Technical Report

UCAM-CL-TR-169
ISSN 1476-2986

Number 169

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Filing in a heterogeneous network

Andrew Franklin Seaborne

April 1989

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted July 1987 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Heterogeneity is becoming a common feature in local area networks as the variety of equipment that is marketed increases. Each such system will have its own mix of hardware and software but it is still desirable to be able to bring in new applications and machines and integrate them with the existing equipment.

Careful design is required to be able to introduce new subsystems into the network in a manner that reduces the manpower needed. If binary compatibility for application programs is achieved then new software can be introduced without the need for source code alterations. If the design of the computing environment is correctly constructed then the introduction of new hardware will not require alterations to or cause disruption of the rest of the system.

There is a reduction in the ability to share information through files being accessible to many people and from many locations when there are a number of filing systems present in the network. Ideally, a single filing system spanning all operating systems that exist in the distributed computing environment would give maximum possible sharing.

Any existing file service will provide a set of facilities for the construction of a name space by its client or enforce a predefined naming structure which is not compatible with any other. This thesis describes a single network filing system that has been constructed by separating file storage from file naming. By introducing a directory service to manage the name space, and using file servers only for file storage, the need for each client to be extended to take account of every file service is avoided. A single network file transfer protocol allows the directory service to authenticate each request and allows for the introduction of new equipment with no disruption to the existing system.

Preface

I have received much encouragement from my supervisor, Dr. Jean Bacon, and other members of the computer laboratory during my research. The discussions held with these people have been useful and have helped to clarify my thinking on many of the issues involved in my work.

I am grateful to the United Kingdom Science and Engineering Research Council for providing the studentship during which the work of this thesis was carried out.

Except where stated in the text, this dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

Contents

1 Introduction **1**

1.1 Objectives 1

1.2 Motivation 2

1.3 Thesis Structure 4

2 Distributed Computing **5**

2.1 Properties 6

2.2 Elements of Distributed Computing 8

2.3 Local Area Networks and Heterogeneity 12

2.4 Caching 13

3 Filing Systems **16**

3.1 Objects in a Filing System 16

3.2 Naming 17

3.3 Sharing 21

3.4 Concurrency Control 22

3.5 Access Control 25

3.6 File Servers 27

4 Related Work **30**

4.1 Design Influences 30

4.2 The Washington Common Store 31

4.3 The Cedar File System 33

4.4 LOCUS 34

4.5 The ROE Filing System 36

4.6 The TRIPOS Filing System 37

5 The Design of a Filing System **46**

5.1 Resource Utilization 46

5.2 Problems of Heterogeneity 47
5.3 Features of a Distributed Filing System 50
5.4 A Network Directory Service 55
5.5 Client Model of File Operations 56
5.6 Data Transfer 61
5.7 Summary of Design Features 62

6 A Prototype Directory Service **64**
6.1 Environment 64
6.2 Data Structures 68
6.3 Replication of Directory Service Information 70
6.4 Elections 76
6.5 Failure Tolerance 78
6.6 Concurrency and Locking 83
6.7 An Example 84

7 The Client Model **88**
7.1 Client Interface 88
7.2 Transfer of Files 94
7.3 Client Structure 99

8 Conclusions **102**
8.1 Work Carried out 102
8.2 Evaluation 103
8.3 Review 105

A Prototype Design **108**
A.1 Data Types 108
A.2 Pathname Lookup 112
A.3 Creating a New Directory 114
A.4 Distributed Locking 115

B Prototype Implementation **119**
B.1 Initialization 120
B.2 Performing an RPC 122

B.3 Requests and Operations 122
B.4 Testing 123

C Client Interface **125**

References **127**

1 Introduction

In any computing system there is a need for permanent storage of data which is often provided by use of optical or magnetic media. This thesis discusses the problems that arise with filing systems in a computing environment which is constructed from many types of computer and operating system connected by a local area network, and it proposes solutions that can be applied to a variety of computing models. A prototype implementation has been built to demonstrate the principle features of these solutions.

1.1 Objectives

The requirements for an effective filing system in a local area network which are addressed by this dissertation are :–

- *Sharing* : Providing a single filing system, rather than a number of filing systems connected by file transfer routes, enables easy sharing of information.

- *Heterogeneity* : The nodes of a local area network do not necessarily run the same operating system or run on the same hardware. Heterogeneity of hardware and software obstructs the objective of a single filing system.

- *Evolution* : A distributed computing environment can not be expected to remain static over a period of time. Bringing in new hardware and software can entail a great deal of time and manpower unless this problem is addressed by the design. Application software should be able to run unchanged in the new environment with no need to recompile or to relink.

- *Effective utilization* : An attractive property of local area networks is the ability to share hardware among the various users and applications in the

distributed computing system. At the same time it is important to allow a resource that is local to a particular machine, such as an attached disc drive, to be used to its full potential.

It is possible to meet these functional requirements without compromising efficiency if solutions that are specific to the problem are employed. It is argued that the use of general solutions, typically those embodied in recent standards, which can be applied to a class of problems, is undesirable because of the additional costs incurred.

1.2 Motivation

The filing system is central to a computer system. This makes it an important area that must be addressed if support for heterogeneity is to be achieved in local area networks.

- There are concepts, such as that of a file as a container of information and that of a directory as a unit of naming structure, that are common to many computer systems. It will not be possible to integrate systems based on widely differing models of permanent storage; it is possible to integrate those that share common concepts and so produce a single, heterogeneous filing system.

