thesis is the problem of how to construct a file-store which is reliable in terms of high accessibility of data and low likelihood of data loss. Let us now elaborate this in the light of chapters 2,3,4.

Firstly, we gave a broad definition of reliability. We said that reliability is a measure of how successfully a system satisfied its specification. We highlighted the problem of rigorous specification and suggested that an incomplete specification would be adequate for most systems, so long as the specification covered the essential aspects of behaviour; indeed the behaviour of a system is often implicit in the type of system, say a car is assumed to carry people.

In the third chapter we discussed local area networks as an implementation vehicle for a file-store, highlighting certain features of local area networks as being useful to the designer, especially the capacity for dynamic redundancy and component independence.

We then discussed fully what is understood by file-stores and a number of related issues. We can draw a number of conclusions regarding the desirable properties of a file-store, namely:-

1. The file-store must provide a sensible naming scheme which permits a user to name any file unambiguously, and in a manner easily understood by the user. Moreover, the naming scheme should be flexible, allowing the users a wide choice of names and the ability to change context. Arguably the hierarchical scheme used by systems such as Unix are the best practical systems currently providing such features.
2. In a multi-volume system, some means of controlling the location of files is required. This is particularly important where multiple copies of a file are kept.
3. The file-store should control access by a number of users, preventing inconsistencies due to concurrent access, and also to provide some form of data protection to prevent the access of a user to data which should not be accessed by that user.
4. The file-store must provide adequate performance in terms of the latency of file access requests and in terms of data transfer rates.

5. The file-store must be reliable in terms of availability of files, the likelihood of loss of files, and atomic update of files.

In the design of the KUDOS file-store the overriding objective was reliability, measured in three ways:-

1. The probability that an individual file is available at any one time.
2. The probability that an individual file is lost.
3. The probability that file updates succeed completely or not at all.

KUDOS provides algorithms which allow a file to be replicated in such a way as to increase availability and reduce likelihood of loss.

KUDOS, however, provides a number of other important features which largely provide the aforesaid desirable properties, namely:-

1. A single global hierarchical naming scheme.
2. Controlled multiple copy redundancy.
3. A neat solution to data placement.
4. Deadlock avoidance.

5. Automatic reconfiguration.
6. Limited recovery facilities for user transactions.
7. File protection.

KUDOS made a small attempt at providing good performance, but the prototype system had too many inefficiencies to highlight this and time was too short to tackle this problem fully.

The rest of this thesis will be devoted to a description of KUDOS and its file-store, and some assessment of its success. A number of points will be made regarding the implementation of the prototype. The prototype itself has probably little scope for extension to a working system, so it will be important to highlight throughout what was learnt by implementing a prototype. It might also be useful to consider how the KUDOS algorithms might be used elsewhere.

CHAPTER 6

KUDOS

6.1 THE KEELE UNIVERSITY DISTRIBUTED OPERATING SYSTEM

KUDOS stands for the Keele University Distributed Operating System, developed by the author. The research presented in this thesis forms part of a research project in the Department of Computer Science at Keele University, nominally investigating distributed file-stores. That project commenced December 1979 and is still active.

The Keele project was funded by the Science Research Council, with two research associates, the grant holder Dr K.H. Bennett, and three research students. The work presented here inevitably benefits from the individuals within the project. However, the work presented is entirely the author's; where an individual has contributed in any way is indicated in the text. This thesis represents a project within a project, and only reflects one aspect of the overall project.

The project began with the somewhat over-ambitious aim of designing and building a distributed operating system. Much will be said about this later, assisted very much by hindsight. This chapter describes the most successful attempt, developed by the author. In all humbleness, it must be said that success is measured largely by the fact that a prototype was implemented, and worked; other approaches, perhaps more adventurous and with greater potential, were being explored by other members of the team.

A small implementation of KUDOS exists on two LSI-11/02 micro-computers, providing a distributed file-store, resource location facilities and a communications scheme. Currently the only processing available to a user is a UCSD Pascal micro-engine [MICROE] which accesses the distributed file-store remotely.

We shall begin this chapter by describing the early goals set by the team, originally detailed in [LUNN2]. Some reflection on these goals will be followed by a description of KUDOS. The existing implementation will be described, finishing with a set of conclusions and experiences with KUDOS.

6.2 AIMS OF KUDOS

The original design objectives for the development of a distributed operating system [LUNN2] are included here (verbatim). These shall be qualified in the light of

experience. The decision to write a distributed operating system was prompted by the lack of available operating systems for local area network based systems at the time. Moreover, it was seen as a vehicle for developing ideas other than those on file-stores, providing a basis for discussion on topics such as process location and communication techniques.

6.2.1 General Objectives

These guidelines would apply to any development of a computer system. They ought to be obvious, but are stated here for completeness.

1) To provide a useful system for the given application, with an adequate response to user requests.

2) To provide a reliable system which ensures

1. accessibility of file-store, measured as the probability of a file being available on the system

2. integrity of file-store measured as the probability that a given file is the latest version of that file

3. reliability of hardware, measured as the probability that a usable system is provided together with the mean time to failure of the system.
4. reliability of software, capable of tolerating certain classes of hardware error (such as parity errors, disc i/o errors) or even component failures (such as disc head crashes or, in the case of a network, loss of one of the nodes).

3) To provide a secure system which enables a user to protect data from corruption or even access by other users, and which can curtail a rogue process attempting illegally to alter currently executing software.

Briefly, these three objectives can be summarised as saying that the system should be useful, reliable and secure.

6.2.2 Objectives Within Our Application

These objectives, though more specific than the previous ones, avoid dictating the technical specification. Rather, they attempt to define ideals which would apply to many similar systems. Such ideals may not be achievable, but we hope to go a long way towards them or discover reasons why we cannot.

1. To provide a Pascal compile/execute system implemented on a distributed network, providing an integrated hierarchical file-store spread across the storage devices of the various machines on the network.

2. To implement in such a way that the reliability of the system increases with the number of machines on the network, rather than decreases.

3. To implement in such a way that the integrity and accessibility of the file-store is significantly better than it would be if implemented on a single machine with reliability equivalent to that of a single node on the network. This seems to imply that the file-store is not dependent in any way on any one node of the network.

4. To make the fact that the system is implemented on a distributed network opaque to the user; that is the system appears as a black box.

5. To ensure that the processing capability of the network increases proportionately with the number of nodes on the network.

6. To implement in such a way that response time is not significantly degraded by increasing the size of the network, perhaps to some upper limit (say ten machines). If possible response time should be improved by increasing the size of the network, by utilising the potential for parallel processing.

7. To provide automatic reconfiguration of the system should any of the components fail, other than the communications medium. This should be hidden from the user as far as possible, though some degradation of response might be expected, firstly through the reduction of processing power in the system, and secondly in the short term through the use of recovery routines.

8. Above all, the system should never grow too complex, large or limited to prevent easy experimentation. This is a research project, and as such is primarily interested in trying out new ideas.

6.2.3 Experience

In an open-ended project one rarely achieves exactly what one set out to achieve. The benefits derive from discovery and self-discovery as much as from the ultimately realised aims. The original objectives provide an interesting yardstick by which to measure success, and they are also useful as guidelines when the tendency to meander is too great.

The general objectives now look a little naive, and could doubtless be refined. The entity "useful system" and the property "adequate response" need much elaboration. The notion of reliability has been extended to include accessibility and atomicity of update. There would be a chariness now of using the word integrity; more specific

statements of properties are needed, integrity being a term too widely and too carelessly used.

The specific objectives were more soundly stated, quite feasible, and are perhaps worth commenting on individually:-

1. The notion of providing a Pascal compile and execute system was perhaps an unnecessary restriction. The project was nominally to investigate file-stores, and a file-store should be able to interface to any sensible processing system. The origins of this objective were embedded in the project no doubt because of the Pascal implementation expertise within the department. The file-store implemented in KUDOS could be interfaced to any particular language/processing system. The fact that the prototype interfaced only to a Pascal system was due to limitations on time, and it could as easily have been interfaced to a BASIC programming system, or whatever.

2. Increasing the reliability of the system with the number of nodes on the system implies exploiting the possibility for dynamic redundancy and implies high autonomy in the system. KUDOS achieved this to a large extent.

3. By integrity it was meant that files rarely be lost. KUDOS provides a flexible response to reliability in this respect, as well as providing a flexible scheme for increasing the accessibility of files. This objective became an overriding one in the realisation of KUDOS.

4. Our view of distributed computing has changed. We no longer see the system as a "black box". This is largely due to our growing realisation that most individual users can be satisfied by the provision of a powerful personal computer at low cost. Under such circumstances the user must be aware of when, why and how he is using a distributed resource. Thus the distributed nature of the system should be translucent rather than opaque.

5. Greater understanding of the limitations of extending local area networks limits the objective of increasing processing capability proportionally with the size of the network. A number of bottle-necks exist in local area networks, such as the bandwidth of the communication system or heavy demand on certain critical services.

6. KUDOS was not designed with the objective of performance foremost. Little can be said about response times in the light of experience, except that the personal computer approach ought to satisfy the needs of most users, and that response time should be little affected by the network except in so far as a user requires a network service.

7. Automatic reconfiguration was largely achieved in KUDOS through the directory system and resource directories.

8. KUDOS did not grow too large or complex. As an operating system it probably has little to offer. Hopefully, however, some of the ideas will find homage elsewhere.

Overall, KUDOS achieved a number of the teams original objectives. To call it an operating system is perhaps too gross; it is more a limited collection of network services. Much has been learnt, however, and there is potential for extending the ideas. One begins to realise, to quote C.P. Snow, one's "inability to do much". To design, implement and evaluate from scratch a system as ambitious as originally envisaged is probably beyond the scope of a small, newly-formed team in the time available.

One might seriously argue that the objectives were wrong. In retrospect, it would have been better to set objectives such as the investigation of existing distributed file-stores and a relative evaluation of their properties. Some classification of file-stores would have been useful, as would some classification of local area network systems. Much of this went on implicitly, but it would have been better to have them as fixed objectives.

Such are the benefits of hindsight. Nevertheless, the original objectives proved useful, if only as a torch to light the way. The consequences of those objectives are now described.

We shall begin by describing the message passing mechanisms developed in KUDOS. The scheme was developed in tandem with the operating system, and was a huge compromise between need, efficiency and compactness of code (a potential recipe for disaster, but it worked - just). We shall discuss the compromises and suggest a better approach which should have been taken, had it been feasible. The experiences with message passing and communication were by no means novel, but they did consume much time and effort.

We shall then describe two underlying philosophies in the development of KUDOS, namely "client-server architecture" and "public/private domain architecture". An attempt is made to clarify and distinguish certain general features of a local area network computing system.

There will then follow a description of the prototype system implemented, and the chapter will end with a number of experiences and conclusions.

6.3 KUDOS STRUCTURE

KUDOS is a message-passing distributed operating system for a heterogeneous system of computers connected by a local area network. It is a client-server system, where a server can in fact be a client of another server. A distinction is made between the "public domain" (shared resources) and the private domain (personal resources). Objects are dynamically relocateable within the system and are located

by "resource directories".

Message-passing was chosen since it offered a simpler solution to providing a heterogeneous system. Procedure calls across a network with nodes having different machine-codes was considered (almost certainly mistakenly) to be difficult to implement. Experience in developing KUDOS indicates that some form of procedure-like call across a network with appropriate synchronisation would be easier to use and less error-prone than ad hoc send and receive primitives.

6.3.1 The KUDOS Message Passing Scheme

Suppose $n[1], \dots, n[q]$ are nodes on the network. Each node $n[i]$ provides a number of ports for data reception. Any process in any node can send a message to any port in any node by specifying the address of that port (eg. $(n[i], p)$ - port p on node $n[i]$). A process can send to a port within its own node.

Any process in a node can receive data from any port in that node; thus more than one process can receive from the same port. The motivation for this was to allow a number of similar servers to accept requests from the same input stream. In practice this did not occur, and had this been necessary an alternative would have been to invent a coordinating process which received all requests for a collection of similar servers and rerouted the request to a

ready server. Multiple servers collecting requests from the same port is potentially quite error-prone.

A process can create a port within a node and later destroy it. Problems were experienced when a port was destroyed and its number reallocated; the newly created port occasionally received messages intended for the process which earlier destroyed a port with the same number. Such situations arose when a process backed out or the protocol between a client and server broke down, often due to errors in the code, having unfortunate consequences and making the fault difficult to trace. To overcome this problem port numbers were allocated so that repetition was infrequent. Messages arriving for a destroyed or non-existent port number disappear; ideally such messages should raise an exception in the sending process, but the synchronisation chosen forbade this.

To send, a process must provide a reference to the data to be sent. To receive a process must provide a buffer for the received data. Two forms of receive are defined - fixed and variable. A fixed receive will only be satisfied when its buffer is filled by one or more sends. A variable receive will be satisfied by the first send, and will return the size of the data in its buffer. The reason for a variable receive was to allow communication with an interactive keyboard, where the length of an expected message cannot be predetermined. It might also have been useful for detecting the end of a file, with a short or

empty buffer being returned, though this was not used in practice. A send which is not exhausted by a receive will have its remaining data held for subsequent receives.

This scheme is very flexible for data transfer. For example, a file-server may send a file 512 bytes at a time, but a client may only want to take 80 bytes at a time. Using a fixed receive on an 80 byte buffer the underlying structure clash is hidden from both client and server.

A system of timeouts is provided for both send and receive. A send will timeout if:

- i) the node does not exist;
- ii) the node does not respond;
- iii) the message is to be sent locally and the port does not exist;
- iv) the message is to be sent locally and the message has been first in the queue for a specified time.

A receive will timeout if the port does not exist or if nothing arrives in the port within a specified time.

Unfortunately timeout (ii) can occur because of Protocol errors. Timeout (iv) was chosen as being more fair than timing out over length of time in the queue - it

ensures that no send waits indefinitely but does not penalise a sender if the queue is unduly long.

Within a node, both send and receive hold up the issuing process until completion or timeout. A send to another node will suspend only until the data has been transported to the node, and not necessarily received from the port; this removes the need for an acknowledge, but leaves the process in an uncertain state regarding the success or failure of the send.

All client-server communication in the system is by message-passing. The communication system on a node has been called the "message exchange". Message exchanges communicate directly via a high bandwidth local area network.

The above primitives are a mixture of design and expediency. It would have been preferable to have used much higher level primitives, with the much greater reliability of, say, a full transport service with its own error checking and correction. However, given the software and hardware available, it was necessary to restrict the size of the message exchange, and to improve its speed by reducing its level of service.

Close parallels can be drawn between this scheme and that of [LISK]. A node corresponds to a guardian, although we provide no nesting of guardians. The scheme is essentially "no-wait send", issuing a message without

waiting for acknowledge or reply. We would prefer a "synchronisation send", waiting for acknowledgement of receipt, and an "invocation send", waiting for a reply as well as an acknowledgement. [HALS] suggests how we might implement these, but it is felt that any improvement must be preceded by a hardware upgrade to provide a faster local area network.

Note, in particular, that message passing is through an address rather than a name. A naming scheme is provided, but must be used independently to obtain an address.

The largest problem on the message exchange is that a remote send does not guarantee receipt. In practice most transactions are of the send request / wait reply type. A remote invocation send (often called "remote procedure call") would be more useful, and reduce the error checking required by a client - presently it is necessary to check for timeout on both send and receive.

Communications we see as the hub of any computer system. On a single processor system the issue is often clouded by other machine specific ones. On a distributed system it becomes clearer. Once a concise and powerful mechanism for communication exists we see local area networks developing from ad hoc interconnections of computers to complete computation systems in their own right.

6.3.2 Client-Server Architecture

In [STURG] and related work we see an emphasis on the idea of a client-server relationship. A client is defined as any process initiated by or on behalf of a user. A server is any process which provides a service to a client. In this way a server can itself be a client of another process. Examples of client-server relationships are (in Unix) shell-editor, editor-filestore, shell-compiler, compiler-filestore.

On a distributed system, the client-server relationship is very important. The client-server interface is often implemented across nodes. Consequently it must be concise and not take advantage of side-effects as is often the case in single-processor systems. This necessity is even more important when a single server can be accessed concurrently from a number of nodes.

6.3.3 Public/Private Domain Architecture

This section expands the philosophy developed within and the projected structure of KUDOS. It is similar to many local area network computer systems, such as the Cambridge Model Distributed System. KUDOS is in an embryonic state at the time of writing this thesis, and this section should be viewed as a projected strategy rather than a description of a completed system.

The most important feature of the strategy is the dichotomy between the public and private services provided on a network, and the isolation of the interface between them. Though such concepts are embedded in many local area network systems, they are not always emphasised. The distinctions ought to be particularly useful for large scale development, especially of a heterogeneous system. One can see a developing trend toward such a philosophy, especially with regard to protocol standardisation and open system inter-connect [ISO]. However, we shall decline to argue the case and hope that the reader appreciates the value of the approach.

The first impact of the micro-processor was to disperse computing. Yet people do not normally work in isolation. KUDOS aims to integrate the dispersed processors to provide a more attractive service to users whose needs overlap or who require close cooperation with one another.

At the same time KUDOS aims to maintain and enhance the independence of a user's work-station so that it provides the level of interactive response now expected of a micro-processor system. If necessary a work-station should be designed to stand alone. Not only does this have advantages with regard to performance and reliability, but security can be improved by restricting remote access to a work station.

KUDOS attempts to take advantage of low cost processing power to provide a responsive service to a user whilst retaining the advantages of a mainframe system with regard to mass storage, inter-user communication, high-quality peripherals, and services such as periodic archive and data preparation.

The user sees the system in two parts. Firstly there is the private domain over which he has complete jurisdiction. Secondly there is the public domain which can be accessed only by certain protocols.

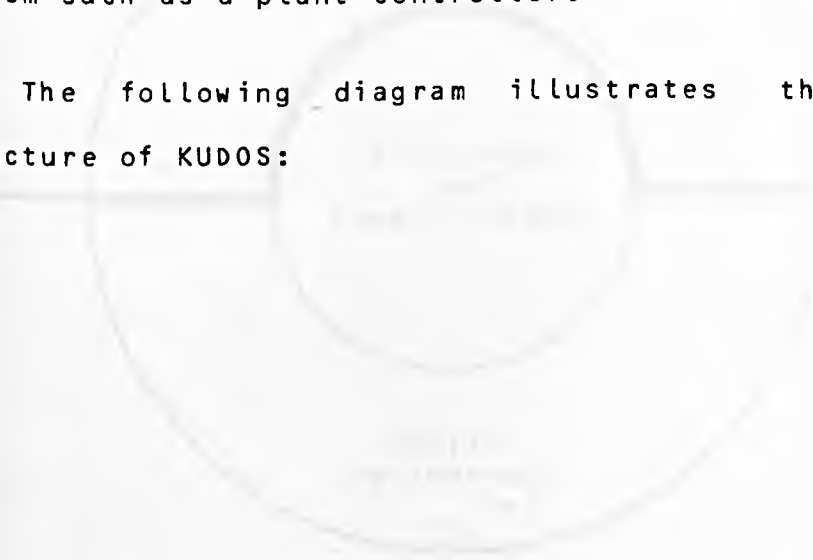
6.3.4 The Conceptual Structure

A personal autonomous work-station (PAWS) is envisaged which provides a user with all his processing needs and which controls all private data storage and private peripheral handling. Each personal autonomous work-station interacts with the multi-access shared service (MASS) through the public system interface (PSI). The public system interface does not provide access to another personal autonomous work-station although this might be achieved indirectly through a service provided by the multi-access shared system.