- It will be shown that distributed systems are naturally heterogeneous. The provision of permanent storage in such a distributed system must be designed to accommodate this variety of hardware and software.

- Sharing through a single filing system is important in existing, stand-alone systems. It is desirable to construct a filing system that carries over this concept into distributed environments.

A filing system gives a degree of integration to a computing system. This could be provided in various ways. One is to host a 'network operating system' on each machine, where each machine has a common interface implemented on top of

the standard kernel to give an impression of homogeneity. Another is to convert existing software systems so that they provide a common interface, at least to network services.

The former choice results in access to local files, those stored on the local disc of the machine, being made through the conventional kernel interface but access to remote files being made through additional operations provided on each machine in the network. It is the responsibility of an application or of a user to decide whether a file is local or remote and to choose the appropriate operation. Sharing can occur only if a file is explicitly placed in the network-wide filing system and hence must be planned.

Conversion of existing systems to provide a common access method for resources, whether they are local or remote, is attractive if the system is homogeneous. The extended system can be made network transparent with no distinction being made between local or remote access to any resource. However, two problems arise in a heterogeneous environment. First, the work needed to provide these facilities is large as each of the different types of system involved needs a large amount of manpower to be devoted to maintaining and extending it. Second, there is no assurance that there exists a single network model of resource access that allows expression of the full range of facilities offered by any operating system. Research in homogeneous systems has shown that it is possible to separate the filing system into one or more network wide resources and for the resulting system still to provide a usable facility.

The principle technique advocated is to separate the storage of files from operations on the name space. By providing a directory service that controls naming and access, a single model of file access can be provided, built on top of existing file storage services. The directory service removes the need for users and applications that manipulate files to understand the semantics of all possible file storage servers in the network. It enables the correct functionality to be implemented on top of file storage servers that do not provide the full range of facilities. Boundaries of trust are established so that operating systems in the network do not need to rely on each other to perform changes to the data structures of directories correctly.

1.3 Thesis Structure

Chapter 2 establishes the principles and features of local area computing which are refined in Chapter 3 in relation to the provision of permanent storage. Chapter 4 looks at other work, concentrating on aspects of other systems which have addressed the research areas of this dissertation.

Chapter 5 then gives the design features of the directory service and the prototype is described in Chapter 6. The problems of adapting a particular operating system to the proposed model of working are described in Chapter 7. Chapter 8 summarises the conclusions of the work.

2 Distributed Computing

The term *system* can be applied in a wide range of situations. A system is a collection of one or more sub-units, such as processes or programs, acting together to achieve some common goal. For entities outside the collection it is possible to consider the system as a single abstraction or as a number of parts of a whole. For example, to a user of an operating system the kernel is viewed as a system because there is a single interface presented by the collection of system calls. From the point of view of the operating system designer the kernel is a number of systems which interact with each other, such as network drivers or virtual memory management routines. To the scheduler these kernel processes may appear to have little connection; to the user the whole collection is a single unit.

What is viewed as a system is, therefore, dependent on the standpoint taken. If it is viewed as a single entity, masking the internal structure, then it may be described as a system. An example of a system in a distributed computing environment is those components that give users access to files. Indeed, from the point of view of a user, all those components that support work in the environment and participate in its management may be described as a system.

A system that provides access to some facility, such as authentication of the credentials of a third party, is termed a service. The service may be supplied by one or more servers, which are the actual machines on the network that conform to the interface of the service. A client is a user of a service. Clients are active in that they initiate actions on behalf of their human users. An application is a software system, usually a single program, built on the facilities provided by the distributed environment to achieve some task. This task is usually quite specific in nature. For example a file server provides one form of access to permanent storage; it makes use of the network and, maybe, the authentication service. From the point of view of an application such as a database management program, the

file service is one element of the distributed system that is employed to provide the user with the facilities of the database. The file service has no concept of the special data structures used by the database manager, but provides facilities out of which these may be constructed.

2.1 Properties

A distributed computing environment differs markedly from other computing models. There are various properties a distributed system can provide for an application which are not usually available in other environments. These derive from the fact that the application is not confined to run on a single processor; the degree of replication can be chosen by the programmer and is not fixed by the environment design. From this the programmer can, if the design allows, choose varying degrees of performance, availability and reliability to achieve the application's objectives.

- *Performance* : An increase in performance of some applications and services can be achieved by simply upgrading the quality of the components implementing the service. However, this approach results in loss of service time during the changeover, development work to incorporate new hardware or software and permanent commitment of resources to the upgrade. The alternative of replicating a resource that causes a bottleneck allows throughput to be increased by giving true concurrency without disruption to the existing system and, if demand drops, it is a simple matter to commit the resources to some other task.

  Replicating code brings true concurrency to a service because more processors are devoted to the task; partially replicating data means that each server deals with requests relating to that part of the data it controls.

- *Availability* : Replication to bring increased concurrency of access means that, when a request arrives, it is more likely that it can be serviced immediately, rather than having to wait for some earlier request to complete. This

reduces the latency of requests and improves the real-time response of the service.

- *Reliability* : Where a service is critical to the functioning of its clients the chances that it is unavailable due to component failure can be reduced to any small probability by replication of components. In this situation it is possible to continue to provide the service, albeit at a reduced level, if there is a failure of hardware or software.