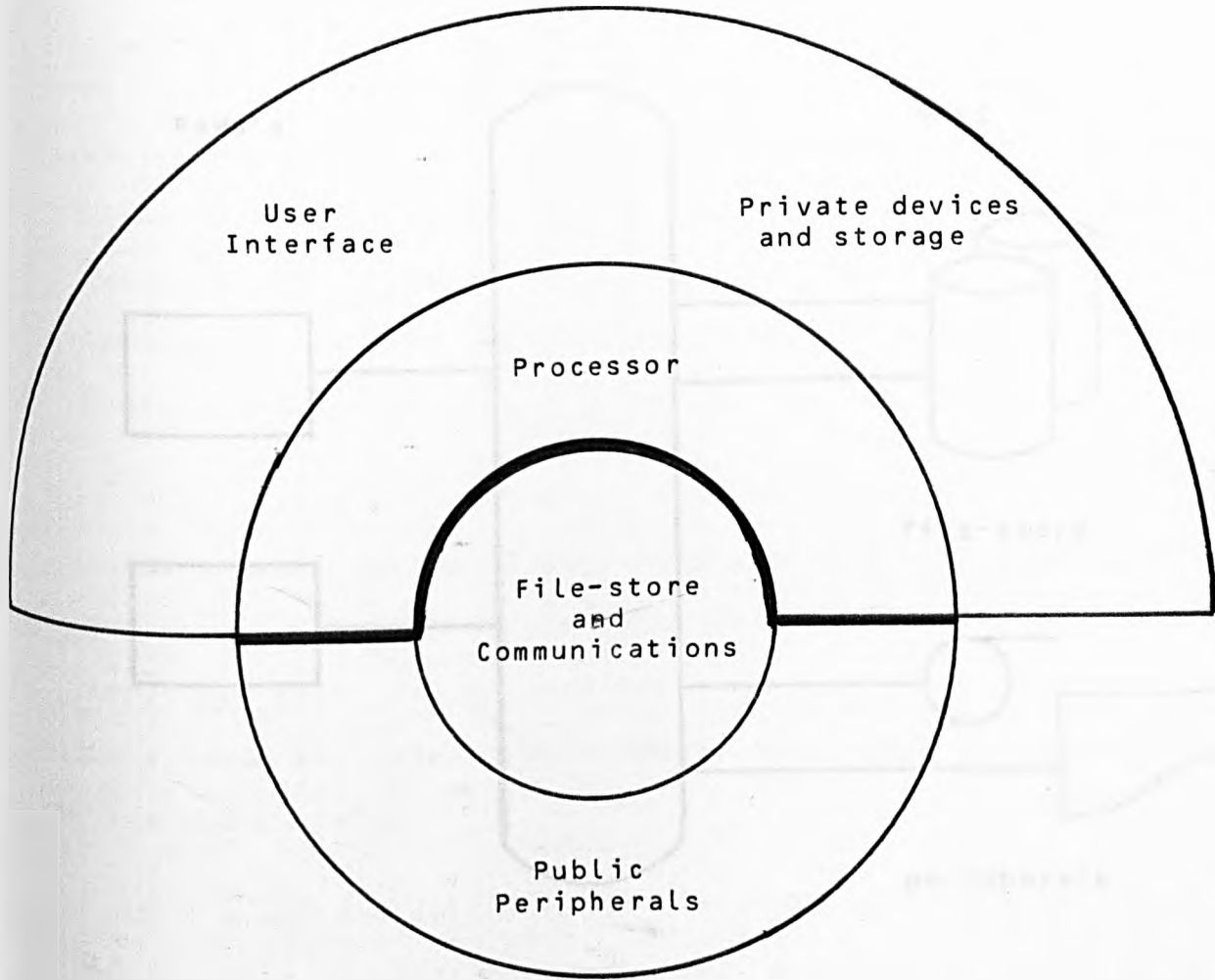
The multi-access shared system provides general services such as a file-store, inter-user communication and public peripheral access. The public system interface provides the primitives for accessing the multi-access





shared system. The multi-access shared system is in no way dependent on any personal autonomous work-station, but a personal autonomous work-station can be made dependent to varying degrees on the multi-access shared system; at one extreme a personal autonomous work-station could be just an intelligent terminal (in which case its autonomy is minimal), at the other extreme a sophisticated system providing graphics and mass storage, or be a specialised system such as a plant controller.

The following diagram illustrates the conceptual structure of KUDOS:

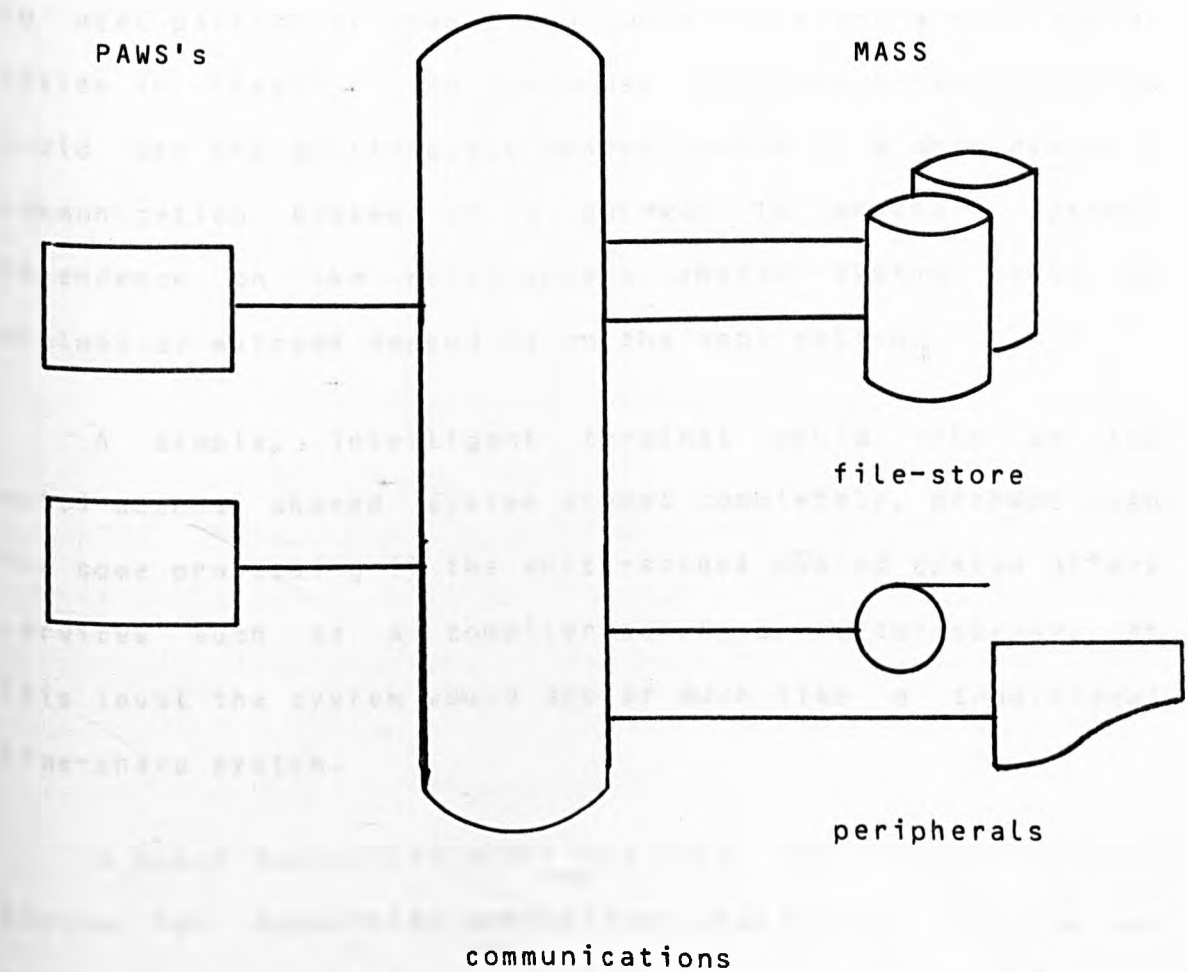


KUDOS CONCEPTUAL STRUCTURE



-  Public System Interface (PSI)
-  Public Peripheral interface (PPI)
-  Personal Autonomous Workstation (PAWS)
-  Multi-access Shared System (MASS)

A likely physical structure is:



6.3.4.1 The Personal Autonomous Work Station -

Each personal autonomous work-station could be tailored to meet particular needs, and could represent a substantial system in itself. The personal autonomous work-station could use the multi-access shared system as a data store, a communication system or a gateway to another system. Dependence on the multi-access shared system could be minimal or extreme depending on the application.

A simple, intelligent terminal would rely on the multi-access shared system almost completely, perhaps even for some processing if the multi-access shared system offers services such as a compiler-server or editor-server. At this level the system would appear much like a traditional time-share system.

A plant controller might use the multi-access shared system for depositing operational statistics. This is not essential to the functioning of the plant, but useful to other users of the multi-access shared system. Here we have high independence of the personal autonomous work-station.

The organisation and function of the personal autonomous work-station is undefined in this system except in so far as it must conform to the requirements of the public system interface. Thus it must be able to tap the communication system and do so in a particular way. An individual personal autonomous work-station can be tailored to a particular application.

6.3.4.2 The Multi-Access Shared System -

The multi-access shared system should provide a high quality, highly reliable file-store with a hierarchical naming scheme. The file-store must encompass protection of data, controlled multiple access, mutual exclusion, backward error recovery to aid failure recovery in the personal autonomous work-station, and forward error recovery to cope with internal failures in the multi-access shared system. There should be controlled access to peripherals, either directly or through a spool in the file-store. Other services which might be offered by the multi-access shared system could be a mail-box system, a buffer-server for inter-PAWS communication, or gateways to other systems.

6.3.4.3 The Public System Interface -

The public system interface provides the primitives for accessing the multi-access shared system. Examples of primitives would be file create, file delete, directory create, directory delete, file write, file read, file lock, file unlock. The public system interface should be concise and as simple to use as possible.

6.3.5 Object Naming, Addressing And Location

Communication with all objects in the system is through message-passing via ports as described earlier. A port is addressed by two integers $(n[i], p)$ where $n[i]$ is the number

of the node, and p is the port number within that node. A server has with it an associated name. The location of a server may change, but its name does not. We describe here the algorithm used by KUDOS to locate servers, and other global resources. This first appeared as [LUNN3]. The algorithm is low on processing requirements and on message-passing demand.

6.3.5.1 Resources -

Each resource in the system is named. Resources are accessed by an address. A resource might well change its address, but not its name. Names of resources need not be unique, although the nature of the resource might require this. Addresses of resources are necessarily unique; that is, two resources cannot live at the same address at the same time. Finally, a resource may not be available, in which case it has no address.

Examples of such resources are dismountable discs which may be moved from one drive to another, or from one machine to another. A disc is likely to have a unique name, such as a volume number. Such a disc must, however, be accessible wherever it is mounted.

Naming of resources might be sophisticated. For example all printers in the system might be prefixed /PRINTER and suffixed by location or device type such as /PRINTER/ROOM2 or /PRINTER/DAISYWHEEL. Location of

resources could then be quite elaborate for example finding the names of all printers available so that a user can choose the place to print a file.

6.3.5.2 Addressing -

We assume that nodes are ordered cyclically, as if on a loop, so that each node has an immediate left hand neighbour and an immediate right hand neighbour. Such an ordering of nodes is quite natural under ring-structured networks, such as a Cambridge Ring, but can easily be simulated under other network structures such as Ethernet.

6.3.5.3 Processing -

It is assumed that each node has a limited processing power. Dumb nodes can however be assimilated into the system since the resources they hold will be managed at another node which does have processing ability. Thus dumb nodes can be considered as not strictly part of the network, at least for the purposes of this algorithm.

It is required that at least one node in the system can dynamically create and execute a non-trivial process. Since this is a requirement for almost any useful computer system this is not a drawback. More than one such node is likely, and this adds to the reliability of the algorithm.

6.3.5.4 Resource Location -

Each node in the system maintains a directory of the resources which are currently available on that node. This is called the local resource directory. Access to a local resource directory (or more strictly its manager) is through a fixed address on all nodes, say address zero. A process locates any resource in the system by sending a message to its local resource directory (ie the one on the process's node).

All local resource directories active in the system maintain between themselves a single total resource directory. The manager of the total resource directory runs at some unspecified node and has an address determined by the process creation mechanism. The total resource directory knows of all available resources in the system and is the ultimate source of reference when seeking a resource.

To locate a resource a process sends a find request to its local resource directory, containing the resource name, and then awaits for the local resource directory to reply with the resource's address or an indication that the resource does not exist. The first task of the local resource directory is to check whether or not the resource is local to the node, and if so it replies with the address. Otherwise the local resource directory passes on the find request to the total resource directory which will reply to the process with the appropriate address. Apart from the

address of the total resource directory a local resource directory maintains no other information about the system external to its own node.

When a resource becomes available the manager of the resource informs its local resource directory of the name and address. The local resource directory stores the information and also passes the information to the total resource directory. When a resource is removed from the system the local resource directory and the total resource directory are informed in a similar manner. Thus the total resource directory reflects swiftly any change in system configuration.

When a node is started up the local resource directory does not know the address of the total resource directory. When finally the local resource directory is asked for a resource it does not know about, it must first locate the total resource directory. Thus the local resource directory sends a message to its immediate left hand neighbour. If the left hand neighbour does not know where the total resource directory is it passes the message on to its left hand neighbour. Assuming the total resource directory exists the message will propagate around the system until the address of the total resource directory is found.

If the total resource directory does not exist then the message will ultimately return to its originating local resource directory. This local resource directory then

knows that the total resource directory does not exist and can therefore set about creating it. To do this the local resource directory sends out a bid to its left hand neighbour. This bid is an indication of how prepared a node is to run the total resource directory. On receiving a bid from its right hand neighbour a local resource directory inspects the bid, if it desires it ups the bid, and passes the bid to its left hand neighbour. Thus the bid cycles around the system until it returns to the originator. The returned bid contains the address of the node which made the highest bid, and the bid originator sends a message to the highest bidder asking it to create a total resource directory. The total resource directory, on creation, cycles a request to all local resource directories for information contained in each local resource directory.

It is likely that more than one bid is originated since two or more local resource directories might simultaneously detect the absence of the total resource directory. Thus when a local resource directory receives a bid it enters "bidding mode". Once in bidding mode a local resource directory cannot change its bid. If bids are ensured unique, say by including the node address as the least significant part of the bid, then the total resource directory will not be created at two sites. When the total resource directory cycles its request for information each local resource directory in turn exits bidding mode and stores the address of the new total resource directory as

well as sending its own information to the total resource directory.

6.3.5.5 Some Effects Of Node Crashes And Start-Ups -

If a node crashes the total resource directory is likely to contain addresses of resources which no longer exist. When a process tries to access such a resource it must inform the total resource directory of the problem. The total resource directory will then duly remove all information concerning the crashed node, perhaps after checking out the node itself.

Possibly a node might crash whilst it holds a message which is being cycled. Thus an originator of a cycled message must implement a time-out to prevent indefinite hold-ups. Another problem arises when an originator crashes before the new total resource directory is created. This is solved by putting a time-out on bidding mode, and any local resource directory in bidding mode which times out originates a bid itself.

Finally, one problem occurs when two bids are cycling the system and a node starts up. If this node misses one bid but makes the highest bid on the second bid then total resource directories would be created on different nodes. One solution is for the left hand neighbour of a node to be fixed whilst the local resource directory is in bidding mode, thus excluding newly-started nodes from subsequent

bids. Alternatively two total resource directories could be allowed to start, but the one with the lower bid could back down; this is facilitated by including the bid value on the request for information circulated by the total resource directory.

Failure of the node containing the total resource directory would ultimately be detected by a local resource directory. This local resource directory would then have to initiate a bidding cycle. Closing down a node containing the total resource directory could involve direct copying of the total resource directory information to another node or by the total resource directory initiating a bidding sequence via its local resource directory and ensuring that a null bid is made.

6.3.5.6 Alternative Schemes -

The above algorithm effectively implements a central directory, although this central directory is easily reestablished in the event of failure. Whilst this runs counter to some philosophies of distributed computing, it is reliable and reasonably efficient. Moreover it makes use of redundancy in a clever way to guard against failure. As such it represents a software implemented solution to the problem of name-servers in the Cambridge Model Distributed System [WILKE].

[CASEY] discusses various alternative schemes, mainly based on a broadcast facility. Without a broadcast facility the feasibility of keeping copies of the total resource directory at each site is much lower. Also, without a broadcast it is much more difficult to search for an object. Moreover, a broadcast in a loosely linked system involves a high overhead, especially since communications are likely to be significantly slowed by protocols.

The above scheme needs storage proportional to the number of resources available in the system, not to the physical size of the system. Nodes handling few resources have few overheads. Replicating a complete directory at each site fails on both these counts. Further, adding a new resource is much less expensive in the above scheme than in updating each site.

In a tightly coupled system where communications are fast and inexpensive, and where access to shared data is relatively easy the above scheme is far too cumbersome and sophisticated. In a loosely coupled system, with communication speeds in the order of milliseconds for transmitting a message, the above scheme minimises the number of messages required without sacrificing reliability.

6.4 KUDOS IMPLEMENTATION TO DATE

6.4.1 File-store

A file-store has been implemented on two LSI-11/02 micro-computers each with a hard disc, and connected by a Cambridge Digital Communication Ring. The code is written in Modula [WIRTH1] using a compiler developed at the University of York [COTT], with a small number of assembler routines. Including communications software, terminal handling, resource directories, file-server and directory-server the code occupies about 4,000 lines and uses approximately 50 kilobytes of main-memory per node.

The file-store includes all the features described in the following chapter, but a number of limitations exist because of the size constraints of the small machines. For example, only a few directories can be active concurrently or buffer overflow occurs.

The code was written entirely by the author, but thanks are due to Dr O. P. Brereton for her assistance on aspects of the Cambridge Ring Basic Block protocol used to provide node to node communication.