There are further features of distributed computing systems that apply to the system as a whole which have increased the attractiveness of this form of computing to areas outside research. The properties of performance, availability and reliability can be provided in other ways, for example by employing specialized hardware. In a distributed computing environment based on a local area network an application can view the configuration in any way it chooses since the cost of communication between any pair of machines on the network is taken as being constant.

- *Flexibility* : The configuration of components in a local area network is not a function of the topology of the network. Since any node can communicate with any other, the logical configuration can be chosen by an application and the real configuration need not be changed for different applications. This can be exploited when hardware is added or removed from the system; if an application does not require some component then the removal of that component may be achieved without requiring that the application be stopped and restarted.

- *Cost Effectiveness* : Resources can be shared between clients rather than providing each client with all the resources it could possibly need but will not fully utilize at any given time. This spreads the cost over more than one client by sharing resources, providing the functionality of a resource without dedicating it to one client.

2.2 Elements of Distributed Computing

2.2.1 Clients and Servers

Many activities in a distributed computing system are modelled by a client requesting some action from a server. The client plays the active role in that it initiates the exchange while the server waits for requests to arrive. On arrival the server decodes the message to determine the function required and the parameters for that function according to some published interface definition; it might be that one or more of the parameters provides the server with justification that the message is valid, that is, the message received is from a machine that is authorized to perform the action according to any restrictions that the server might impose. Once the server has performed the action a reply message is sent to the client.

In its simplest form the dialogue will consist of one request and one reply with the client waiting for the reply, similar in manner to a procedure call. However, there is no reason why the protocol implemented by the server should produce exactly one reply packet for each request.

2.2.2 System Models

One of the most widely used models of computing in a distributed environment is that based on the use of personal workstations. Each user has a machine on which most of his computing is done which is usually equipped with a local disc, screen, keyboard and pointing device. The network is not used while the user is simply working on the workstation since only the personal machine, the display, input devices and the local disc are in use. If, however, a shared service such as mail is required then the user accesses an appropriate service on the network via his workstation. Other such services are file servers and high power computation engines.

An alternative approach to the use of workstations is that of the processor bank [Needham82] where the computers are not allocated to a user until requested. The user is only provided with a terminal and access to the network; when a user wishes

to perform some computation the terminal is used to request a machine to be allocated from the processor bank according to the policy currently in force. This machine is connected to the user's terminal. The machine itself does not have any permanent storage for the user's files since the same machine may subsequently be allocated to a different user and there is no assurance as to the particular machine that a user will be allocated. The network is used more frequently in this model because interaction between the terminal and the processor takes place across the network.

In the workstation model the user has total control over the hardware of his personal machine. This machine is assumed to be sufficiently powerful to cope with most of the user's requirements. If additional computing facilities are required then the workstation is used to access further resources that are on the network and are shared by all the users but are accessed in a non-transparent manner via the local workstation.

The processor bank model gives uniform access to machines in the pool of computers available; this pool may be a heterogeneous collection. The user is allocated one that meets his requirements so that only the necessary resources are given away. By this means hardware can be utilized effectively because of the sharing of the machines that is achieved, not all users being active at the same time. However, there is now more use of the network for interaction because access to the processor is made across the network.

When the workstation is located in the office of its owner there is the problem of the heat and noise generated. In the processor bank the machines and permanent storage are located in some central room where power and air conditioning can be provided and the noise and power dissipation problems of the office are removed.

Instead of workstations providing the computing facilities for a single user there are models of working that attempt to present the distributed computing environment as a single, large computer, masking the boundaries of machines from the user. This network transparency makes access to any resource, including permanent storage, similar to that found on a time shared, stand-alone system. Applications need not take into account the distributed nature of the machines

that provide the service. However, if the properties that a distributed system can provide are to be employed to solve some problem then this model is unsuitable because it is designed to masks out these features. Typically the kernel interface will provide functions that have the same semantics as a stand-alone system and will also provide additional operations to allow the distribution properties that are available to be manipulated.

2.2.3 Local Disc

One important difference in the two models of single user computing systems is the incorporation of local discs into the model. The local disc on a workstation can be used to give fast access to the data currently being worked with because no network access is involved, because authentication is not needed and because the load on the disc is less. However, the data can only be shared if the workstation explicitly provides a service to access the data and then access to remote data is performed in a different manner from operations on local data. In a processor bank environment sharing is a natural consequence of the pattern of working for the user does not store any data locally. A distinction is drawn in the workstation model between local and remote permanent data storage; the processor bank model has only remote permanent data storage.

In the processor bank model no resources connected with the machine a user has been allocated are retained when the user has finished. When the machine is next allocated any attributes associated with that machine are under sole control of the new user. Thus, if there were a local disc its contents would not be secure and its state would be unknown should the first user ever be allocated the machine again. To reuse the data on a disc would require verifying the contents against some reference copy before use thereby negating any benefits that might have been gained. All permanent storage must be controlled by a network service that can impose authentication of the client and maintain the integrity of the filing system.

In the workstation model, local disc on the user's own machine is not administered by the network system and use of this resource is at the user's discretion. The contents of the local disc are controlled solely by the user so that it can be

used for data, securely and without verification each time. When the user is accessing some other machine the situation is similar to the processor bank model. Having finished, the user will return the machine to the controlling authority – the system, if it is some special purpose computer such as one with a faster processor, or another user, if it is another workstation being used with the owner's consent. The contents of any local storage are no longer controlled by the user.

When a user is on his 'home' machine the local disc provides a valuable resource. It can be used as a cache of data from a file server enabling faster access by cutting out overheads from the network and from the service that administers the remote disc, it can be used as storage if the file server is not accessible because of network failure or failure of the file server and it can be used for temporary data that does not need to be backed up and will not be shared with a user or application on another machine. However, the administration of the local disc is completely under the user's control so it is the user who is responsible for the archiving of the disc to the file server or other permanent storage.

2.2.4 Sharing

The sharing of resources has always been an important aspect and justification of a local area network and both models of single user machines in local network computing assume that this is possible. In the workstation model there needs to be sharing of the special computation engines and services that allow for transfer and storage of data. In the processor bank model any machine may be accessed by different users at different times; therefore all services that a client uses must be accessible from any machine, in particular the transfer and storage of data must be done by some fixed service.

When there is more than one person working on a project, exchange of information between these people is important. The members of the project will want to be able to access resources allocated, such as target hardware or file space, but be protected against other users and faulty software. Such sharing of information can not always be planned, for it is not necessarily known at the time of creation of the data that access will be needed from some other user or application.

Although the cost of computing components is decreasing there are still compelling economic reasons for some resources to be shared. Examples include large magnetic discs and supercomputers where the cost is still too high to allow one per user as a single user does not utilize their full potential. Once some equipment is no longer in the possession of a single user of the distributed computing system there is a need to provide security, a centrally administered policy and maintenance.

2.3 Local Area Networks and Heterogeneity

The growth in local area networks has led to distributed computing environments arising in many new areas. As new networks appear in offices or research establishments heterogeneity occurs for three reasons, growth of the system, connection of other systems and the conflicting objectives of users of the system.

Within a network the resources available tend to grow as use of the distributed systems increases. However, advances in technology mean that the most cost effective solution of two years ago may not be the same today, but the equipment which is two years old is too costly, in financial and manpower terms, to be replaced. The result is a variety of equipment in the computing system. Similarly, as use of a distributed system increases, new applications are found leading to the need for different hardware and software. Thus, within a single local area network, there is potential, caused by evolution, for a variety of both hardware and software.

If two systems are in use within some organization it is often desirable to connect them. Initially, this may simply be for mail transfer but, in a larger system, services can be shared among the maximum possible number of clients so minimizing the cost per user of each service. By connecting systems together the flow of information between the two groups of users, for example two departments, is made easier by increased sharing with simpler access methods.

A distributed computing system is rarely used for a single task. Users are working towards different goals and hence their requirements for network components are different. Thus, even in a system constructed in one phase of growth,

there may be different hardware and software components to enable a wide range of application areas to be served.

Heterogeneity is a consequence of the flexibility gained over single machine systems by distributed computing environments. It does, however, present its own problems. The variety of hardware and software can make the manpower required for maintenance higher because expertise in each system is needed. Any change that is made in the environment must be made in each system whereas, in a homogeneous environment, the change need only be made to one piece of code which is then released to all machines.

One feature of a distributed system is that resources can easily be shared between the different components. If, however, there are many kinds of machines in the computing system then each will require certain support services such as file servers. When the number of any one kind of machine is small it can be difficult to generate enough sharing to bring the cost down to an acceptable level.

In a system with more than one independent filing system the user is required to handle several interfaces, each presenting a different model of filing. To make full use of each operating system the user must understand each interface. It is the user's responsibility to manage replicated data; all copies must be manually kept in step. Similarly, the user is responsible for knowing in which filing system data is stored.

2.4 Caching

The principle of caching is to store the results of a computation so that, rather than performing the calculation again, the stored result can be returned for subsequent calculations with the same arguments. Where the calculation is expensive significant gains can be achieved. The cache is a list of key and value pairs where the key gives the arguments to the calculation and the value is the result of performing the calculation with those arguments. Whenever a calculation is to be done the cache is searched for the key. If the key is found then the value is re-

turned; if it is not found then the computation is done and the result stored in the cache under the key as well as being returned to the caller.

For a distributed cache, one where there are several places on the network where key and value pairs are stored, there is a problem of keeping the various local caches valid so that the result of a cache hit is up to date. Two schemes exist; either a check can be performed to see if the result is still correct or the results can be kept up to date by propagating changes to the local caches when they occur. In the second case the local caches are always up to date so no validation is required on reading the cached result.

1. *Validate on lookup* : It is the responsibility of the cache lookup operation to check that the value stored is still up to date. The checking must be less expensive than recalculation for there to be any advantage to this scheme. An operation that updates the value held under a key need only make the change at one particular site. This distinguished site for each value is used to provide the validation operation for that value.

2. *Invalidate on change* : The results held in local caches are always valid so a lookup need not perform any checking. When a change is made, so that the computation would yield a different result for some key, it is the responsibility of the operation that made the change to notify all caches which have a value held under this key that their cache entry is now invalid. To avoid possible race conditions, arising from concurrent changes to one value, a distinguished site coordinates the changes on a data item.

The scheme that should be chosen depends on the relative costs of checking an entry, invalidating cache sites on update and maintaining lists of cache locations for each key as well as on the ratio of lookup operations to change operations; if lookup operations are common then reducing their cost at the expense of update operations will lessen the total cost. In a distributed system caching is one of several schemes that can be used to allow replication of data. It has advantages over weighted voting [Gifford79] in that the overheads are much less; a transaction mechanism is necessary to use weighted voting techniques. When a cached copy

is no longer valid a message is sent to the coordinating site to get a new copy. This may be combined with the checking of an entry or with the notification of a change.