6.4.2 UCSD Filer Interface

A filer for accessing the KUDOS file-store has been written in Pascal for a UCSD Pascal micro-engine [MICROE]. The filer runs as an application program and allows a user

to transfer files between KUDOS and the local storage on the micro-engine. Facilities exist for manipulating directories as well as files.

The code occupies about 1500 lines, but could be shortened considerably. Transfer is slow (less than 1Kbyte a second), but again considerable optimisation is possible. Problems were experienced with the "typing" of UCSD files when storing them as "untyped" KUDOS files, but these are resolved.

6.5 EXPERIENCE AND CONCLUSIONS

6.5.1 Hardware

It is a sobering thought that hardware which was a good buy two years ago now seems out of date. It is now possible to buy 16-bit micro-processor-based systems with 128-kilobyte main-memory, a multi-megabyte disc and an operating system included for around ten thousand pounds or less.

On communications, we are only just beginning to see commercial local area network's. In the UK we see a number of companies developing Cambridge Ring based systems. A token passing ring is used in Apollo. Ethernet is now commercially available.

Ideally any node on KUDOS (or any other local area network system) should be at least the power of a 16-bit processor. The size and speed constraints of 8-bit processor configurations is not cost-effective in terms of the effort required to overcome them.

Moreover, any node for development purposes, should not be seen as a raw machine. An operating system should be seen as part of a node, not something imposed on the hardware. The task would have been much easier if our nodes had, say, a Unix operating system available. In terms of effort required, it is better to adapt rather than create, depending of course on the quality of the system to be adapted. Much time and effort was spent on writing drivers, file-server and support/development software instead of concentrating on the main aspects of research.

This can, however, be said with hindsight. Our original research aims, regarding processing in particular, have been tempered. The decision to take a personal computer approach to providing user processing clarified a number of issues.

Much the same can be said regarding communications, but there is much less off-the-shelf hardware and software available. At Keele work is continuing on an "access logic unit" for the Cambridge Ring, which should provide a node with a transport level service. Presently all communication in KUDOS is handled by a node down to packet level,

consuming much needed cpu cycles and main-memory and providing a slow, crude, inefficient service.

6.5.2 Software

Our main development tool has been Modula, using the compiler developed at the university of York. Experience with the language coincides largely with that of [HOLD]. Essentially it is a great step forward from the use of assembly languages for real-time programming, but has a number of irritating features.

Some of the features (or lack of them) smack of pedantry. For example, without a GOTO statement some forms of error recovery become convoluted with many nested IF's. On occasions it is preferable to use the LOOP construct with a single iteration merely to exploit the EXIT feature. The omission of off-stack storage and pointers, of a FOR statement, of set types and variant records mars a very useful language.

One of the main features of Modula, namely its strong typing, causes a number of problems too. Sometimes it is useful to consider, for example, a status register as an INTEGER and at other times as a word of type BITS. On Cambridge Ring transmission/reception it is sometimes necessary to interpret a packet as an integer, sometimes as a record of two bytes. The York compiler provides a UNIV parameter declaration, which allows a procedure to accept

any type of parameter provided it is of the same size. This feature was used frequently. It would be much better to have a "non-type" variable definition, or be able to declare a variable as having a number of types; limitations would of course exist because of storage representation, say a variable could not be both INTEGER and REAL.

One feature which would have been desirable in Modula is some means of scheduling processes. Modula-2 allows this. The only scheduling in Modula is round-robin with a process relinquishing control only on issue of a wait. This means that a rogue process can hog the processor.

There were no run-time debugging aids available. Locating bugs usually involved tracing procedure calls explicitly using the console, or inspecting the stack, or in moments of complete desperation single-stepping. This adds to the case for adapting an established operating system, rather than writing ones own; debugging software should be more readily available.

In all the writing of a message exchange, resource directories, file-server, directory-server and personal computer filer spanned twelve months. The underlying design evolved gradually in the twelve months prior to this. It would have been preferable to experiment further with the algorithms, particularly with respect to performance analysis. Time constraints forbade this.

A large amount of programming effort was spent on communication difficulties. The Cambridge Ring Basic Block Protocol was adapted for use by the message exchange. Ideally a transport level service such as the Cambridge Byte Stream Protocol would have been used, but the problems of space and performance were considered too great.

Experience strongly indicates that the Cambridge Ring architecture, as it stands, is not suited to this application. A block transfer architecture such as Ethernet would probably be more appropriate. The current Cambridge Ring seems more suited to connecting slow, dumb peripherals to processors, rather than processors to processors. However, one cannot totally ignore the success with which it is utilised in the Cambridge Model Distributed System. It might be useful here to list some notable drawbacks of the Cambridge Ring. One caveat is to remember that similar criticisms can be levelled to some extent to most existing communication systems.

Firstly, a packet on the Cambridge Ring consists of 38 bits, only 16 of which are data. The other bits contain addressing information local to the ring and of little direct use to the user. Thus a 10 megabit/sec ring provides approximately 4 megabit/sec of data transfer. Increasing the size of a packet would improve this ration, but increase the likelihood of waste within a packet.

This does not imply, however, that any two nodes can communicate at 4 megabit/sec. A transmitter can only use one slot at a time, and cannot reuse a slot until it has passed round the ring after returning from a transmission. Thus on an n -packet ring only every $(n+1)$ th packet can be used; at best on a one-packet ring a single node can transmit at 2 megabit/sec.

Further limitations are imposed by the packet assembly/disassembly. Most data flow is in the form of variable length blocks. Consequently a protocol is required to disassemble a block, transmit it word by word, and assemble the block at the receiver. Most block protocols require at least 3 words per block on top of the data to ensure correct transmission. A transport service would require even more overheads.

Packet assembly/disassembly is also a great overhead on a processor if the packets are handled on interrupt, or by polling in the main processor. To free a processor from this, some form of DMA interface to the ring is required. For example, if it takes 100 microseconds to process each packet, then this immediately reduces the data flow to 20 kilobytes/sec, with the processor just dedicated to packet handling.

The Cambridge ring has a fair share policy for slots. A node is guaranteed a slot within a fixed period, dependent on the network configuration. This seems to imply that the

ring will provide a fair share policy on all data flow. This is unlikely to happen in practice.

A receiver can accept packets from any other node on the ring. On block transfers this would mean that two concurrent senders would interleave at a receiver. The receiver would have the added task of inspecting each packet for who sent it before deciding what to do with it. To avoid this a receiver can opt to listen to only one node. This is used by block protocols to remove the interleaving problem. Typically a receiver accepts the first header packet on a first-come first-served basis, then receives only from the sender of the header until all the block is received, then opens its ears for the next header.

Contention for the wire (as in a carrier-sense network such as Ethernet) has been removed, but a more worrying contention for the receiver has been introduced. Simulation results [LUNN1] indicate that an average delay across the ring because of this is in the order of a few milliseconds, depending on the loading of the ring.

Basic block protocols themselves have problems. If a receiver times out a block prematurely, it might accidentally interpret a data packet as a header packet and accidentally swallow a subsequent block which is then lost because of a checksum error. If polling is used for block send/receive then the transmission/reception effectively becomes half-duplex; thus if two nodes send to each other

simultaneously they will both fail, back off and retry. If these retries occur immediately they will fail again, and this can continue indefinitely. These, and other possibilities, are rare but can happen with unfortunate consequences.

The slow communication, and the difficulty of implementing protocols on the Cambridge Ring restricted, to a large extent, the development of the message exchange. A lower latency for message transmission would have allowed waiting for acknowledgements, and the implementation of remote invocation sends, where a send waits for a reply.

However, we encountered the Cambridge Ring in an early stage of its development; add on hardware/software to provide a more powerful interface to the Cambridge Ring may well improve the performance and simplify its use significantly. Such a device is under development at Keele University [BENN].

6.6 OVERALL CONCLUSIONS

Despite certain tactical mistakes, the design and implementation of KUDOS has been a worthwhile exercise. Firstly, it provided a vehicle for the file-store. It also involved some novel ideas, and as such is justified as a piece of research in itself. An interesting question is whether the effort of developing a prototype system prevented the author from concentrating more on the

theoretical aspects. Perhaps proving the algorithms in the mathematical sense rather than the practical sense would have been better. However, the practical problems had lessons of their own and did affect significantly a number of decisions (though not always for the better), and a tangible form of proof is to construct a working model.

It is designed with a local area network architecture in mind, rather than being an ad hoc interconnection of stand-alone systems. However, it does not exclude stand-alone systems from participating. A major principle has been that the hub of the system should be the communication system.

The implementation, on rather restricted hardware, has proved the feasibility of the design, although the performance left much to be desired, especially with regard to communications. Future developments would be more promising if they took advantage of improved hardware, now available at lower cost, together with established software adapted to the KUDOS design.

A significant conceptual division highlighted by KUDOS is that between those components of a system which must be shared among users, and those components which can be provided on a per-user basis. It is the contention of the author that processors can be provided per-user, except for certain specialist applications. The existence of systems such as Apollo, Perq [ICL] and Xerox Star [SMITH] add weight

to this argument.

A component which inherently must be shared is the file-store. Users have a need for common data which is continually updated. It this component to which the rest of this thesis is devoted.

CHAPTER 7

KUDOS FILE-STORE - DESCRIPTION

This chapter describes the KUDOS file-store. There is a distinction made between the file-server in the system and the file-store as a whole. KUDOS has largely ignored the problems of reliability and recovery within a file-server, and utilises algorithms which recover file-server failures through multiple-copy policies.

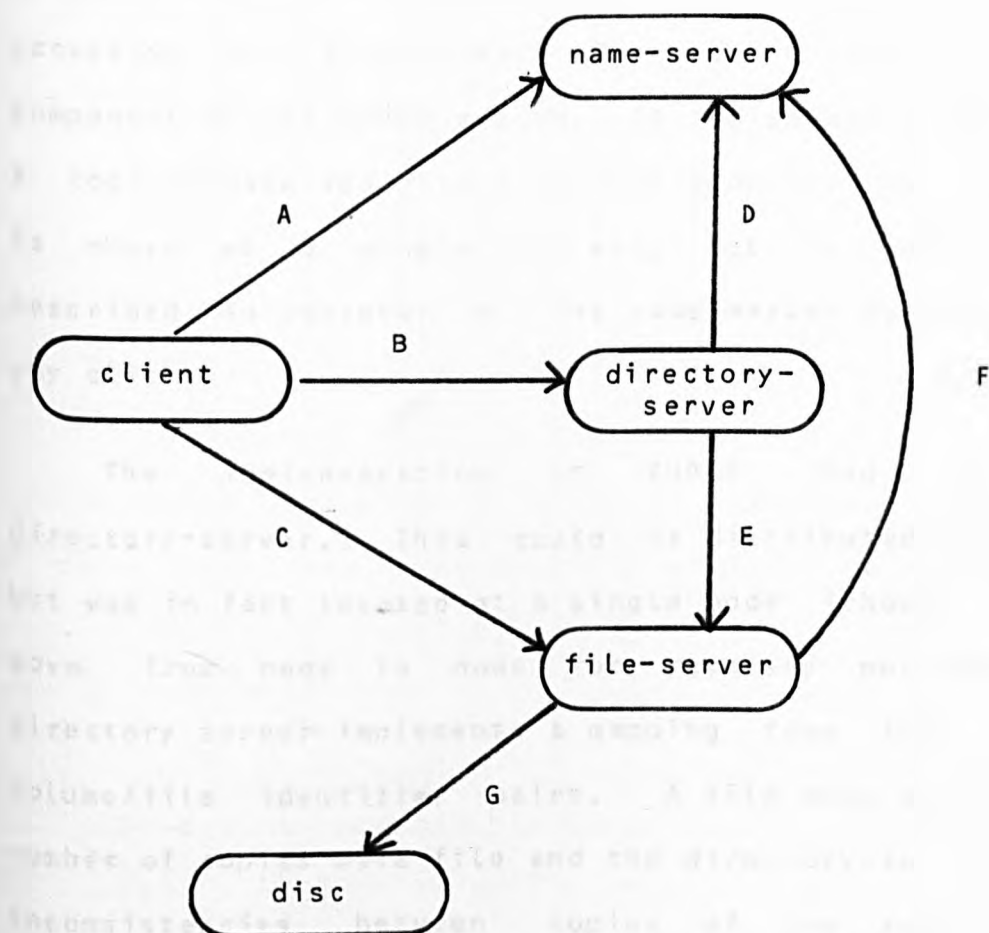
That is not to say that reliability within a file-server is unimportant. An essential feature of KUDOS is that it takes a set of file-servers and combines them to make a file-store which is more reliable than an individual file-server, in terms of the likelihood that a file is lost and the likelihood that a file is available.

The chapter subsequent to this will discuss the KUDOS file-store, highlighting a number of its features and suggesting further avenues for exploration. This chapter will concentrate on a detailed description of the file-store, beginning with the file-server, and then the directory system.

The structure of the file-store is described by the following diagram:



The diagram illustrates the structure of the file-store, showing a hierarchy of nodes and their relationships. The nodes are represented by rectangular boxes, and the connections are shown by lines. The structure is a tree diagram, with a root node at the top left, branching down to several intermediate nodes, which then branch further into a larger number of leaf nodes at the bottom. The lines are very light, and the nodes are represented by simple rectangular boxes.



A - locate directory-server

B - locate/add/delete files

C - file I/O, file creation/deletion

D - catalogue location of directory-server

E - file I/O, file creation/deletion
(for storage of directories and copies of files)

F - catalogue location of volumes

G - physical I/O

A client is any software acting on behalf of a user accessing the file-store. The name-server is a vital component of the KUDOS system. It implements a mapping from a logical name space to a network address. The name-server is shown as a single process, but is implemented as described in chapter 6. The name-server is accessible to any client.

The implementation of KUDOS had a single directory-server. This could be distributed in practice, but was in fact located at a single node (though it could move from node to node for recovery purposes). The directory server implements a mapping from file names to volume/file identifier pairs. A file name may refer to a number of copies of a file and the directory-server resolves inconsistencies between copies of the same file. A protection scheme exists and will be described later.

The file-server provides access to the contents of files, and the ability to create and delete files. Again, a protection scheme exists.

The interfaces were implemented by message-passing, but effectively were remote invocation send (remote procedure call) since each send of a request was followed immediately by a wait for a reply. The interfaces on the above diagram are:

- A - i) find the directory-server
- ii) find the file-server handling a volume

- B - i) locate/delete/add files
 - ii) protect/change protection on files
 - iii) locate/add/delete directories
 - iv) list directories
 - v) lock/unlock files
- C - i) create/delete files
 - ii) read/write contents of files
- D - i) catalogue location of directory-server
- E - as C for storage of directories and copies of files
- F - i) catalogue location of file-server handling a particular volume
- G - i) physical I/O

The file-server and directory-server will be described in detail. A problem exists in how to describe them. The code in MODULA would not help; the message passing scheme proved unwieldy and unclear. Ada has been chosen to describe the interfaces. It provides, through package specifications a way of describing the appearance of a component without disclosing its implementation. It is also widely known, and a language with which the author is familiar.

A package in Ada is a collection of procedures, processes, data types and data objects arbitrarily grouped together by the programmer. A package is divided into a specification and a body. The specification defines precisely those features of a package available to any

Procedure or process using that package, in terms of procedures with their parameters, the data types available, and any data objects available. The package body is the actual implementation of the package, and may include procedures, processes, data types and data objects not visible to a user of the package.

The reader should not worry unduly about the details of Ada. A knowledge of Pascal [JENS] or similar such Procedure-oriented language should be sufficient to make the Ada specification clear, together with the following points.

1. The IN prefix to the type of a parameter indicates that the value of the parameter is used by the procedure, but not changed, equivalent to a non-VAR parameter in Pascal; the OUT prefix indicates that the value passed to the procedure is not used, but the returned value may be set by the procedure; a prefix IN OUT means that the value passed will be used by the procedure and the procedure may determine a different returned value, equivalent in effect to a VAR parameter in Pascal.
2. The only novel feature is the notion of an exception. An exception is a means in Ada of notifying an error; an exception causes the block in which an exception is raised to abort and execute an exception handling routine. A number of standard exceptions exist in Ada, for situations

such as array bound violations or divide by zero.

A programmer can also define and raise exceptions.

Another problem exists in how to describe the algorithms used in the directory-server. A compromise used is to invent a Pascal-like language with extensions for set operations. This combines brevity with precision. Anyone familiar with elementary set theory should have little difficulty in following the descriptions. More appreciation of the problems of software specification might have been useful here (eg [JONES]). The following chapter will discuss the development of KUDOS more fully.

7.1 THE KUDOS FILE-SERVER

7.1.1 Introduction

We consider a file to be a user-arbitrary sequence of bytes. A file-server is a repository for files which provides some name or address for a file, and which allows insertion, deletion and update of files. The file-server may have its own authorisation and protection scheme to restrict access to files stored in it.

How data is arranged on disc depends on the file-server, but can affect performance and reliability. A number of schemes exist. Typical of these is Unix [RITCH] which uses a scheme called i-lists, allowing flexible use of files. Partial recovery is sometimes possible if the disc is corrupted through software or hardware faults; certain erroneous states in the Unix file-store can be rectified by running utilities which check the correctness of the state of the disc, say to ensure that a block is not accidentally allocated to two distinct files. [LAMP] describes a much more robust scheme with redundancy on disc sectors which permits substantial recovery after corruption of part of the disc.

For the initial implementation of KUDOS a very simple scheme was chosen where a file is reserved as a contiguous sequence of disc sectors. All of a file's space must be reserved before it is filled, which is a limitation. Apart from this, however, it provides a neat, economical

file-server which requires at most two disc accesses to retrieve any byte. A cache scheme is used to reduce swapping when a file is sequentially accessed.

The internal structure of the file-server is independent of the KUDOS structure. A file-server could be implemented in different ways, providing a common interface is used.

Protection is by a system of capabilities (or access tokens). To access a file it is necessary to provide the appropriate capability. A file has a number of capabilities associated with it, corresponding to different modes of access (read, write, etc.). Capabilities are randomly generated integers, and a client must remember or find out capabilities in order to gain access.

There is no "structure" to the addressing of a file. Files are denoted by an integer address within a volume, generated by the file-server. Any structure, such as a mnemonic naming scheme, must be imposed by a client, such as a directory-server.

Access to a file-server is available to any client in KUDOS, providing appropriate capabilities are known. A number of file-servers are provided and are addressed by a volume number. A volume number is unique to a volume, and published in the resource directories when the volume is online through a file-server.

7.1.2 File-server Primitives

The primitives used for manipulating files on a file-server are presented here, described using Ada package specifications [ADA]. No description of the implementation will be given, except to justify the interface, either in terms of the positive benefits, or the compromises made in implementation. The file-server is by no means unusual enough to elaborate in great detail.