3 Filing Systems

In any computing system there is a requirement for permanent storage to hold data across user sessions and to enable users to share data. This requirement is often met by the use of magnetic or optical media to retain information. This is, however, not the only possible solution. Where the permanent storage is to be used to provide the user or application with a more sophisticated model of working other solutions are possible. To the user of a database or persistent object store there is no concept of a file, whether or not the underlying system uses files to implement the model it presents. This thesis does not address these more specialized applications; it is concerned with the problems of conventional filing systems in heterogeneous local area networks and not with those systems that do not present the concept of the file to the user.

3.1 Objects in a Filing System

The filing system is a single entity that is seen by all users of the system. An object in the filing system is a passive entity upon which operations are performed. The types of object that are to be found in a filing system will vary depending on the features and style of the system but there are certain common types such as files, directories and symbolic links which are rather more common than others.

Files are the containers of data and are the leaf nodes of the directed name graph. Each file has associated with it certain properties such as create time, current size and the data it holds.

Directories provide a way of structuring names in the filing system. There is one or more distinguished directories called roots that define which components of the graph are accessible. Each directory has a series of named arcs that are

the next components on any path through this directory that are currently valid. Each arc will also contain access control list information.

Symbolic links are pointers to other objects that are resolved at path name lookup time and do not guarantee the existence of the object named; they map one textual name into another. By contrast, the hard links of the system, the arcs of the directory graph, must point to an object that exists and map a textual name to the system name for the object.

3.2 Naming

A name for an object is a means by which the object can be identified without possession of the object itself. It is usually convenient to have some fixed length unique identifier (UID) which a system can use to access the object in question. Such UIDs are not convenient for the user; a human readable mnemonic name is required, chosen by the user, as well as a method of grouping related objects together. Structured text name are used for this purpose. It is necessary to provide a mapping from text names to UIDs. This can either have the guarantee that the named object exists, the name being bound to the UID when the name is created, or the name can be bound to a UID each time it is used, in which case there is no guarantee that the named object exists or is the one that existed when the name was created. Both forms are useful in different circumstances.

The translation of text name to low level UID is a function that maps many text names to a single UID. A property of a tree naming structure, as found in some filing systems, is that only one text name can refer to an object so the mapping function becomes a (1-1) correspondence between text names and UIDs.

3.2.1 Low Level Names

In many computer systems there is a further level of mapping from UID to the current location of the object. This is especially true of distributed systems. What is required is for the manager of a UID to be able to locate the object named by

that UID. This may be a trivial mapping when all identified objects are stored on one server and that server is responsible for managing the UIDs of the objects it stores.

Another simple mapping that is used is for the location to be encoded in the UID. This provides a simple and fast mapping of the UID to the location of the object but it complicates the relocation of the object if, at some later time, it is desirable to change the place where the object resides. More flexibility is obtained by not requiring the UID to contain the location of the object, but an explicit map of UID to location must now be maintained. Techniques for speeding up the lookup of a UID to yield the location of the object include encoding the original site of the object in the UID, but regarding this as merely a hint to the location. If it is found that an object is not at its original site then the UID can be passed to a location service which implements a more complete algorithm to find the current address but at greater cost.

The Aegis operating system for the Apollo DOMAIN network [Leach82] uses UIDs to name filing system objects. The algorithm used to map a UID to a location involves a hint manager, which will accept hints from, for instance, the string name manager to the effect that a UID is located at the same node as its parent object. Should a hint not be found for a UID, or if the hint is not true, then a search is made of the local discs and finally the node named in the UID, which is the site that created it, is tried. This does not guarantee to find an accessible UID, remote objects on removable volumes can not be located until hints are registered with the hint manager.

Mapping of UID to location may occur several times in accessing an object. Once the correct server has been located the server will need to map the UID to, say, the disc address of the first block of the object. Similar techniques may be used at each mapping of UID such as caching the results of recent lookups or structuring the UIDs to make the mapping very cheap.

3.2.2 High Level Names

If the number of files in a system is large there is a need to structure their storage, grouping related files together into further abstract units. When the number of files is low it is possible to have a single collection of all files since their contents and names can be remembered by the user. Restrictions on the length of the name become more obtrusive as the number of names increases and the contents of each file becomes less easy to remember. What is required is a directory structure. A directory gives a list of pairs of name and UID that assists in imposing a user defined structure on all the files possessed by the user. The pair of name and UID is a link between the parent directory and the file or subdirectory named.

When given a name, called a component, the directory maps it to a UID. So, given a list of components, the algorithm to determine the object named is to take some distinguished directory to start the search, and, for each component in turn, to look up the name in the directory to yield a UID to another directory. This latter directory is used to look up the next component. The last object, obtained by looking up the last component in the relevant directory, is the object named; it is the only object on the path that need not be a directory.

To define what is currently accessible there needs to be a set of directories that are accessible without being looked up, that is, they are considered to be well-known. These directories are known as roots and the graph structure is defined to be all those objects that are reachable by traversing the graph from the roots along the links.

The list of components is called a path name and is usually written as a single string with a distinguished character to separate the components. For example, in the UNIX operating system there are two candidates for the start of the search to resolve the path name, either a current directory or the single, global root of the name space. If the path name begins with a '/' the search begins at the root of the name space, if not, then the search starts from the current directory. So path names might include `/a/b/c` or `d/e`. The separator varies from one operating

system to another; in TRIPOS names have the form a.b.c.d, using '.' as separator with multiple named roots identified by ':', as in sys:x.y.z. Other differences in these two systems include a restriction in UNIX to an acyclic naming graph, whereas TRIPOS allows arbitrary cycles, and the character sets allowed in a component. Only alphabetic characters and '-' are allowed in TRIPOS names, with upper and lower case being treated as equivalent on name lookup, whereas a UNIX name may contain any character, other than the separator, regarding upper and lower case as distinct.

Access Patterns

The naming graph of any filing system will not be used in an even manner. Studies for UNIX show [Floyd86] some significant patterns in the names used and the directories accessed.

There are a small number of system directories that hold the commands invoked by users that are much more heavily used than any other directory. Indeed, 75% of all references are to system directories, and the read to write ratio for these is high, about 24; for the command directories it is higher still.

Temporary directories, where intermediate data is held, have a greater proportion of writes made to them than other directories. However, the files created there are not shared between users and will be accessed by the creating program, or by another program run by the same user to process the results. An example of this is the intermediate files produced by compilers. These directories should not be placed in a network wide filing system since the benefits that could be gained are small compared with the loss of efficient access. If the operating system can identify temporary directories then it can direct operations on them to the local disc and not use any network facilities.

User directories account for a much lower proportion, about 10%, of the use made of the directory structure and have a read to write ratio of about 16. Sharing of these directories occurs but is much less than for the system directories; only 10% of user directories were observed to have more than 1 reader in 7 days (excluding

system administration), less than 1% had more than 1 writer, while the figures for system directories were 27% and 3% respectively.

3.3 Sharing

Sharing of data is important to enable communication between users. An additional feature of a filing system in a heterogeneous distributed computer system is that the various accesses made to data may be from the same user, but from a different operating system. For example, in order to be able to use some particular application the user will need to use an operating system on which it will run; while on this system the user may wish to access information created when on another system. In a distributed homogeneous there is a similar situation that arises if a user wishes to access data from a different location, but without the problems of incompatibility of the filing systems that arise when heterogeneity is present.

Whether between users, or between the same user in different sessions, sharing is not necessarily planned and does not occur until the second access is made. When stored in the filing system the data is no longer under the direct control of its creator. Mechanisms must exist to permit sharing, but these must allow for restrictions on who may access the information to protect it against unauthorized release.

Passing of data through shared files is an essential feature to enable users to make effective use of the system. When a group of people are engaged on some project there will be material that each member will want to access and be able to pass to others. At the same time, the data must be protected against accidental corruption and from users who are outside the project group. Similarly, access to the information that makes up the naming structure of the filing system must be restricted. No user can be trusted to change the filing system's data structures directly, because an error will affect everyone and because some data is confidential and needs to be controlled.

In a network environment sharing is a complex task. With no central authority

there are increased difficulties because of the different systems available and yet sharing is still desirable. Tools may exist on one system but not on another; for example, to develop software for a small system it is often preferable to be able to edit and compile on some other, larger system. There may be minimal user support in the target environment, but on the larger system there may be sophisticated editors, simulators, debuggers and other software engineering tools. To move as much as possible of the development work onto the system providing the tools can increase the productivity of the programmer. This requires the transfer of data from one system to the other, whether it is to execute the compiled version on the target machine or to process the results of some run with tools on the development machine. For both systems to share files would remove the need to move files around by hand, reducing mistakes such as executing an out of date version of a program.

3.4 Concurrency Control

When it is possible to have more than one sequence of actions being performed in overlapping periods of time there is the possibility that these sequences may involve the same filing system object. Similarly, there may be some relation between a group of objects which results in a semantic constraint, which can not be determined by the filing system, and that must be maintained by the clients. Changes to a file do not naturally occur as a single, indivisible operation but a number of updates of the files contents. It must be possible for clients to concurrently access the filing system, and to have a way of ensuring that the interaction between clients is not detrimental.

Consider a single file in some filing system which permits parts of the file to be read or written. Two situations arise, temporary inconsistency of information where there is one reader and one writer, and permanent inconsistency where two or more writers attempt to modify the file. Two or more readers of a file may coexist because they do not change the object by the act of reading part of its contents. Read operations are independent of each other in the absence of any writers.

Temporary Inconsistency

When there is one writer and one or more readers it is possible that a reader may see some of the contents before the writer has applied its changes to the file and see some of the contents after the writer has updated the file. Any relationship between data items in the file may be invalid during the time the writer is updating the file. Once the writer has finished the file is consistent, provided that the writer has maintained any invariants contained in the file. Any reader looking at the file before or after the writer's updates will see the file with any constraints still holding, but a reader looking during the writer's sequence of updates may not. Any inconsistency that may arise is only temporary, existing during the interval of time during which the writer is active.

Permanent Inconsistency

Permanent damage can be done if there are two or more writers. Consider two writers acting on a file. When they are updating sections of the file's contents each may overwrite changes made by the other, without knowing that this has occurred. The change to a section of the file that is visible when the writers have finished is that which happens to have been made last. Further, the decision of what to write may be based on out of date or inconsistent information if a writer has read the contents of the file.

3.4.1 Grain of Concurrency Control

It is possible that the presence of multiple writers or one writer and several readers will not lead to undesirable effects without any controls being applied but this can not be guaranteed or detected in advance. A database system can not afford to exclude multiple clients who are acting on different parts of the database and still achieve acceptable availability. Therefore, the database is split up into records with a concurrency control scheme applied at the grain of a record, ensuring a safe situation for clients acting on independent parts of the file.