PACKAGE file-server IS

TYPE capability IS INTEGER;
TYPE file_id IS INTEGER;

PROCEDURE create(size: IN INTEGER;
file: OUT file_id;
read_permit, write_permit: OUT capability);

PROCEDURE delete(file: IN file_id;
write_permit: IN capability);

PROCEDURE expand(file: IN file_id;
write_permit: IN capability;
new_size: IN INTEGER)

PROCEDURE shrink(file: IN file_id;
write_permit: IN capability;
new_size: IN INTEGER)

PROCEDURE read (file: IN file_id;
read_permit: IN capability;
displacement: IN INTEGER;
length: IN INTEGER;
buffer: OUT ARRAY INTEGER OF BYTE);

PROCEDURE write (file: IN file_id;
write_permit: IN capability;
displacement: IN INTEGER;
length: IN INTEGER;
buffer: IN ARRAY INTEGER OF BYTE);

FUNCTION size (file: IN file_id;
read_permit: IN capability)
RETURN INTEGER;

volume_saturated,
no_such_file,
incorrect_capability,
access_outside_file : EXCEPTION;

END file-server;

7.1.2.1 Create And Delete -

To create a file the size of the file must be specified. The file-server returns a file address and two capabilities, permitting subsequent read access and write access to that file.

One benefit, perhaps not too significant, of reserving space before writing is that writes cannot fail because of a volume becoming full. An obvious drawback is that the size of a file is often unknown on creation, and therefore excessive areas of a disc may be needlessly reserved to avoid the failure of a process writing to the file.

To delete a file a process needs to know the write capability. There are arguments in favour of having a delete capability, which would allow a process to be able to write to a file without being able to delete it. This was, however, unnecessary for the purposes of KUDOS.

7.1.2.2 Expand And Shrink -

To overcome the fixed file size, an expand (not implemented), and a shrink (implemented) primitive have been included. It would have been preferable to be able to create empty (zero-sized) files and to have automatic expansion on write. The shrink removes spare from the end of the file; the expand creates space at the end of a file.

7.1.2.3 Read And Write -

Read and write are random access. There is no concept of "file open". Any pointers to within the file must be kept by the client. To read/write data it is necessary to specify the file, the displacement in the file, the amount of data required, and the appropriate capabilities. Files cannot span more than one volume.

Not having a "file open" notion frees the file-server from maintaining contexts, and from worrying about clients which do not complete or which omit to issue a "file close". It does put an imposition on the client wanting sequential access, but this is a small price to pay. The lack of a "file open" also means that any number of files can be accessed concurrently - there is no table of open files to maintain. The file-server is autonomous in the sense that it does not depend on its environment to maintain it in a correct state; it has only one mode of operation, namely file access.

7.1.2.4 Failures -

The exceptions which can be raised are as follows:

1. Volume_saturated (by create and expand). There is insufficient space remaining on the volume to satisfy the create or expand.

2. `No_such_file` (by `delete`, `expand`, `shrink`, `read`, `write`). The file referred to does not exist. Either it has not been created or has been deleted.
3. `Incorrect_capability` (by `delete`, `expand`, `shrink`, `read`, `write`). The capability provided with the operation does not correspond to the one generated on creation of the file. This prevents illegal access by processes not knowing the capability of a file, or mistaken accesses to a file which has been created with the same `file_id` (address) as a previously deleted file.
4. `Access_outside_file` (by `read`, `write`). The operation tried to read from or write to a data area beyond the end of the file specified on creation or on the most recent shrink or expand.

7.1.2.5 Desirable Extensions To The Primitives -

The primitives implemented are clearly not ideal. In particular, the need to reserve a file's data area on creation is very limiting. However, the primitives were adequate for the implementation of the directory-server and provision of remote archive filing for a UCSD Pascal micro-engine.

A particular deficiency of the above file-server is the lack of atomic write to files. Such a facility would have been most useful in the directory-server implementation. More useful would have been a recoverable transaction at file-server level which spanned a number of files, with full commit/rollback capability. A transaction which spanned more than one file-server would have been even better.

It is arguable that mutual exclusion should be implemented at the file-server level, though we shall not argue the case here; the granularity of lock implemented in the directory-server might have been finer had it been implemented at the file-server level, though problems of synchronisation might have arisen.

It is the author's belief that much useful and highly practical work could be done on the provision of a better interface to files than is commonly implemented. Many systems enforce an artificial structure on files, such as fixed length blocks corresponding to disc sectors, sometimes as variable length records. Unix has overcome a number of these problems by defining a file as a sequence of bytes with no other apparent structure. Certain other features would possibly improve the versatility of file systems, such as the ability to insert arbitrary length strings in the middle of a file, or to delete arbitrary length strings in the middle or at the beginning of a file.

Perhaps the sort of interface to a file most desirable is not a simple sequential or random access read, but more on the lines of a text editor, with roll-back capability. This sounds like wishful thinking, but this approach might be highly profitable for remote access to files where a minimum of data exchange between a client and a file-server is desirable.

There also exists a large class of files with a record basis, used commonly in data processing applications. Indexed sequential files are a widespread example, where a record in a file can be randomly accessed on the value of specific key fields. A philosophy typified by Unix is that such files can be built on top of a random-access byte-oriented file, though efficiency and performance considerations cast doubt on the advisability of such an approach. Such files are largely being replaced by database systems, though a number of database systems currently advertised turn out, on closer examination, to be little more than indexed or hashed file maintenance systems.

7.1.3 File-server Protection

Files are protected by two capabilities. A capability here is used in the restricted sense of a "key" or "token" permitting access, not in the wider sense understood in capability-domain architectures [CASEY]. The read capability allows inspection but not alteration of a file. The write capability allows alteration.

Capabilities are randomly generated integers (current implementation size 16 bits). With a 16-bit capability there is a 1 in 65,536 chance of guessing it correctly. Greater security can be gained by increasing the size of the capability. A client permits access to a file created by it only by passing the appropriate capability to another client.

A client must remember capabilities in order to retain access to files. There is no direct means of finding a file's capabilities. If a capability is lost, then effectively so is the file. Garbage collection, in conjunction with the directory system, is possible, by marking all files with a directory entry and deleting all others.

Note that there is no official ownership of files at this level. However, knowledge of a capability could be considered a form of ownership. A client can restrict access to a file by not disclosing its capabilities. A client can also permit limited access by releasing only the

read capability.

The notion of ownership was largely ignored in KUDOS. One might argue that in many areas a system of ownership is irrelevant. Ownership was considered a separate issue from the problems being tackled. However, for many practical applications an ownership scheme would be necessary, especially where accounting is needed.

7.1.4 File-server Reliability

Three aspects of reliability should be considered here. Firstly there is continuity of service, secondly recovery from breakdown, and thirdly atomicity of update.

Continuity of service is largely dependent on hardware reliability, and there is little the software engineer can do to prevent it. Of course, there is the possibility of software failure, but it is the naive assumption of this thesis that software performs to specification.

Recovery from breakdown is well within the software engineer's scope. Advantage can be taken of the physical structure of a disc to reduce the damage caused by a head crash for example. Moreover, good software will prevent failure leaving the disc in indeterminate state.

[LAMP] and [REDE] describe a scheme where each block of a file is stored together with its file identity and displacement within a file. Directories exist for normal access to a file, but in the event of damage to a directory it can be reconstructed, or partially reconstructed, from the actual data blocks of files. Furthermore, each block is checked on access to see that it is the correct block for that file. This is a form of backward error recovery using protective redundancy.

Another scheme using redundancy is described in [STURG]. Here, certain critical data is stored on two different surfaces, so that in the event of a head crash at least one copy will remain undamaged.

An important software feature is the avoidance of or recovery from broken contexts. If a file-server fails and recovers without a client detecting the failure, then there must be nothing of importance altered by the recovery which would harm the client. This is one reason for removing the open/closed concept from the KUDOS file-server. A recovery of the file-server does not harm any operations which do not occur during recovery, and any operations during recovery will detect failure.

Atomic read/write would be useful to many clients, in the sense that a read or write either completes successfully or has no effect at all. The KUDOS file-server does not implement atomic read/write, because of the implementation used. This is a serious omission. One solution is to reduce a file write to a single disc access which is assumed to be atomic. Another solution is to use careful replacement strategies for update within a file, allowing roll-back if a write fails, and a distinct point (the commit point) after which roll-forward is guaranteed. Intention lists provide this [STURG]. Under the current implementation, the failure of a disc during a write which spans more than one block can cause only part of the write to succeed.

KUDOS file-server commands return a success/fail condition, expressed as exceptions in the above Ada specification, though in an actual Ada program the exception would have a wider implication. If the condition is received by a client, then it knows the state of the file. If the condition does not arrive, then it knows the file is indeterminate, and can take corrective action.

Because read and write are random access rather than sequential, a read or write can be repeated without side-effects; that is, they are idempotent. A repeated read could cause problems if a reply was delayed rather than failed, but this could be handled by the communications protocol.

The topic of this thesis, however, is not to consider reliability of an individual file-server, but to construct out of a number of file-servers a file-store which is more reliable than an individual file-server.

7.2 THE DIRECTORY SYSTEM

7.2.1 General Description

7.2.1.1 Hierarchy -

KUDOS provides a single hierarchical file-store using redundancy to improve reliability and continuity of access for file storage. The hierarchy is similar to that of Unix [RITCH], but without links (more than one path-name for the same file).

A root directory exists, which is replicated on all volumes. The root directory contains subdirectories (not necessarily replicated on all volumes). Each subdirectory in turn can have subdirectories of its own. All directories can also contain files.

Files and directories are named according to the Unix convention. A name within a directory is an arbitrary sequence of characters. The root is referred to as "/". The directory "usr" in "/" is referred to as "/usr". The directory "ken" in "/usr" is referred to as "/usr/ken". The file "modules" in "/usr/ken" is referred to as "/usr/ken/modules".

To access a file it is necessary to activate all directories in the path to that file. Activation of a directory involves creating a process to handle operations on files in that directory. Such a process is called a directory manager, and is created on behalf of a client by

the parent directory's manager. The manager of root is permanently active.

The address of the root manager is published in the resource directory, and any client can access root. To activate any directory the whole path-name can be given to root. Alternatively, if a directory is known to be active, a sub directory (or sub sub directory, etc.) can be activated by providing the path-name from that directory to the directory manager.

The manager of a directory is responsible for all operations on files in that directory, for ensuring consistency of various copies of that directory, and for deactivating itself once a series of transactions are complete.

The manager of a directory is also responsible for multiple client access to the directory. Only one manager is allowed to exist for any one directory at one time. It must therefore provide features for mutual exclusion, such as locks.

7.2.1.2 The Active File-store -

We can now make a useful distinction. We shall call the active file-store the hierarchy of directories and their contained files which have currently active managers. The dormant file-store is all the rest. Files can only be accessed through the active file-store.

We can conclude that it is only necessary to ensure that the active file-store is consistent and up-to-date. Inconsistencies can be permitted in a dormant part of the file-store until that part is activated.

In a large file-store with many users the active file-store is likely to represent a small proportion of the total file-store. Certain areas of the file-store are likely to be accessed more often than others. More frequently used areas should be held active to reduce the overheads of activation. This is achieved by holding a directory active for a fixed period after any operation on a file; thus a heavily used directory will not deactivate.

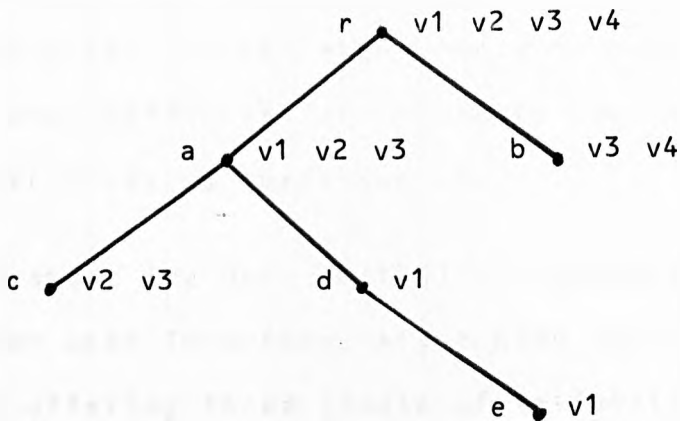
7.2.1.3 Associated Volumes -

Each directory has a set of "associated volumes". The associated volumes of a directory are exactly those volumes on which any file in that directory is replicated. The directory too is replicated on all its associated volumes.

The set of associated volumes of a directory must be a subset of the associated volumes of its parent directory. This means that the associated volumes of root must be all the volumes in the system. Thus if a directory has three associated volumes, then any file stored in that directory is replicated three times. Any sub directory can have at most three associated volumes. There is no point storing a directory on volumes where it does not keep files. Thus a

directory is only stored on its own associated volumes, not those of its parent.

The overlay structure is illustrated by the following diagram:



The root (r) has associated volumes v1 v2 v3 v4. Directory a has associated volumes v1 v2 v3, b has v3 v4, c has v2 v3, d has v1 and e has v1.

Consequently a volume contains the full path to any files held by it; if a volume is online its contents are accessible. This makes volumes independent of each other for purposes of access.

Originally considered was the weaker condition that the associated volumes of a directory could be any volume in the system. This allowed greater flexibility, but meant that some files on a volume could only be accessed when another volume was online. The only other envisaged solution to that problem was to replicate all directories on all volumes; this was considered too prohibitive.

Associated volumes allow tailoring of the file-store with respect to reliability to meet varying needs. For example, a user might be allocated three associated volumes in his initial directory. He can then create sub

directories with one, two, or three associated volumes. None-critical files might only be kept as one copy, those which are difficult to recreate as two copies, highly critical files as three copies.

A shell (re Unix [RITCH]), command-language interpreter or other user interface, might hide the details from a user, simply offering three levels of reliability (or n-levels), the level to be specified by a user at directory creation, perhaps decided by default when not specified. Accounting and charging techniques could be used to prevent indiscriminate use of replication.

The overhead of keeping the full path to all files on a volume is minimal. Typically a volume will contain only a handful of extra directories, reflecting a tiny portion of its storage capacity. This is easily outweighed by the benefits of independence of volumes, allowing access to all files on a volume if that volume is available.

7.2.1.4 Mount/dismount -

When a volume comes online its file-server publishes its volume number in the resource directories. There is no explicit mount procedure apart from this. It is the responsibility of the directory server to detect its presence and to act accordingly.

Removal of a volume may cause problems for a directory manager. The directory manager should detect the absence of the file-server when it fails to respond, and act accordingly. This may mean aborting some processes, but the manager could recover without rolling back. Currently if an associated volume fails the directory manager backs out.

The current implementation does not acknowledge a volume until the active file-store diminishes to just root. This is not necessary, but was a compromise. Volume removal also causes a dependent directory manager to back out, aborting any incomplete transactions. Again, this is not necessary, but was a compromise.

In principle it ought to be possible to dismount a volume, update it, and reintroduce it to the system. The system and the volume would then reconcile each other. This could be the technique used for recovering from a volume crash. The repaired volume could be restored to a previous state, and it would eventually catch up with the rest of the system. Alternatively, it could be restored with an empty root, and it will restore itself from copies on other volumes, except for files which had only one copy and which were kept on that volume.

7.2.1.5 Directory Resolving -

We assume a strictly increasing timestamp universally available throughout the system. When a file is stored through the directory system, the filename together with the timestamp is stored in each copy of the directory.

On activation, all online copies of the directory are inspected. If a copy is out of date, this is detectable by comparing file for file the individual timestamps. If one copy refers to a more recent file the other volumes can be forced to catch up, by copying the newer file onto the out-of-date volumes and inserting the new timestamp in the out-of-date directories.

A problem can occur when a file is deleted from one copy of a directory whilst another copy is offline. The solution to this is to place an assassin in a directory if all associated volumes are not online. If the assassin is stored with the timestamp of the file deleted it can safely remove that file from other directories at a future resolve. Creation of a more recent file will destroy the assassin.

Directories are not stored with a timestamp - it is not sensible to copy directories on resolving the parent. Neither can an assassin be left for a directory. One condition of the system is that a directory can only be deleted if it is empty. This can make directories very difficult to delete if their associated volumes are repeatedly offline. There is no apparent simple solution to

this.

Suppose X_1, X_2, \dots, X_n are n copies of directory X stored on n distinct volumes. Suppose that for each file x in X_j ($1 \leq j \leq n$) that $x.T_j$ is the timestamp on insertion of x in X_j . Then the algorithm for resolving X_1, X_2, \dots, X_n is.

```

FOR EACH  $x$  IN UNION( $X_j, j=1..n$ ) DO
     $T = \text{MAX}(x.T_j, j=1..n)$ ;
    SELECT  $i$  FROM  $1..n$  SUCH THAT  $x.T_i = T$ ;

    FOR EACH  $j$  IN  $1..n$  WHERE  $x.T_j < T$  DO
        COPY  $x$  IN  $X_i$  TO  $x$  IN  $X_j$ 
    END;
END;
```

The algorithm for resolving using assassins is

```

FOR EACH  $x$  IN UNION( $X_j, j=1..n$ ) DO
     $T = \text{MAX}(x.T_j, j=1..n)$ ;
    SELECT  $i$  FROM  $1..n$  SUCH THAT  $x.T_i = T$ ;

    IF <assassin for  $x$  in  $X_i$ > THEN
        IF <all associated volumes online> THEN
            FOR EACH  $j$  IN  $1..n$  DO
                 $X_j := X_j - \{x\}$ 
            END;
        ELSE
            FOR EACH  $j$  IN  $1..n$  DO
                <place assassin for  $x$  in  $X_j$ 
                and set  $x.T_j$  to  $T$ >
            END;
        END IF;
    ELSE
        FOR EACH  $j$  IN  $1..n$  WHERE  $x.T_j < T$  DO
            <remove any assassin for  $x$  from  $X_j$ >
            COPY  $x$  IN  $X_i$  TO  $x$  IN  $X_j$ 
        END;
    END IF;
END;
```

Note that the assassin will not delete a file later than the one it was set to assassinate, and if a later file exists

then the assassin is removed from all online directories.

The resolving of directories is the responsibility of the manager, and the resolve must complete before any operations on files in that directory are allowed to proceed. This should ensure that a directory remains up to date. It is possible, however, to envisage circumstances where old copies of files could be presented to a user, say in a two volume system where the first volume is offline one day, the second volume offline the next, then the first volume offline the next.

Extra conditions could be imposed on resolving. One could be that a resolve can only take place when the majority of associated volumes is online. This means that at least one of the associated volumes contains an up-to-date copy of the directory. This reduces the availability of directories, but for three copy directories the availability should be greater than that of a single volume. A deeper discussion of this will take place later.

7.2.1.6 Timestamping -

The notion of a timestamp is important in distributed computing, but there are a number of problems associated, particularly with regard to synchronisation (see[LAMP0]).

The Cambridge Model Distributed System [HERB] has a single time-server. This reduces the autonomy of other nodes. A more sensible solution is to have a clock in each node which is kept running by a backup battery even when the node is not operational. Provided these are not used for inter-process synchronisation they ought to be suitable for most applications with even a poor synchronisation of within a few seconds. Thus crude, periodic synchronisation would be adequate.

Since only one manager for a directory is allowed in KUDOS, provided that manager always consults the same clock, and that clock is accurate to within a few seconds, there should be no problems. The KUDOS prototype did not have a clock available, and simulated the effect by a simple counter incremented on each read.

Alternatively, any strict monotonically increasing integer would suffice as a timestamp for a directory, and could be stored in each copy of a directory, the maximum one among online copies being used. This may cause problems if a directory can be activated with a minority of online associated volumes.

7.2.1.7 Atomicity -

To provide recoverable update to files, KUDOS enforces careful replacement. To update a file a client must first make a copy of that file, update the copy and replace it in

the directory system. When the directory-server receives a file to be inserted in the system it assumes that the file is correct; i.e. file contents are the responsibility of the client.

As stated before, our implementation does not provide an atomic write to a file. However, it does provide one feature which allows atomic update of a complete file. Given two file pointers (or addresses) in a file-server there exists a command to make the first file pointer refer to the contents of the second file. This is done by a single sector write to disc which is assumed in KUDOS to be atomic; this assumption is weak, and improvements in the file-server are required to remove it.

Using this feature it is possible to provide an atomic insert of a file in a directory by creating a new copy of a file, creating a new copy of the directory and if all operations succeed so far only then to swap the old copy of the directory for the new copy. Otherwise the old copy remains, so that the insert is atomic.

Delete file should also be atomic, since this again involves only one write to disc. Deletion of a directory however can cause problems if some associated volumes are offline. It might be sensible to insist that a directory can only be deleted when all associated volumes are online. This is an unfortunate niggle, but directories are typically not deleted very often.

The assumption that a single write to disc is atomic is somewhat risky. A disc write can partially complete, leaving useless or unreadable data in the disc sector. Schemes for increasing the reliability of disc update exist, such as stable storage in the Xerox DFS. A major deficiency of the KUDOS file-server is its lack of a truly atomic write, and this must be seen as an important improvement.

7.2.2 File Protection

7.2.2.1 System Defined Capabilities -

The file-server provides two capabilities for file access - a read capability and a write capability. To inspect a file in the file-server a client must quote the read capability; to update the file a client must quote the write capability. The directory-server utilises these capabilities to protect the files it refers to. To insert a file, the directory-server makes a new copy with new capabilities. These capabilities are recorded in the directory.

If a client wishes to inspect a file, the directory system will divulge the read capability, but not the write capability. A client of the directory-server can therefore inspect a file in a directory but not update it. To update a file a client must create a new copy in a file-server and request the directory-server to replace the old copy on its behalf. Thus the directory-server enforces a policy of careful replacement. This will only be done if the directory server discerns that the client has the appropriate authority.

The capabilities of a directory are never divulged. Otherwise a client could inspect a directory and gain illegal access to the contents of a file. The directory-server will release other information on its contents freely, such as names and timestamps.

7.2.2.2 User Defined Capabilities -

On inserting a file a client can provide two further capabilities, a user-defined read capability, and a user-defined write capability. These are stored in the directory, and any request for information on that file must be accompanied by the appropriate user-defined capability.

The directory system will not delete nor overwrite a file without the user-defined write capability, nor will it provide inspection rights without the user-defined read capability. If no user-defined capability is provided on initial insert then any client can access files through the directory system.

The user-defined capability is the principle data protection mechanism for a user. A simple mapping can be arranged from a mnemonic password to a capability. A user can allow general read access by not imposing a user-defined read capability, or general write access by not imposing a user-defined write capability. Alternatively a user can allow limited sharing by informing a restricted set of users of the appropriate passwords.

All of a single user's user-defined capabilities might be identical or might vary. It is possible to change these capabilities, which would be seen by the user as a change of password.

Sophisticated sharing mechanisms might be handled by storing user-defined capabilities in protected files. Elaborate schemes for group access could be implemented through special client software. However, this approach to ownership is probably too inflexible and needs more careful consideration, and ignores such problems as the need for accounting in a shared system.

Directories have no user-defined capabilities. It is therefore not possible to prevent access to a whole subtree without protecting each individual file. This is because of the special nature of directories, but careful thought should provide some means of providing such protection if it was thought necessary. However, this would probably involve extra capabilities being handled by directory managers for each directory operation.

7.2.2.3 Locking -

In a multi-access system it is necessary to provide some means of mutual exclusion for access to shared data. Usually this is by locking files. KUDOS provides a locking mechanism on files which allows multiple reads or single write (but not both).

To prevent locks being held indefinitely for a failed client, locks must be refreshed by a client periodically. A lock which is not refreshed within a set period can be broken by another client.

A successful file lock returns a lock capability. This capability must be quoted when refreshing the lock, removing the lock, or in the case of a write lock when inserting or deleting a file.

It is permitted to issue a write lock for a file which does not exist. This prevents clashes of names on inserting files under a new name.

To update a file it is necessary to issue a write lock, and to provide the write lock with the update request. It is the responsibility of the client to use the lock sensibly. That is, the client is expected to lock all appropriate files before updating, rather than updating and then issuing the locks.

A more lenient approach to inspecting files has been taken. A read lock exists, which excludes writes to a file, but a file can be inspected without a read lock. Most inter-active file inspections are not so critical as to require the overhead of locking.

The locking and update of files in KUDOS are major deviations which prevent KUDOS being used for database applications. The unit of transaction is a file. Any usable database would require locking and update at record level, permitting concurrent update of a file by many clients.

Although no attempt at describing a database extension of KUDOS is included in this thesis, it is speculated that the solution through KUDOS would be to spawn a file manager to control access to a file (which may be replicated). This file manager would have similar responsibilities to a directory manager. A file manager, however, must impose a structure on a file, whereas the inner echelons of KUDOS consider files as arbitrary sequences of bytes. Such a file manager would utilise a file-server in much the same way as a client. Locking of records under replication would be done at the manager rather than at file-server level. There could only be one file manager for a given file at any one time.

There would then have to be two classes of files interpreted by KUDOS. Database files would not be accessible to normal clients (other than a file manager). This could easily be achieved by use of capabilities. Resolving database files would also be handled differently - there is no need to copy the whole file, just the inconsistent records.

7.2.3 Directory System Primitives

As with the file-server primitives, the interface to a directory manager is described using Ada packages. The implementation was actually message-passing using Modula.

PACKAGE directory_manager IS

```
TYPE capability IS INTEGER;
TYPE file_id IS INTEGER;
TYPE path_name IS ARRAY INTEGER OF CHARACTER;
TYPE name IS ARRAY INTEGER OF CHARACTER;
TYPE resource_address IS RECORD node, port: INTEGER END;
TYPE volume IS INTEGER;
TYPE volume_list IS ARRAY INTEGER OF volume;
```

```
PROCEDURE activate_subdirectory
    ( directory: IN path_name;
      directory_manager: OUT resource_address;
      permit: OUT capability);
```

```
PROCEDURE read_associated_volumes
    ( permit: IN capability;
      volumes: OUT volume_list);
```

```
PROCEDURE create_subdirectory
    ( permit: IN capability;
      directory: IN name;
      volumes: IN volume_list);
```

```
PROCEDURE delete_subdirectory
    ( permit: IN capability;
      directory: IN name);
```

```
PROCEDURE insert_file
    ( permit: IN capability;
      file-server: IN volume;
      file: IN file_id;
      file_read_permit: IN capability;
      lock_permit: IN capability;
      user_write_permit: IN capability;
      user_read_permit: IN capability);
```

```
PROCEDURE find_file
    ( permit: IN capability;
      filename: IN name;
      user_read_permit: IN capability;
      file-server: OUT ARRAY INTEGER OF volume;
      file: OUT ARRAY INTEGER OF file_id;
      read_permit: OUT capability);
```

```
PROCEDURE delete_file
    ( permit: IN capability;
      filename: IN name;
      user_write_permit: IN capability;
      lock_permit: IN capability);

PROCEDURE read_lock
    ( permit: IN capability;
      filename: IN name;
      user_read_permit: IN capability;
      lock_permit: OUT capability);

PROCEDURE write_lock
    ( permit: IN capability;
      filename: IN name;
      user_write_permit: IN capability;
      lock_permit: OUT capability);

PROCEDURE refresh_lock
    ( permit: IN capability;
      filename: IN name;
      lock_permit: IN capability);

PROCEDURE end_lock
    ( permit: IN capability;
      filename: IN name;
      lock_permit: IN capability);

PROCEDURE list ( permit: IN capability;
                 name_list: OUT ARRAY INTEGER OF name);

PROCEDURE new_read_permit
    ( permit: IN capability;
      old_user_read_permit: IN capability;
      new_user_read_permit: IN capability);

PROCEDURE new_write_permit
    ( permit: IN capability;
      old_user_write_permit: IN capability;
      new_user_write_permit: IN capability);

name_in_use, name_locked, not_a_directory,
directory_not_empty, name_not_locked, no_such_file,
cannot_lock_directory, name_write_locked,
name_read_locked, illegal_permit:

EXCEPTION;
```

END directory_manager;

The managers are actually interfaced through the message passing mechanism. Thus the "activate" primitive returns the address to which operations on the newly created manager should be sent. A manager generates a capability which allows operations on the directory, and any client must provide the appropriate capability to gain access; this is largely an accident-prevention mechanism to stop a manager with the same address as an old deactivated manager being accessed as if it were the old manager.

A set theoretic description of the algorithms will be given below. X is the directory and X_1, X_2, \dots, X_n the online copies of the directory. R is the set of files read locked in the directory X , and W the set of files write locked. For each x in R , $x.count$ is the number of read locks on x . For each x in X , $x.directory$ is true if x is a directory. To simplify the presentation, capability checking has been omitted, but it is a trivial exercise for the reader to include capability checking in the algorithms. The raising of an exception causes immediate abort of the operation.

7.2.3.1 Activate_subdirectory -

The activate command can be issued to any directory manager. It specifies the path-name from that directory. The activate may in fact not have to create a directory manager if the directory is already active, but it is necessary to obtain the address of the directory manager. The activate will in fact activate all directories on the

path. A directory deactivates when all subdirectories have deactivated and no file transactions are current. No explicit deactivate is available to clients.

The actual implementation of directory activation depends largely on the process creation mechanism of the system. Under the Modula implementation a pool of processes were held waiting for a directory for them to activate. This was a poor solution enforced by the fact that Modula does not garbage-collect space used by a terminated process. Under Unix, the fork mechanism of process creation, where a process effectively duplicates itself, would have been ideal; Ada tasking is another attractive option. There is no requirement, however, that a directory manager should run on the same node as its parent manager.

The sequence of actions a manager takes before allowing operations are:

manager:

```
<locate all online associated volumes>  
<read the online directories X1,...,Xn>  
<resolve X1,...,Xn>
```

```
WHILE <active request within timeout limit>  
  DO <service request>;
```

END manager;

The associated volumes of a directory are recorded in the parent directory.

7.2.3.2 Read_associated_volumes -

This primitive returns the list of associated volumes of the directory. This is necessary for a client wishing to create a subdirectory or wishing to know the level of replication of a directory.

7.2.3.3 Create_subdirectory And Delete_subdirectory -

Creation and deletion of directories is straight forward. On creation empty subdirectories are written to the associated volumes of the subdirectories. There is no distinction between directory and file names. A directory and a file with the same name are not allowed simultaneously. A directory can only be deleted when it is empty, and only when all associated volumes are online.

<create directory d>:

```
IF d IN X THEN RAISE name_in_use;
IF d IN W THEN RAISE name_locked;

FOR EACH i IN 1..n DO
    <create empty subdirectory d in Xi>
END;
```

END <create directory d>;

<delete directory d>:

```
IF NOT (d IN X) OR NOT d.directory
    THEN RAISE not_a_directory;
IF <d not empty in each of X1,...,Xn>
    THEN RAISE directory_not_empty;

FOR EACH i IN 1..n DO
    <delete d in Xi>
END;
```

END <delete directory d>;

7.2.3.4 Insert_file -

To insert a file a client must provide a new copy of the file, and the appropriate capabilities. The client's copy of the file is unchanged and remains the client's property. The directory server takes its own copies.

A file cannot be inserted with the same name as an existing file; to insert a file it is necessary to delete the old copy. The file to be inserted must be locked and the appropriate lock capability provided.

<insert file x>:

IF x IN X THEN RAISE name_in_use;

IF NOT (x IN W) THEN RAISE name_not_locked;

<set T to current time>

FOR EACH i IN 1..n DO

COPY x TO x IN Xi WITH x.Ti=T;

END;

END <insert file x>

7.2.3.5 Find_file -

This provides a client (with the correct capabilities) with read-only information on a named file. It is up to a client to decide whether or not to issue a read lock. Without a read lock it is possible that the directory-server will scratch that file because of update by another client.

<find file x>:

IF NOT x IN X THEN RAISE no_such_file;

IF x.directory THEN RAISE no_such_file;

<return address of copies of x>

END <find file x>;

7.2.3.6 Delete_file -

To delete a file it must be write locked.

<delete file x>:

IF NOT (x IN W) THEN RAISE name_not_locked;

IF NOT (x IN X) THEN RAISE no_such_file;

IF <all associated volumes online> THEN

FOR EACH i IN 1..n DO

Xi=Xi-{X};

END;

ELSE

FOR EACH i IN 1..n DO

<place assassin for x in Xi>

END;

END IF;

END <delete file x>;

7.2.3.7 Read_lock, Write_lock, Refresh_lock And End_lock -

The locks are straight forward. The refresh_lock refreshes a lock. If a lock is not refreshed within a fixed period it can be broken by another client after a specified time limit. End_lock removes a lock. A file can have one write lock or a number of read locks, but not both a read and a write lock.

Without refreshing and timeout the locking algorithms are as follows. Implementing timeout is a relatively painless extension, involving storing the lock time in R or

W and updating it on refresh.

<read lock x>:

```
IF NOT x IN X THEN RAISE no_such_file;
IF x.directory THEN RAISE cannot_lock_directory;
IF x IN W THEN RAISE name_write_locked;

IF x IN R THEN x.count:=x.count+1
ELSE BEGIN R=R+{x}; x.count:=1 END;
```

END <read lock x>;

<write lock x>:

```
IF (x IN X) AND x.directory THEN
    RAISE cannot_lock_directory;
IF x IN R THEN RAISE name_read_locked;
IF x IN W THEN RAISE name_write_locked;

W:=W+{x};
```

END <write lock x>;

<end lock on x>:

```
IF x IN R THEN
    BEGIN x.count:=x.count-1;
        IF x.count=0 THEN R:=R-{x};
    END
ELSE IF x IN W THEN W:=W-{x}
    ELSE RAISE name_not_locked;
```

END <end lock on x>;

7.2.3.8 List -

Directories are not directly accessible to a client, preventing illegal access to their contents. To find out the contents of a directory a special primitive is provided. The current implementation returns only the names of the files, but could well return information such as the date of insertion of the file.

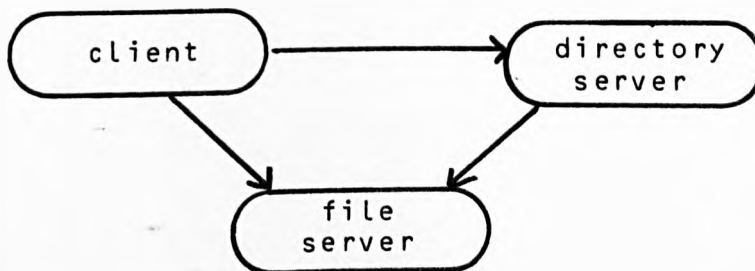
7.2.3.9 New_read_permit And New_write_permit -

These allow a client to change the user-defined capabilities. This can be seen as the equivalent of a password change.

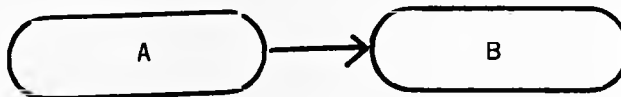
7.3 CLIENT VIEW OF FILE-STORE

7.3.1 File-server/directory Relationship With Client

As stated before, the directory-server is a client of all file-server in the system. Diagrammatically this can be seen as



where



means A is a client of B.

This is a sensible division, since the client-file-server interface can be tailored for fast, efficient access, and the client-directory interface tailored to provide reliable, secure storage.

Access to the file-server means that a client can use scratch files without worrying about naming conventions. A major deficiency in Unix is the lack of a scratch file mechanism; all files in Unix have to be named.

The dichotomy between file-server and directory-server also permits more flexibility. In principle more than one directory system could be implemented on the same set of file-servers.

7.3.2 Client Error Detection/Recovery

All KUDOS file-store commands return a result. Not all KUDOS commands are, however, atomic under current implementations. Only inserting and deleting files and changing user-defined capabilities in a directory are atomic.

A client must necessarily detect errors. A KUDOS command which does not return a result may or may not have completed. Idempotent commands, such as file read and write can be repeated. Some instructions, such as locking, have side-effects and cannot automatically be repeated. In many cases the client must test to see whether or not a command completed.

How clients recover from failures both in KUDOS and in themselves is dependent on themselves. KUDOS merely promises to behave consistently, and to inform of failures where possible. We shall discuss later how, by better design, KUDOS might have assisted error recovery in clients.

7.3.3 Deadlock Detection/recovery

The timeout on a lock is the mechanism for deadlock avoidance in KUDOS. No deadlock detection is provided, nor is queueing for a lock. It is up to a client to respond to a file which is locked by another client, say by retrying after a certain interval.

A rogue client could cause deadlock by repeatedly refreshing a lock on the file. KUDOS cannot detect this. Guidelines can be given for the design of clients but if a client does not adhere to them there is little KUDOS can do.

The principle guideline for deadlock avoidance is - in a transaction involving multiple files do not refresh a lock until all files are locked; if a subsequent refresh fails then back out the transaction. Otherwise, refreshing a lock before all locks are made effectively locks a file indefinitely and causes potential deadlock with two clients waiting for each other.

7.3.4 Example Of Use Of File-store

(input of a text file)

Let us suppose a user wishes to input a text file with the name `"/usr/ken/thesis"`. That user has a personal work-station which is a client of the KUDOS file-store, and which has no local file storage.