Traditionally, filing systems have used a multiple reader or single writer policy (MRSW) which allows for either many readers or just one writer but not both.

The grain of locking is the file. This grain can be chosen because it is rare that there is conflict in the intentions of clients, that is, a system will only infrequently have to refuse or delay a client's request to access a file because of the presence of another client. The locking policy is applied when a client states its intention to perform a sequence of actions by opening a file. The lock obtained stops the MRSW property being violated.

A coarse grain of locking decreases the overheads in performing a sequence of operations and spreads its costs over more operations but if the sequence occurs over a long period of time then the availability of the file is reduced. A trade-off exists between maximizing throughput, that is, the total number of actions that can be performed in a period of time, by decreasing the amount of system work, and the average response time for a single operation due to locking costs or lock contention. A common design decision is to apply locking at the grain of the file with the lock being obtained during the open operation and released during the close operation.

Where a locking scheme exists it may be enforced, so that a client must obtain a lock that is compatible with the operations it intends to perform, or advisory, where there exists a subsystem that will apply a locking policy but its use is not mandatory. The guarantees the latter can provide are only valid if all those accessing a file abide by the scheme.

3.4.2 Immutable Files

Some filing systems [Schroeder85] make files immutable objects. Once written a file can not be changed so that the problem of concurrency control on one file is avoided because the file does not become visible to potential readers until the writer has finished; once visible to all clients a file can not be changed. If the contents of a file are to be altered then a new version of the file is created. The rule that if a reader does not specify which version of a file is to be read then the highest version number is chosen may be used. A problems that must be addressed are that, once a client has read the highest version of a file, the filing system must be able to guarantee that the client is about to write the next version, and that

no other version will exist between the copy read and the copy written. This guarantee ensures that the information used to create the new version is up to date. Another problem is one of controlling the number of files. Since file space is not automatically reused by overwriting in place a mechanism to return unwanted file space is needed, whether the medium is write-once, such as an optical disc, or magnetic and may be written several times. The need to control which versions are to be kept for a write-once medium arises when a copy is taken to a new disc to create free space for further allocation. At this time unwanted file space must be detectable.

3.5 Access Control

Once there is one single scheme that allows sharing of data there needs to be control over which programs may access the data because it will occur that sharing should be permitted between certain entities but not others. There are two classes of entity, subjects, who are active and who may perform operations on data, and objects, which contain data or naming information used by the system [Lampson71]. A subject may be a program or a user and must have an identifying name. The access granted may depend on context such as which terminal the program is being run from.

For each pair of subject and object there is a set of accesses that can be granted upon request. Given the large number of such pairs and the redundancy in this matrix (many pairs will have some default access) it is necessary and desirable to compress the access control information. This leads to two styles of protection, access control lists and capabilities.

3.5.1 Access Control Lists

An access control list (ACL) is attached to each object in the filing system. It contains a description of which subjects may perform the various possible operations on the object. Often this will be held as a number of pairs of subject and access for that subject; to compress the ACL further and to simplify the naming

of collections of subjects, a group may be specified as an ACL member rather than a subject. The group name avoids the need to enumerate every group member and simplifies changes in the access to more than one object. Now only the record of the group membership need be altered and it is not necessary to locate every object that the group may access in order to add or delete a member. Group membership will usually express some logical connection between its members such as all the maintainers of a software system.

With the introduction of groups a subject may be mentioned more than once, either explicitly or implicitly as a member of a group, with different accesses. When a subject is looked up to determine the access that can be granted the access is normally taken from the union of all rights the user can gain on this object.

Coupled with the ability to grant access to a subject is the ability to revoke rights already granted. In an ACL system this requires enumerating all subjects who may access the object in question which is simple because they are listed with the object. The list can be modified to remove those who are no longer to gain access, with care being necessary to ensure that the subject may not still gain access as a member of some group. Negative rights can also be used to forbid a subject access to an object. These override any access right granted in the list.

3.5.2 Capabilities

An alternative to storing protection information with the object is to have the information associated with each subject. A capability is a name of an object and some level of access to that object. To perform an operation on an object a subject presents an appropriate capability to the object's manager. The capability serves to name the object and to grant access. This may be viewed as compressing the access matrix by subject, rather than by object as in the case of ACLs.

A capability must be protected against modification by the subject so that additional rights are not acquired; similarly, possession of the capability must be controlled to prevent unauthorized subjects acquiring it. In a non-distributed computer system hardware support has been used to implement capabilities in

an efficient manner [Wilkes79a]. In distributed systems such centralized management is not possible so capabilities are either encrypted [Mullender84] or labelled with a random number. This random number is taken from a sparse name space [Girling82] and this gives a very low probability of another subject guessing the random identifier and hence the capability.

A related case is access control imposed by associating a password with a file; to access the file a path name and a password must be supplied; together, this pair may be thought of as a capability.

Revocation in a capability system needs some care. It is difficult to locate every occurrence of a capability, especially if they may reside on secondary storage. If, instead, the owner issues a capability for the controlling capability, not for the object itself, revocation is achieved by compelling access by third parties to proceed via the owner's capability, and not by going directly to the object [Redell74]. An alternative is to check that a capability is still valid at the time of use, but this is additional checking which must be performed on every access.