The personal work-station must first locate a file-server (any will do) in the resource directory system. The personal work-station will then create a file in a file-server of a default size larger than the expected text file. Then, using a series of writes place the text file sequentially in the file-server. At the end of the input, the personal work-station then shrinks the file in the file-server to the correct size.

The personal work-station then locates root through the resource directories, and activates `"/usr/ken"`. It then write locks the name `"thesis"`, and inserts the file, and removes the lock. Once the file is inserted, the original file-server copy can be deleted.

If any of the directory requests fail, the user can be informed, and corrective action requested. For example `"/usr/ken/"` may not exist. Note that the file-server copy is untouched and the user will not have lost any data.

The personal work-station might periodically insert the input text in the directory /usr/ken every few minutes, to avoid loss of data if the file-server crashes. It may also issue the write lock at the beginning of the transaction and hold it (by refreshing) until the transaction is complete, thus preventing conflict with another user trying to update that file; this policy is probably safest in general.

CHAPTER 8

KUDOS FILE-STORE - ASSESSMENT

This chapter assesses the KUDOS file-store in terms of its success as a design, as an expression of new ideas, and as a lead in to future research. The design will be assessed principally in terms of its reliability and performance. The new ideas are mainly centred around the overlay mount scheme and the the notion of an active file-store, based on associated volumes and directory managers.

8.1 THE DESIGN

The major goal in the design of KUDOS was to provide a file-store which is reliable in terms of the high likelihood that a file is access, and the low likelihood that a file is lost. Performance aspects were not considered a major part of the design aims, but of course performance cannot totally be ignored. In this section we shall first discuss the structure of KUDOS, with respect to what lessons can be learned, especially with respect to its distributed nature. The reliability aspects of KUDOS will be examined. Finally,

some comment will be made on performance.

8.1.1 Structure And Methodology

The author has a growing belief that the structure of a system is more important than the finer details. KUDOS was developed to prove (in the practical sense) a set of algorithms; as such it largely suffered as an operating system design. A good set of underlying algorithms does not necessarily imply a successful system. The file-store in particular would have benefitted from a more abstract approach.

Good design is difficult to measure. If the designer follows a methodology there is the criterion of how rigourously that methodology was applied. A methodology itself has criteria for assessing the quality of a design. However, how does one compare two designs derived using two different methodologies? Is a design developed without a recognised methodology necessarily bad?

Certain principles of good design have gained popular approval. For example, it is widely held that the use of GOTO in programming can lead to code which is difficult to understand and maintain, as well as permitting dangerous habits to be developed and applied [DIJK]. The question of style is raised by [KERN]; it is argued that clarity and simplicity are of more importance than efficiency in the design of a program.

It is worth exploring what the development of KUDOS and its file-store has uncovered in the design of distributed systems. What lessons can be learnt from the mistakes as well as the successes?

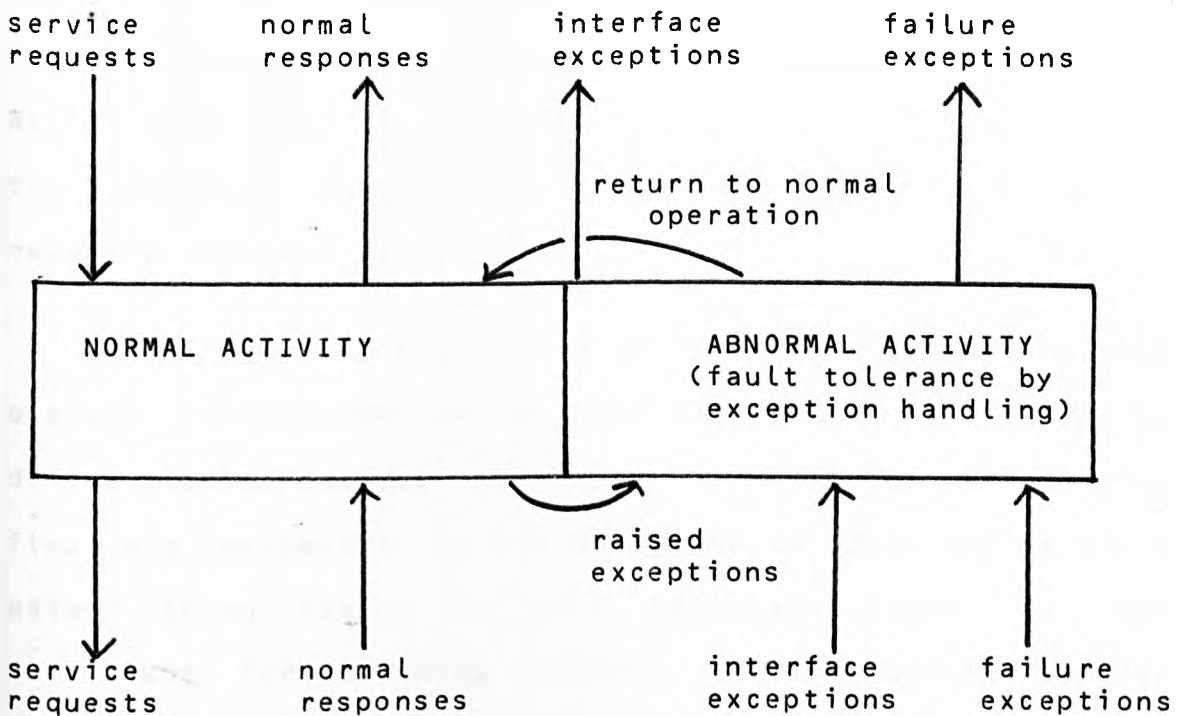
The design technique used by the author for the development of individual processes was step-wise refinement ([DIJK2] and [WIRTH4]). The partitioning of functions into processes was done with no methodology for guidance. A technique for recognising the natural concurrency in a system would be most useful. Traditional programming methodologies tend to hide natural concurrency. Developments such as communicating sequential processes [HOAR] provide ways of constructing concurrent programs with the aim of rigorous proof, but in themselves do not help the problem solver in the early stages. The author is aware of, though not yet fully conversant with, system design techniques such as [JACK] which do attempt to model concurrent systems. Designing an operating system with such techniques would be an interesting exercise.

It must be said, however, that a product was not being developed; the KUDOS project, as developed by the author, was an experiment which demonstrated and refined a set of algorithms which solved specific problems.

If KUDOS and its file-store were to be rewritten, greater consideration of its overall structure would have to be taken. In particular, the model of an ideal fault

tolerant component described in [ANDE2] would have significant influence. The model is illustrated by the following diagram:





Masking of errors is achieved by raised exceptions (detected errors) invoking redundant code which takes corrective action and returns to normal processing. Detected errors which cannot be recovered within the component are indicated in the outputs as interface exceptions (illegal requests for service) or failure exceptions (errors due to the component), and must be dealt with outside the component. The component interfaces to other components and may have to deal with their errors.

This model is essentially simple (as most generalities are, once realised). Without it, ad hoc application of reliability techniques can cause structural problems. This was clearly evident in the KUDOS file-store. The file-server component in particular was deficient in this respect, especially regarding lack of atomic update to disc.

More careful consideration of directory managers would have solved some of the recovery problems described later. Better consideration of exception reporting and handling by the operating system as a whole would have aided client recovery substantially.

KUDOS provided high reliability in certain aspects, but overall had a number of deficiencies such as the failure to define atomic actions which, by controlling information flow, are fundamental to the design of reliable software. A major lesson learnt is that technique alone is not sufficient for designing systems. KUDOS algorithms applied techniques such as redundancy very successfully, and derived much from other algorithms and techniques; a better structure would have displayed them more suitably.

Another aspect of the structure of KUDOS is its distributedness. A feature worthy of note is that the distributedness is not overly reliant on a particular physical configuration. The location of software components such as the directory-server or a file-server is relatively unrestricted. The name-server (resource directories) is the mechanism that permits this, allowing binding of names into software rather than physical addresses; reconfiguration is therefore much simpler.

An interesting question is whether the design and programming of systems for concurrent computer architectures, including local area networks, is

fundamentally different from the design and programming of the same systems for single processor computer architectures. The problems of process definition, synchronisation and communication are essentially similar. If communications and object location are adequate and well-defined, then the location of processes hardly matters in terms of the functionality of the system. The tools for implementation differ, as do the characteristics of the hardware, especially with respect to performance.

KUDOS exploits the "distributedness" of a local area network in terms of its scope for dynamic redundancy. By having a number of similar hardware components it is possible to reconfigure the software system in response to hardware failure. However, many of the algorithms would be appropriate in more traditional single processor architectures; in particular, the overlay mount scheme is potentially of use in any system where multiple file redundancy is required.

On the subject of technique, some comments can be made. An interesting observation is that whilst the KUDOS implementation used a no-wait send, the usage was almost always as a remote invocation send (remote procedure call), since a service request was usually followed by a wait for a reply. Ignoring the implementation argument, an original motivation for the no-wait send was that a process requesting a service could be doing something else before processing the reply. This is an efficiency argument, and a

particularly weak one. In practice there is usually little else that a process can do whilst waiting for a service to complete. Extra complexity is introduced if the requesting process does not wait for the reply; it must either poll the reply port periodically or respond to an interrupt.

There is also the question of synchronisation. The points at which processes wish to exchange information are typically points at which they require to synchronise. Moreover, if they do not synchronise at points of information exchange, problems of recovery can arise. For example, a no-wait send to a non-existent or failed process would result in a reply never returning. A requesting process would then have to implement time-outs, which is an added complexity. Its recovery may also be more difficult if it had continued processing after the send of the request.

Perhaps an analogy can be drawn between the GOTO in sequential programming and a no-wait send in concurrent programming. Both relate most directly to typical machine-level implementations, and form the underlying mechanisms for building higher level constructs. Both are tempting to use for rather weak efficiency arguments. Both lead to problems in terms of complexity of code.

The use of time-outs as an error detection mechanism in KUDOS has raised serious doubts. KUDOS used these extensively because of the way communication was

implemented. A time-out returns no information about a failure, and does not always indicate that a failure has occurred. Choosing sensible time-out limits also proves difficult. A too short time-out causes more trouble than it is worth; a too long time-out can mean that response to a failure is unbearably sluggish, especially if it causes an interactive user to wait. Essentially a send should return an error condition if it fails to deliver the message, even if that is across a network. Thus a synchronisation send, at least, is necessary.

To sum up the lessons in technique, the author has learnt that concurrent systems are difficult to program. The choice and disciplined use of appropriate tools is important. However, given that the tools are adequate, especially with respect to communications, the fact that a concurrent system is implemented on a number of machines rather than time-sliced on a single processor should make little difference.

On structure, more fundamental questions need answering. To successfully develop a significant software system embodying concurrency, a more rigorous approach to software design is necessary than was applied to KUDOS. In defence, KUDOS was developed to exemplify certain algorithms for a file-store. In practice an operating system and file-store should be designed with the choice of algorithms dictated more by requirements and the derived structure.

8.1.2 Reliability

As stated in Chapter 2, the reliability of a system is a measure of the success with which it meets its specification. In the design of the KUDOS filestore, three aspects of the specification were selected, namely that files should not be lost, that files should be accessible, and that updates to files should complete successfully or have no effect at all.

The main faults tackled by KUDOS are data destruction, through a disc head crash for example, or data unavailability, due to any number of reasons such as communications failure or a failed storage device. Failure during processing is handled, though not in an altogether satisfactory manner.

Fault tolerance was the major strategy, though no attempt to provide non-stop processing was made. The major fault tolerant technique was protective redundancy, by replicating files and access paths to files. This reduced the likelihood of loss of a file in a manner to be elaborated more fully later, and increased the availability of files.

Under a multiple-copy policy, an erroneous state exists when two copies of the same file are different in content (there is an interesting debating point here regarding whether a duplicated file is logically a single object or not). These errors are detected through timestamps (or

alternatively generation numbers) stored with files and compared before access to any copy of a file is allowed. The recovery from such an erroneous state is to overwrite the less recent copies of a file with the most recent.

This policy still does not ensure that a file is kept up to date, but remarkably increases the likelihood (see later). Stronger conditions can be imposed which make the likelihood of a triplicated (or more) file being out of date insignificant.

Ensuring that updates succeed or fail without effect is through a policy of careful replacement. Thus, if a process updating a file fails before reinstating it in the directory system, the effect is to leave a previous correct version of the file available.

Component independence, too, has played a significant part. In particular, storing access paths to all files on a volume on that volume increases accessibility of files.

Aspects of reliability theory not used fully, and which would have been useful, are multi-level recovery policies and atomic actions. The lack of appreciation of the value of the latter caused a number of problems which will be discussed later.

We shall continue this section with a consideration of some simple combinatorial mathematics which emphasise the effects of data replication in terms of access and

likelihood of loss. Some discussion will then take place regarding the way KUDOS tackled these aspects, and finally a look at other reliability considerations in KUDOS.

8.1.2.1 Elementary Reliability Calculations -

There seems to be a distinct lack of quantifiable reliability information used in system design. There are perhaps many reasons for this. A piece of equipment such as a disc might have a short production span before it is superseded by a superior design. A piece of computer equipment is complex, and the task of calculating its reliability from the known reliabilities of its components is too gross a task, at least with the design tools currently to hand. To test a significant number of items for a sufficiently long period would be too expensive.

Thus we are left with components whose probability of failure is unknown. Any reliability calculations must therefore, on the whole, be relative. Rather than ask how reliable a system is, perhaps we can say how reliable it is relative to its constituent components. Making worst case assumptions about the reliability of components we can then derive worst case information about the system.

A complex component, such as a disc, provides us with even more problems, in that it can fail for any number of reasons. A power-supply failure might remove the disc from the system for a while, but not damage stored data. A head

crash could effectively destroy all the contained data, and at best only a partial recovery of data can be made. The length of time a disc is faulty can also effect the system and is dependent on external matters such as the proximity of a service engineer.

We shall firstly make some elementary calculations based on the naive assumption that at any given time a component functions correctly with known probability. We shall then discuss more suitable probability distributions which may allow more representative modelling of a distributed computing system.

Suppose that a disc is online with probability P . Then, given independence of discs, if there are n copies of that file then under KUDOS algorithms the file is online with probability

$$1 - (1-P)^n$$

Tabulating this for various values of P and n we get:

P\ n	1	2	3	4
.1	.1	.19	.271	.344
.2	.2	.36	.488	.590
.3	.3	.51	.657	.760
.4	.4	.64	.784	.870
.5	.5	.75	.875	.938
.6	.6	.84	.936	.974
.7	.7	.91	.973	.992
.8	.8	.96	.992	.998
.9	.9	.99	.999	.9999
.95	.95	.9975	.999875	
.98	.98	.9996	.999992	
.99	.99	.9999	.999999	
.999	.999	.999999		

P\ n	5	6	7
.1	.401	.469	.522
.2	.672	.738	.790
.3	.832	.882	.918
.4	.922	.953	.972
.5	.969	.984	.992
.6	.990	.996	.998
.7	.998	.99927	.99978
.8	.99968	.99994	.999987
.9	.99999	.999999	.9999999
.95			
.98			
.99			
.999			

The figures not entered in the bottom right of the table all exceed .999999 .

Thus if a disc has a down time of 1 hour in 100 ($P=.99$), then a file with two copies will be unavailable 1 hour in 10,000 (over 1 year), and with 3 copies 1 hour in 1,000,000 (114 years). If a disc loses all its data once in 1,000 hours (42 days), and the disc is repaired immediately and returned to the system and brought up to date, then a file copied on two discs will be lost once in 1,000,000 hours (114 years), and copied on 3 discs will be lost once in 1,000,000,000 hours (114,077 years). These figures of course ignore failures in other parts of the system.

Earlier we considered the possibility of insisting that updates to replicated files could only proceed if a majority of associated volumes are online. This removes the possibility of old files reappearing unless individual discs are rolled back. We can use the binomial theorem [FREU] to derive the probability that a majority of associated volumes are online for a file replicated n times. The binomial theorem gives us the probability of m discs being online as

$$\frac{n!}{m!(n-m)!} P^m (1-P)^{n-m}$$

The probability that a majority (over half) of the volumes is online is

$$\text{SUM}_{m=M..n} \left[\frac{n!}{m!(n-m)!} P^m (1-P)^{n-m} \right]$$

where M is the least integer greater than $n/2$.

For various values of P and n we tabulate as follows:

P\n	1	2	3	4
.1	.1	.01	.0280	.0037
.2	.2	.04	.104	.0272
.3	.3	.09	.216	.0837
.4	.4	.16	.352	.1792
.5	.5	.25	.5	.3125
.6	.6	.36	.648	.4752
.7	.7	.49	.784	.6517
.8	.8	.64	.896	.8192
.9	.9	.81	.9720	.9477
.95	.95	.9025	.9928	.9869
.99	.99	.9801	.999702	.999408
.999	.999	.998001	.999997	.999994

P\n	5	6	7
.1	.0085	.0013	.0028
.2	.0579	.017	.037
.3	.1631	.0704	.1278
.4	.3174	.1792	.2897
.5	.5	.3422	.5
.6	.6826	.5452	.7102
.7	.837	.7442	.8741
.8	.9421	.9011	.9667
.9	.9914	.9841	.9973
.95	.9988	.9977	.9998
.99	.99999	.999804	.9999996
.999	.9999999		

Under the majority online constraint, availability is appreciably reduced. If a disc is offline 1 hour in 100 ($P=.99$) then a three copy file will be offline 1 hour in 3,333 (20 weeks) compared with 1 hour in 1,000,000. If a disc is irretrievably lost once in 1000 hours, then a file replicated 3 times will be lost once in 333,333 hours (38 years), but weakening of this constraint under a disaster would allow recovery with much greater probability.

It is important to note how the number of copies affects the majority online constraint. It is better to have an odd number of copies. There is a significant availability drop by increasing an odd number of copies by one in all cases. Intuitively this can be explained, since increasing an odd number of copies by one (say from 5 to 6) increases the majority (from 3 to 4) but does not increase the maximum number which can be offline at any one time (2). Intuition fails, however, for low values of P ; for $P=0.7$ we find it better to have 3 copies than 6.

A more realistic approach will include a reliability factor for the communication system, say C is the probability that the communication system is working, and a factor for a node, say N is the probability that a given node is working (assuming all nodes have the same factor). Then, if a personal work-station is working the availability of any copy of a file will be PNC . If a file has two copies kept on separate nodes then the probability it is available is

$$(1) (2PN^2 - (PN)^2)C = 2PNC - P^2NC$$

but if the copies are kept on the same node, though on different discs on that node, then the probability a copy is available is

$$(2) (2P - P^2)NC = 2PNC - P^2NC$$

and since $P^2NC < P^2NC$ when $N < 1$ we see that (1) < (2).

That is, as one would expect, it is better to keep copies of files on separate nodes as well as separate discs. If N is reasonably close to 1 however, the difference may be insignificant.

The simple assumption that a component functions correctly with a fixed probability P is naive. The probability a component functions correctly can depend on a number of factors, particularly time. Some components become increasingly likely to fail as time progresses, perhaps through wear and tear. A disc is likely to produce such properties, and a major function of maintenance is to prevent such failure. Other components become less likely to fail as time progresses; this could be the case for many electronic components which may fail in the first few hours because of slight defects in manufacture, but successful operation for a certain period implies continued success. Other components might have a high incidence of early failure, typically give trouble free operation for a long period, then begin to develop faults after a certain period.

A number of probability distributions can be used to model such behaviour. However, a useful approach to reliability calculation is to use the mean time to failure of components to deduce the mean time to failure of a system. [SH00] gives a thorough discussion of this, much too detailed to discuss here.

8.1.2.2 Access -

KUDOS uses multiple copy redundancy of files and directories, not only to reduce the likelihood that a file is lost, but also to ensure that the access paths to a file are available if a copy of the file is available. The mechanism is the overlay mount scheme. This is an important application of component independence, the component in question being a volume. By placing an access path to each file on a volume on that volume access is increased. KUDOS achieves this with a small overhead in disc space usage.

8.1.2.3 Prevention Of Loss -

In KUDOS any file can be replicated to whatever level of redundancy is required, up to a maximum equal to the number of volumes in the system. As related earlier, the likelihood of loss can be made arbitrarily small.

An important feature of KUDOS is the correction of out of date files through directory resolving by directory managers. By imposing constraints, such as the majority of

associated volumes must be online for activation to be complete, the likelihood that an old copy of a file is imposed on a user is substantially minimised. This is a principle form of error detection and recovery in KUDOS. A discrepancy between copies of a directory is an error which is resolved before the system continues processing.

8.1.2.4 Other Aspects Of Reliability Theory -

KUDOS did not begin with any aims to explore the general use of reliability techniques, but inevitably applied them where possible. KUDOS does not present a reliable system in terms of non-stop processing, and its granularity for recovery is somewhat coarse; this does not imply that KUDOS is inherently incapable of non-stop processing, nor of having a finer granularity for recovery.

Atomic actions (or transactions), and atomicity are important considerations. The critical file operations in the KUDOS directory system are atomic, in the sense that they succeed or have no effect. This is, however, based on the naive assumption that a single write to disc is atomic. Write to disc can be made atomic by a number of means, such as stable storage. Some directory operations are not atomic because of certain implementation difficulties, but this only causes minor irritation such as a directory reappearing when it has been deleted. A set of stronger conditions, such as a directory only being deleted when all associated volumes are online, or by leaving assassins and insisting

that a majority of associated volumes are online, would provide atomicity.

A major deficiency of the current KUDOS implementation is the lack of atomic write to a file in the file-server. Moreover, a file-server which supported transaction processing would be a great boon to KUDOS, especially if a two-phase commit protocol were available. It would simplify many of the directory commands and allow more powerful ones, perhaps permitting extension to database.

Atomic actions are a vital feature for aiding error recovery in a client. A client may wish to compose a whole sequence of operations into an atomic action. This cannot presently be done in the KUDOS directory system - single file operations are atomic, but cannot be rolled back on completion. Extension of KUDOS to allow recoverable atomic actions must be seen as an important step. Recoverable atomic transactions would be significant, bringing the file-server more into line with the Xerox DFS.

Under the current scheme this might be achieved by writing back a directory only when such a sequence of concurrent operations has completed. This is complicated by the possibility of concurrent client access. How to implement recoverable atomic actions encompassing more than one directory is not clear, at least with the current structures.

However, few file-stores offer atomicity even at single file operation. For example, GEC OS4000 can lose a complete file if an edit crashes. Unix does not offer atomicity of file operations. Systems with generation schemes do offer roll-back since a previous generation is available if creation of a new generation fails; this is not atomic however if an incomplete new generation is left after a crash.

An important concept in reliability is the notion of idempotence. An idempotent command is one which can be repeated without side-effects. This feature of a command allows a client simply to retry without roll-back if the result of a command is indeterminate. The result of a command may be unknown for a number of reasons such as a communications failure. For example, sequential write is not idempotent, but random access write is. A sequential write implicitly moves a pointer within a file-server which would be moved twice on a repetition. With random access write no such pointer exists.

Some commands in KUDOS cannot be made idempotent. For example, repeating a write lock will reject second and subsequent locks if the first lock had succeeded. Under the capability scheme it would be difficult to make locking idempotent.

The other directory command which is not idempotent is creating a directory. However, retrying this command will inform the client that the directory already exists, and so a client can determine whether the previous command succeeded with a simple retry.

In the file-server, file create is not idempotent in the sense that retry will leave a file in the file-server which no one knows about. This is not a client's concern and so file create does appear idempotent.

A client should make use of idempotence as a means of error recovery. For example, the directory-server will retry a file write up to five times (it is very pessimistic about communication protocols).

A major underlying aim of KUDOS was to avoid static servers. Static servers mean that the whole system is dependent on single nodes, giving serial dependencies rather than independent parallel options. Much work could be done on process relocation within the KUDOS multi-access shared system.

Currently the directory-server is implemented as a set of processes resident at a single node. There is, however, no reason why it should not be distributed across a number of nodes. It may, for performance reasons alone, be sensible to locate a directory manager on a node where a copy of the directory is kept.

Presently a directory-server is located at all file-server nodes, but only one is operative. If the operative directory-server node fails then this is detected by the other dormant servers and one of them becomes operative and publishes the address of the new root directory manager in the resource directories. Only one address for a name is allowed in the resource directories, so there is no likelihood of two directory-servers operating in parallel.

It would be preferable to see a much more flexible scheme, allowing partial recovery of a server rather than complete roll back. If a directory-server fails under the present scheme then all current operations fail, and clients must recover for themselves. Running secondary managers on separate nodes would allow some operations to be recovered. A more elaborate communication scheme with virtual calls which could be redirected might help on this.

8.1.3 Performance

8.1.3.1 File-server -

The file-server performance, particularly on file read/write, is likely to be the most critical aspect of the KUDOS file-store. Some delay on other aspects could be tolerated, such as file create, file delete.

The current KUDOS file-server requires at most two disc accesses to obtain any block of any file. A cache of most recently used blocks means that repeated access to a file should only need one disc access to obtain a block, and a recently used block would not need a disc access at all. The file-server I/O is therefore in principle fast.

This is one reason for the dichotomy between file-server and directory server. A client can then access a fast, efficient file-server for I/O, and consolidate this with the rather slower directory system.

The one major failing of KUDOS is the communication mechanism, and this must be improved significantly over current performance. The best inter-node process-to-process transfer rate achieved to date is about 6 kilobytes/sec. Admittedly, this is on slow LSI/11 machines with no DMA, but a sensible file-server must be able to deliver data at least an order of magnitude faster.

8.1.3.2 Directory System -

The directory system is a prime target for optimisation. The current implementation is very slow, requiring seconds to activate a directory.

Firstly, directory managers time out after a short period because there is no room for unused processes in the Modula system. A better solution might be to force out the manager least recently used and with no current operations

on activation of a new directory.

A more powerful node with a swap device would reduce the frequency of activation and deactivation by increasing the number of potential concurrent managers. Such a node would also reduce the heavy swapping of buffers between main-memory and disc necessary on the current implementation.

Activation can also involve a lot of file copying. Perhaps some of the copying could be done concurrently with client access to the directory, thus reducing client wait time for activation. This wait time could be high if a whole path of directories has to be activated.

Insert also involves a high amount of copying, but other commands involve very few disc accesses at all. The number of disc accesses would be even less on a more powerful node where caching was more feasible.

Tabulated below are the file-server accesses and messages required by the directory system for each primitive. This assumes that a directory manager can keep a copy of the directory in virtual memory, and also holds locking information in virtual memory. Note in particular that the messages required are proportional to the level of replication of directories, and not to a higher order of the level of replication.

primitive	file-server accesses	messages (other than file-server accesses)
activate (single directory)	$2E * D$	2
activate (path)	$2E * D$ (for each directory in path)	P+1
assocvols	0	2
mkdir	2D	2
deldir	2D	2
insfile	$(B+2) * D$	2
find	0	2
rlock	0	2
wlock	0	2
reflock	0	2
endlock	0	2
delete	2D	2
list	0	E+2
changeread	D	2
changewrite	D	2

where D=no of copies of a directory
 (ie no of associated volumes)
 E=no of entries in a directory
 B=no of blocks in a file
 P=no of names in a path

It is clear that directory activate and file insertion are the most expensive primitives. Inserting a file can be speeded up by increasing the block size and reducing B for a given file.

These figures would change with a different file-server implementation with different primitives. Moreover, they are worst case figures. For example, on activation, rather than read and write each record in a directory separately, buffering could be used to take a block of records at one go, reducing the file-server access by a factor equal to the number of records in a block.

8.2 NEW IDEAS

There are two ways in which KUDOS contributes to computer science research. Firstly, it includes novel ideas, particularly the overlay mount structure realised through associated volumes and the active file-store, and also a distributed name-server. The second form of contribution is a use of existing ideas in a new way, often as variations on a theme. We shall discuss KUDOS with respect to both these aspects.

Firstly, there is the overlay mount structure of the KUDOS file-store. Since Unix has become a paradigm as an operating system and file-store, we shall compare the KUDOS file-store with Unix. In KUDOS each volume contains a full path to all files contained by it on behalf of the directory

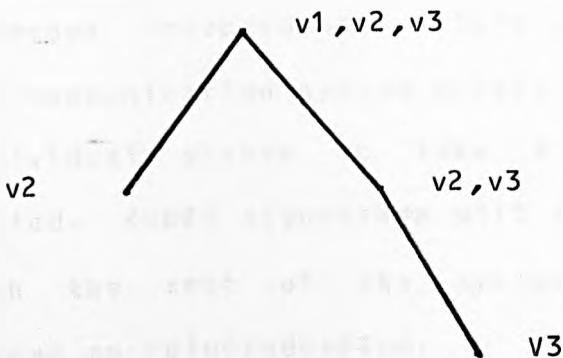
system. Thus a client is only dependent on the volume on which a file is stored in order to be able to access that file. In Unix a volume is mounted as a file system which is a subtree of the root file system. All volumes depend on the continued function of the volume containing the root file system.

We refer to KUDOS as having an overlaid mount structure, because the higher echelons of the hierarchy are replicated on a number of volumes, and to Unix as having a subtree mount structure. Note that the overlay structure of KUDOS means more than simply that a volume contains the names of all its contents. The directories in KUDOS contain much more than just a name and provide cross-reference to other volumes.

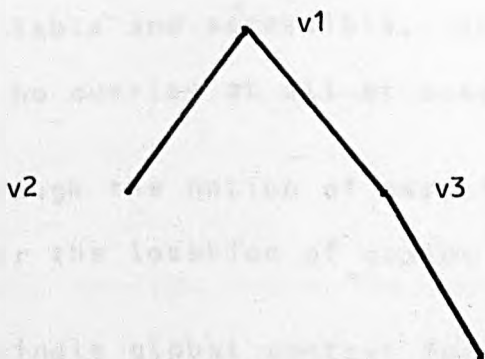
Diagrammatically, in a three volume system we might see a structure such as the following figure. In the subtree mount scheme v2 and v3 are dependent on v1. In the overlay scheme, no such dependency exists.

OVERLAY AND SUBTREE MOUNT

Overlay:



Subtree:



Overlay provides the following features:

1. Independence of volumes, increasing the reliability and access of the files. It is much more attractive in a distributed scheme where volumes are physically separated by a communications system.
2. A volume can be removed from the system and accessed independently. This might be the case if the communication system breaks down, or say an individual wishes to take a computer home for a period. KUDOS algorithms will restore consistency with the rest of the system for such a removed volume on reintroduction.
3. A clear tailoring of the system to provide different reliability for different parts of the system. The root is fully overlaid, and so is most reliable and accessible. Overlaying can be reduced to no overlay at all at some of the leaves.
4. Through the notion of associated volumes, control over the location of copies of a file.
5. A single global context for naming all files in the system. Thus a client can uniquely reference a file from any part of the system. In some distributed systems the access path to a file depends on the location of that file. On Unix, by

mounting a volume at a different point the path-name of a file can change.

Some problems with overlay have not been resolved in KUDOS. The main one is how to mount a volume and resolve it with the active file-store without damaging current operations. It might be possible to prompt a directory manager to resolve during activation as well as at the start of activation. Presently the lazy-man option has been taken of waiting for a directory to activate before introducing a new volume.

A large number of file-stores do provide independence of volumes. However, such systems usually embed in the filename the volume containing the file, which is an unfortunate restriction, and do not handle multiple copies of the same file.

As a total system, KUDOS is a development in a very active field of research. It overlaps and draws benefit from a number of systems, such as the Cambridge Model Distributed System, the Xerox distributed file-server, and LOCUS. We shall now review the KUDOS file-store in the light of other systems and in its own right.

The major aim in the development of KUDOS is reliability. However, reliability is a vague term and must be refined before any rigorous statements can be made. KUDOS concentrates on a limited aspect of providing a reliable system. This aspect was tackled in a very

different way in LOCUS.

KUDOS chose the principle unit of transaction to be a whole file. The Cambridge Model and Xerox systems chose blocks within files as the principle unit of transaction. KUDOS also does not permit multiple concurrent update to a single file. KUDOS needs revision in these areas, especially if database is to be supported.

KUDOS is unusual in that the major mechanism for enforcing consistency and providing high reliability is the naming mechanism, that is the directory scheme. The Cambridge Model and Xerox file-stores do not impose naming schemes, and thus cannot enforce constraints at that level. Perhaps the underlying difference in motivation is that KUDOS aimed implicitly at providing a user-oriented file-store rather than just a file-server which needs another layer to provide a user-oriented naming scheme.

The major aspects of reliability theory used in the KUDOS file-store are redundancy, component independence and careful replacement. At the operating system level, reconfiguration strategies are used on component failure. However, the granularity of transactions, based on the unit of a file, implies that a component failure can affect a large number of concurrent transactions, requiring them to roll back a considerable way. Thus KUDOS cannot be regarded as a non-stop system; rather it is one which fails safe and which can recover quickly.

In KUDOS a firm distinction was made between file-store and file-server. Effort was concentrated on developing the file-store, largely at the expense of the file-server. KUDOS provides, at present, a rather poor file-server, with a minimal set of properties required to support the directory scheme. The definition of file, too, was restricted, ignoring structured files such as indexed sequential. This was a way of simplifying the problem. Likewise, the ignoring of database was a simplification.

KUDOS imposed a protection scheme, not based on any global concept of ownership within the system, but by capabilities. This approach is common in distributed systems, such as the Xerox file-store. Protection was implemented at file level and at directory level. Access is prevented on an individual file basis, but there may be arguments in certain applications for limiting access to whole directories or subtrees.

The mutual exclusion in KUDOS is at file level. The Xerox file-store provides mutual exclusion at block level. Both adopt a locking policy with a similar timeout mechanism to avoid deadlock. To provide a finer grain of locking, KUDOS would have to revise its policy of careful replacement; update in place might still be avoided, but a suitable commit/roll-back policy in the file-server should achieve the same effect as careful replacement at a lower cost than complete data replacement.

Consistency, in terms of two copies of a file being identical, is enforced through the active file-store structure. Careful replacement prevents incomplete operations on files leaving a file partially updated. Multiple copies of files are used to reduce likelihood of loss.

To sum up, KUDOS provides a complete file-store with a full, single, global, hierarchical naming scheme, with multiple-copy redundancy, deadlock avoidance, controlled data-placement, automatic reconfiguration, limited checkpointing and recovery and file protection. In so doing it has introduced a number of new ideas, particularly overlay mount, associated volumes, active file-store and a distributed name-server. A number of features and ideas could be developed and explored more fully. These are elaborated in the next section.

8.3 FUTURE RESEARCH

KUDOS has certainly provided food for thought. It has concentrated on two aspects of file-store reliability and paid a limited respect to the general techniques of reliability theory. There are two ways future research might develop from KUDOS. Firstly, the algorithms could be explored and developed further. Secondly, on a more personal note, the insight gained through the development of KUDOS might be used by the author to explore related topics in reliability theory and data storage.

As stated, the current implementation of KUDOS can only be considered an experimental model. It has, however, proved the feasibility of KUDOS algorithms, and would justify a more ambitious implementation. One approach would be to adapt Unix time-sharing systems as nodes. This would involve a substantial reorganisation of the file-store, but it would retain much of the Unix style at system call level. The i-list structure is more sensible for a general purpose file-server than the current KUDOS scheme. Moreover, a wide variety of software already exists to ease the burden of development.

Such nodes, under a KUDOS regime, would not provide user processing, and would therefore have ample capacity for handling directory-server mechanisms. A single node would be capable of running a directory server for a substantial distributed system. The economic feasibility of this is now high. A small Unix system can be purchased for approximately ten thousand pounds, and there should be a continuing diminution of costs. A network of such nodes, both as members of a multi-access shared system and as personal autonomous work-station would be very attractive.

Apart from optimisation of some of the KUDOS algorithms, there are some areas which could be improved. Firstly, there is the mounting of a volume and its interaction with the active file-store. On the one hand it is desirable to bring the volume into the active file-store as soon as possible. On the other hand, one must avoid

upsetting current operations. This might be solved by devising methods of resolving a directory both during as well as at the beginning of activation.

Another problem is what to do if a volume in the active file-store fails. Currently this will back out the whole directory-server and abort all current operations. This is not a neat solution and could be refined.

The problem of how to handle a directory-server failure needs more examination. Perhaps failure can be prevented by duplicating directory managers. Alternatively, it could be minimised by distributing directory managers, or a separate journal of incomplete operations might be used to allow reactivation of the directory server elsewhere. Better structuring of KUDOS is certainly desirable here.

The area of communications needs closer examination. At least a remote invocation send (remote procedure call) should be provided. Other features such as automatic rerouting of virtual calls on a node failure would be very useful.

An area of growing interest to the author is database. It was an expressed aim of the Keele file-store project not to discuss databases. This was possibly a serious omission, and closed a number of interesting avenues of investigation. Certainly, at the file-server level a file-store ought to be able to support a database. Database is becoming more important to a variety of applications. Any future computer

system which cannot support database will be at a serious disadvantage.

It is debatable whether the KUDOS algorithms could be sensibly developed to handle database. Certainly the current granularity of locking and the policy of careful replacement would need substantial revision. In particular, the file-server would need improvement. The file-server would need to provide recoverable transaction processing, probably with a two-phase commit protocol. More functionality would be desirable, say by adding the capacity to handle indexed sequential files.

The mathematical treatment of the work presented could be developed significantly. In all there seems to be very little mathematical justification behind much system design. However, techniques of probability theory exist and need only be applied to applications such as the one described here. Related earlier was the difficulty of providing thorough mathematical analysis with current design tools. However, the use of computer-aided design of computer systems should facilitate the automatic calculation of system reliability from the known reliability of components.

CHAPTER 9

SUMMARY AND CONCLUSIONS

In this final chapter we shall summarise the thesis and discuss the achievements culminating in this thesis.

9.1 SUMMARY

The thesis began by citing the problem of reliable file storage, and restricted attention to the problem in the context of local area network computer systems. The topic of reliability theory was discussed and relevant aspects highlighted for inclusion in a solution to the stated problem. In particular the notions of redundancy, component independence, and atomicity were considered most relevant to the solution proposed.

Local area network computer systems were considered on the one hand as a highly topical computer system architecture, and a subject area receiving a great deal of attention for a number of reasons. Local area networks offer a number of attractive features, notably ease of expansion, high performance, applicability, cost and reliability. On the other hand local area networks were

considered as a particularly relevant architecture for providing reliable file storage.

Filestores were discussed with reference to a number of existing systems. A number of issues were raised, notably problems of data placement, consistency, shared access and naming. A definition of file-store was made which included a file-server as a component part.

A distributed operating system called KUDOS (Keele University Distributed Operating System) was described. This was developed by the author as a vehicle for implementing a prototype file-store. To a large extent it became a project in its own right, including some novel ideas, notably on dynamic resource location. Some interesting experiences are related, especially on communications and on the approach taken.

The KUDOS file-store was then described. This constructs a reliable file-store out of a number of independent file-servers. This file-store provides a single global hierarchical naming scheme, controlled multiple-copy redundancy, a neat solution to data placement, deadlock avoidance, automatic reconfiguration, limited checkpointing and recovery, and file protection.

Finally, the file-store is discussed in terms of its design, its contribution, and possible future research.

9.2 ACHIEVEMENTS

The achievements can be divided broadly into three categories, namely personal, local and public. The personal achievements are what the author himself derived from the exercise. The local achievements are seen as those directly relevant to the department and the research team within which the work took place. Finally, the global achievements are the actual, tangible contributions to the subject.

An important achievement is the recognition of what went wrong, and hopefully why. Some self-criticism is important here, though the psychological trick of regression tempts one naturally to ignore this exercise.

9.2.1 Personal

(The reader should forgive the use of the first person in this section, but it seems most natural)

One of the original motivations for embarking on a Ph.D. was to broaden my technical appreciation of the subject. Having spent two years as an applications programmer/analyst my appetite for the subject was wetted, far more than in the minimal exposure to computer science as an undergraduate following a Pure Mathematics course. However, working in a commercial environment tends to tunnel one's vision, giving a detailed understanding of a limited number of aspects of the subject.

A Ph.D. seemed an ideal way of stepping off the treadmill and looking around. The first year was largely a process of catching up, involving much broader reading than is represented in this thesis. Teaching undergraduates was a particularly useful exercise; an excellent way of understanding something is having to explain it to someone else.

Secondly, I wanted to see something of the academic world, other than the rather limited view of an undergraduate. To many outsiders the academic world is a mystical brotherhood, reserved for an intellectual elite, and frequently divorced from reality. It is reassuring to find that this is not true.

I was rather fortunate in the contact I had with other academics. I attended a number of conferences, workshops and seminars, and also had the honour of presenting my own work at some of these. A great deal of this is due to the department and project in which I was working, and I am duly grateful.

Thirdly, there is the mountaineering aspect. I did a Ph.D. because it was there. Even if one does not make it there is the satisfaction of having tried.

These objectives were all achieved. There was never any serious wish to become a grand master overnight, and to make outstanding contributions to the subject; if one is lucky enough then so be it. A small contribution suffices

for most. In all, then, the personal aspects of the Ph.D. were successful.

9.2.2 Local

The research presented took place within a research project involving a number of people, and one should assess it in terms of that project. However, the work was self-contained and entirely the author's.

The overall project divided into a number of fairly independent projects, including the construction of a Cambridge Ring, design and construction of network interfaces, some simulation work on networks, communication protocols, robust processing (or process survivability) in a hostile environment, and another file-store project.

It is difficult to relate directly the development of KUDOS to the other projects. It was not developed in conjunction with anyone else, though the free flow of ideas was helpful both ways.

KUDOS, then, should be seen as a representative of the groups work, though not typical. KUDOS most nearly approached the title of the global project, though the other projects were by no means irrelevant.

It is open to speculation whether the group would have benefitted from a more constrained development. There might have been distinct benefits from having a single core

project on which everyone worked, with other projects spinning off. A serious limitation on the team's progress was lack of equipment. Whilst hardware does not solve everything, the lack of certain tools involved a great deal of time and effort in developing them or making do without.

9.2.3 Global

This section is perhaps the most difficult to write, though the credibility of the thesis relies upon it. What contribution is made to the subject?

On a tangible note, the work for the Ph.D. resulted in three externally published papers, [LUNN1],[LUNN3],[LUNN4]. The latter paper represents the early ideas for the KUDOS file-store before it was developed, and is of historical significance to this thesis. [LUNN1] was the result of a simulation study of the Cambridge Communications Ring, and was an early self-contained project. [LUNN3] describes the distributed name-server algorithm presented earlier in this thesis.

The KUDOS file-store has a number of features, as described in the introduction, namely:

1. Single global hierarchical naming scheme. The naming mechanism appears to the user as a single hierarchy similar to that provided by Unix and many other operating systems. There is no notion of location embedded in the name of a file, unlike

many such file-stores on distributed systems.

2. Controlled multiple copy redundancy. It is possible to replicate a file on all or any subset of the set of volumes in the system. If a copy falls behind because its storage volume is inaccessible, it is brought up to date before it is accessed by reference to other copies in the system. The algorithms presented can respond flexibly to requirements by minimising the likelihood that an out of date file is accessed or by removing that possibility under normal progress of the system (including the response to system failures). Non-critical files can be stored as one-copy, as are uncommitted files during update.
3. Neat solution to data placement. Through the hierarchy of directories, though not embedded in the name of a file, the placement of copies of a file is controlled. This is by a mechanism called "associated volumes".
4. High availability. Associated volumes also provide another benefit in that a file can be accessed if any volume containing a copy of that file is online. This is because each volume contains a copy of all paths to all files held by it.

5. Deadlock avoidance. It was decided to duck the issue of deadlock detection and to adopt a policy of deadlock avoidance. This is perhaps a better alternative in a system where no centralised control exists, but we do not argue the case. The mechanism chosen is based on timeout of locks, and relies to some extent on the reasonable behaviour of software using the file-store.
6. Automatic reconfiguration. If a node in the system fails, the file-store will reconfigure itself to provide a continued service. Any software dependent on a failed node for a particular transaction may, however, have to back out and retry in the new configuration.
7. Limited checkpointing and recovery. By adopting a policy of careful replacement, process failure should not leave a file in a partially complete state. It is felt that this area of KUDOS requires more scrutiny, especially with respect to atomic update in the file-server.
8. File protection. A system of keys, generated by the system and by the user prevent illegal access to data stored in the file-store, and implement a data privacy scheme for users.

The first four features are perhaps the most significant. The pertinent underlying mechanism for implementing these are associated directories and active file-store management.

The topic of local area networks is of particular interest to a number of research groups, and the experiences and ideas will no doubt have relevance to them. In particular the distributed name-server algorithm and the notion of overlay mount of volumes are novel ideas worthy of further exploration.

To sum up, KUDOS is an example of a computing system in a topical area whose recipe includes some new ingredients and a variation on the use of existing ingredients. Whilst it is not an earth-shattering success, it provided the author with a deeper understanding of the subject area, and introduced some ideas of note to others. And finally, despite the occasional and inevitable troughs of depression, it was enjoyable; what more can one ask.

CHAPTER 10

REFERENCES

[ADA] "Reference Manual for the Ada Programming Language".
United States Department of Defence, July 1980.

[ALMES] Almes, G.T., Lazowska, E.D. "The Behaviour of
Ethernet-like Computer Communications Networks". Proc. 7th
Symposium on Operating System Principles, pp 66-81.

[ANDE1] Anderson, T., Randell, B. "Computing Systems
Reliability". Cambridge University Press, 1979.

[ANDE2] Anderson, T., Lee, P.A. "Fault Tolerance Principles
And Practice". Prentice/Hall International 1981.

[APOLLO] Apollo Computer Inc., various sales and information
literature, 5 Executive Park Drive, N. Bellerica, MA 01862,
USA.

[BARN] Barnett, J.K.R. "A Highly Reliable File System which
Supports Multiprocessing". Software Practice & Experience,

Vol 8, pp 645-671, 1978.

[BENHAM] Benhamou, E. "Integrated Software Design for Z-Net, A Local Micro-computer Network". Proc 2nd International Conference on Distributed Computing Systems (IEEE Catalog 81CH1591-7) pp 397-403, April 1981.

[BENN] Bennett, K.H., Singleton, P. "The Design of a Microprocessor-based Access Logic Unit for the Cambridge Ring", Internal Report DCP/R7, Dept of Computer Science, Univ of Keele, England.

[BREN] Brenner, J.B. "A General Model for Integrity Control". ICL Technical Journal, 1978, 1, 71.

[BRER] Brereton, O.P. "Message Passing Performance On A Cambridge Ring" Software Practice and Experience (to appear).

[BRIN1] Brinch Hansen, P. "The Architecture of Concurrent Programs". Prentice Hall 1977.

[BRIN2] Brinch Hansen, P. "Operating System Principles". Prentice/Hall.

[BROWN] Brownbridge, D.R., Marshall, L.F., Randall, B.R. "The Newcastle Connection" Software Practice and Experience Vol 12, pp 1147-1162, December 1982.

[CASEY] Casey, L.M. "Computer Structures For Distributed Systems" Ph.D. Thesis, Univ of Edinburgh, December 1977.

[COTT] Cottam, I.D. "Functional Specification of the Modula Compiler", Internal Report 33, Dept of Computer Science, Univ of York, England.

[DIJK] Dijkstra, E. "Go To Statement Considered Harmful" Communications of the ACM, Vol 11, No 3, pp147-148, March 1968.

[DIJK] Dijkstra, E., Dahl, O.J., Hoare, C.A.R. "Structured Programming" Academic Press 1972.

[DION] Dion, J. "The Cambridge File Server", ACM Operating Systems Review, Vol 14,4, October 1980.

[ESWA] Eswaran, K.P., Gray, T.N., Lorie, R.A., Traiger, I.L. "The Notions of Consistency and Predicate Locks on a Database System". Communications of the ACM, November 1976, Vol 19, 11, pp 624-633.

[FREU] Freund, J.E. "Mathematical Statistics". Prentice/Hall International 1972.

[FRID] Fridrich, M., Older, W. "The Felix Fileserver", Proc. 8th Symposium on Operating System Principles, December 1981.

[GEC] Various systems documentation for OS4000 systems. GEC computers Ltd, England.

[GIFF1] Gifford, D.K. "Violet, an Experimental Decentralised System". Integrated Office System Workshop, IRIA, Rocquencourt, France. (Nov. 1979). Available as Report CSL-79-12 Xerox Corporation, Palo Alto, CA.

[GIFF2] Gifford, D.K. "Weighted Voting for Replicated Data". Proceedings of Seventh Symposium on Operating System Principles. December 1979, pp 150-159.

[HALS] Halsall, F. "Interprocess Communication In Multiple Computer Systems". Microswap, Vol 3, No 2, November 1979.

[HERB] Herbert, A. "The User Interface To The Cambridge Model Distributed System". Proc of 2nd International Conference On Distributed Computer Systems (IEEE Catalog 81CH1591-7), April 1981, pp 503-508.

[HOAR] Hoare, C.A.R. "Communicating Sequential Processes, Communications of the ACM vol 21,8 August 1978.

[HOLD] Holden, J.C., Wand, I.C. "An Assessment of Modula". Software Practices Experience. Vol 10, pp 593-621 (1980).

[ICL] Various systems documentation. International Computers Ltd., England.

[ISO] "Reference Model for Open Systems Interconnect". ISO/TC97/SC16 N227, British Standards Institution, 101 Pentonville Road, London N1 9ND.

[JACK] Jackson, M. "System Development". Prentice/Hall International 1983.

[JENS] Jensen, K., Wirth, N. "Pascal User Manual And Report". Springer-Verlag 1978.

[JOHNS] Johnson, M.A. "Ring Byte Stream Protocol Specification". Internal Report, Computing Laboratory, Univ of Cambridge, England.

[JONES] Jones, C.B. "Software Development a Rigorous Approach". Prentice/Hall International 1980.

[LAMP] Lampson, B.W. "An Operating System for a Single User Machine". Proceedings of Seventh Symposium on Operating System Principles, December 1979.

[LAMP0] Lamport, L. "Time, Clocks and the Ordering of Events in a Distributed System". Communications of the ACM, July 1978, Vol 21-7, pp 558-565.

[LANT] Lantz, K.A., Rashid, R.F. "Virtual Terminal Management in a Multiple Process Environment". 1979 ACM 0-89791-009-5/79/1200/0086.

[LINDS] Lindsay, B.G., Selinger, P.G. "Notes on Distributed Data Bases". Advanced Course on Distributed Data Bases. Sheffield City Polytechnic. 9th July, 1979.

[LISK] Liskov, B, "Primitives for Distributed Computing". Proceedings of the 7th Symposium on Operating System Principles, December 1979.

[LOGICA] various sales and information, Logica Ltd, England.

[LUDER] Luder, G.W.R. et al, "A distributed Unix System Based on a Virtual Circuit Switch". Proceedings of the 8th Symposium on Operating System Principles, December 1981.

[LUNN1] Lunn, K., Bennett, K.H. "Message Transport On The Cambridge Ring - A Simulation Study". Software Practice & Experience. Vol 11, No 7, July 1981. pp 711-716.

[LUNN2] Lunn, K. "System Design Objectives" Internal Report DCP/WD/23, Dept of Computer Science, Univ of Keele, Sept 1979.

[LUNN3] Lunn, K., Bennett, K.H. "An Algorithm For Resource Location In A Distributed Computer Network". ACM Operating Systems Review. April 1981.

[LUNN4] Lunn, K., Bennett, K.H. "A Highly Reliable Distributed Filestore Directory System", Proc. 2nd

International Conference on Distributed Computer Systems (IEEE Catalog 81CH1591-7), pp 299-307, April 1981.

[LYCK] Lycklama, H., Christensen, C. "A Minicomputer Satellite Processor System". The Bell System Technical Journal. July 1978. Vol 57, 6/2, pp 2103-2114.

[MICROE] "Pascal Microengine Reference Manual", The Microengine Company, 3182 Redhill Avenue, Newport Beach, CA 92663, USA.

[MITCH] Mitchell, J., Dion, J. "A Comparison of Two Network-based File Servers" Proc. 8th Symposium on Operating System Principles, Dec 1981, pp 45-46.

[MONTG] Montgomery, W.A. "Polyvalues: A Tool for Implementing Atomic Updates to Distributed Data". Proceedings of Seventh Symposium on Operating System Principles, December 1979. pp 143-149.

[NATA] Nataragan, N. "Atomic Actions and Timestamps". ACM Operating Systems Review, April 1980. Vol 14-2, pp 21-24.

[PEAK] Peake, P.J. "Another Distributed Filestore Proposal". University of Keele, Dept of Computer Science internal document. DCP/WD/26.

[PENN] Penney, B.K., Baghdadi, A.A. "Survey of Computer

Communication Loop Networks", Research Report 78/42, Dept of Computing and Control, Imperial College of Science and Technology, London, England.

[PERQ] various sales and information literature, ICL, England.

[RAND1] Randell, B., Lee, P.A., Treleaven, P.C. "Reliability Issues in Computing Systems Design". Computing Surveys, June 1978, Vol 10, No 2.

[RAND2] Randell, B. "System Structure for Software Fault Tolerance". IEEE Transactions on Software Engineering, Vol SE-1, No 2, June 1975.

[REDE] Redell, D.D. et al. "Pilot: An Operating System for a Personal Computer". Communications of the ACM, February 1980, Vol 23, No 2, pp 81-91.

[RITCH1] Ritchie, D.M., Thompson, K. "The Unix Time-sharing System". The Bell System Technical Journal, July 1978. Vol 57, 6/2, pp 1905-1930.

[RITCH2] Ritchie, D.M. "Unix Time-sharing System - A Retrospective", The Bell System Technical Journal, July 1978. Vol 57, 6/2, pp 1947-1970.

[RITCH3] Ritchie, D.M. and Thompson, K. "The Unix Time

Sharing System". Communications of the ACM, Vol 17, July 1974. pp 365-375.

[SALT] Salter, J.H. "Naming and Binding of Objects". Operating Systems - an Advanced Course. Edited by Bayer, R. et al. Springer-Verlag, 1979. pp 99-208.

[SELIG] Seligman, D.R. "On The Performance Evaluation Of DECNET" Proc. 2nd International Conference on Distributed Computer Systems (IEEE Catalog 81CH1591-7), pp 484-496, April 1981.

[SH00] Shooman, H.L. "Probabilistic Reliability: An Engineering Approach". McGraw Hill, 1968.

[SMITH] Smith, D.C., Irby, C., Kimball, R. "The Star User Interface: an Overview". Proc. 1982 National Computer Conference, AFIPS, June 1982, pp 515-528.

[STURG] Sturgis, H., Mitchell, J., Isreal, J. "Issues In The Design And Use Of A Distributed File System". ACM Operating Systems Review, Vol 14, No 3, July 1980, pp 55-69.

[TOML] Tomlinson, G.M., Keeffe, D., Wand, I.C., Wellings, A.J. "The Pulse Distributed File System" Internal Report, University of York, England, November 1982.

[VERH] Verhotstad, J.S.M. "Recovery Techniques for Data

Base Systems". Computing Surveys 10-2, 1978, pp 167-195.

[WILK1] Wilkes, M.V., Needham, R.M. "The Cambridge Model Distributed System". Operating Systems Review, Vol 14, No 1, pp 21, January 1980.

[WILK2] Wilkes, M.V., Needham, R.M. "The Cambridge Model Distributed System". Internal Report, Computer Laboratory, Univ of Cambridge, England, 1979.

[WIRTH1] Wirth, N. "Modula: A Language For Modular Multiprogramming". Software Practice & Experience, Vol 7, 1977.

[WIRTH2] Wirth, N. "The Use Of Modula". Software Practice & Experience, Vol 7, 1977.

[WIRTH3] Wirth, N. "Design And Implementation Of Modula". Software Practice & Experience, Vol 7, 1977.

[WIRTH4] Wirth, N. "Data Structures + Algorithms = Programs". Prentice/Hall 1976.