3.6 File Servers

Permanent storage in a network computing system is usually provided by magnetic disc. Large discs are cheaper per kilobyte of storage than smaller ones so there is economic pressure to make efficient use of large drives. However, a single user does not need all the space of a large unit. The solution to these problems is the file server [Svobodova84] whereby a service is provided that allows access to one or more disc units. By centralizing the service, maintenance may conveniently be provided and archiving of the information on disc to magnetic tape can be performed automatically.

The interface of the service can be at several levels. At the lowest level the interface to magnetic storage provides functions that manipulate the disc in the manner of a device driver. The physical nature of the disc is entirely visible to a client who must understand all of the various types of disc that are provided.

The next level of abstraction is that of the virtual disc. A single model of a disc is provided by the server so that the physical characteristics of each type of drive are not visible to the client. Instead, the client makes requests referring to areas of disc by block number but it is the clients' responsibility to manage those blocks. Clients are trusting one another to perform changes correctly. Sun's ND is an example of this.

When management of the blocks on disc is introduced we have a block server. Now the server understands ownership and allocation of blocks to clients and will refuse requests that refer to blocks not owned by the client. This approach is taken by the Amoeba system in its block server as part of a structured file system [Mullender85].

The term file server is conventionally used to describe a server that has an abstraction of a group of related blocks that may be referred to as a single object through a file identifier. A simple example of this is the Woodstock File Server [Swinehart79]. Indeed, the notion of a block may not be visible to a client and files are to be thought of as a sequence of bytes or records.

A universal file server [Birrell80] is defined to provide the abstraction of a file and to mask the underlying physical characteristics of the discs used but it also introduces the notion of an index, where the identifiers of objects on the file server can be stored. Objects in the directed graph, defined by all indexes and files that can be reached from some distinguished root index, are guaranteed to be preserved, while objects not reachable from the root are periodically garbage collected and the space released for future allocation. By use of indexes and files a client can construct a directory structure on top of the basic facilities of the universal file server.

Very high level interfaces have also been provided whereby a complete naming structure is provided with access control and authentication of clients (eg [Richardson84]). However, such systems are designed to operate in one particular environment with particular problems to be addressed. Their specialized nature makes them unsuitable for use in a general heterogeneous environment; they impose a single model of file access and naming that does not allow for other filing

systems to coexist because they are designed to support only one particular type of client.

4 Related Work

This chapter looks at a number of systems and discusses the problem areas addressed, the assumptions made and solutions devised. The systems discussed are :–

1. The Washington Common Store – heterogeneity and sharing.
2. CFS – the filing system for Cedar.
3. LOCUS – a distributed UNIX-like filing system.
4. The ROE filing system.
5. TRIPOS – filing in a local area network.

4.1 Design Influences

A design for a filing system in a distributed computing environment is aimed to solve the problems presented by the environment it will be operating in. There are various factors that can be used to define the environment; the relative importance of these will decide the nature of the filing system and allowance must be made for the fact that they are not all independent issues.

- *Heterogeneity* : If the filing system is to support just one operating system the interface and operation of the service can be tailored to this single client. Should there be a wider range of client systems then, potentially, there will be a loss of efficiency because of conflicts of interest.

- *File Replication* : If high availability or performance is deemed necessary then a method of file replication may be needed. The trivial case of a single copy generates low overheads but the server holding the file is a potential

bottleneck. At the other extreme, a weighted-voting scheme [Gifford79] may be used to allow for some copies of the file to be out of date while still ensuring that a client will always read up to date information. Transaction support is needed for weighted voting and may itself be used as a way of managing file replication by keeping all copies up to date (similar to a voting scheme with a read quorum of 1 and write quorum equal to the number of votes). Caching can also be used to provide replication of files (see 2.4). The objective of all these schemes is to enforce consistency on the data seen by the filing systems clients.

- *Scale* : It is difficult to take a filing system and increase the number of clients that it can handle. Therefore, the size of the final system must be taken into account in the design. A related issue is whether access will be made across a uniform network, such as a local area network, or whether some service components or clients will be separated by lower bandwidth communication links.

- *Network Environment* : The level of security of data that can be achieved, or is desirable, will depend on the support in the environment for authentication and authorization. Even though a filing system will impose access control polices, there may need to be checking of the identity and current rights of clients as they identify themselves to the service. The method of data transfer which is most suited to the network may influence the design of the service's interface. If a model of remote procedure call can be used then this may be chosen to give a model which is readily accepted and understood by programmers.

4.2 The Washington Common Store

The Washington Common Store (WCS) [Black86] [Black87] is part of work into the interconnection of heterogeneous computer systems at the University of Washington. Principle objectives are to reduce the cost of introducing a new system and to increase the set of common services that can be accessed. Filing,

electronic mail, printing and remote computation have been identified as the key services, and these will be constructed on an RPC mechanism, a naming service, to provide uniform access to all underlying naming services in the component systems, and an authentication service.

A Heterogeneous Remote Procedure Call (HRPC) is the basic communication mechanism. It is used to access WCS and the on-the-wire format of the HRPC system is used as the format for the data in a file. WCS supplements the filing system of each host so that sharing between systems is achieved but is not automatic; only information explicitly stored in WCS can be shared. This speeds the introduction of new equipment, for only the HRPC mechanism needs to be installed for access to WCS to be possible, but there is not transparent access to remote files and hence application programs need to be changed in order to access information in WCS.

WCS is designed to support file access at the language level; each file has an associated type, which is a sequence of user-defined records, that is type-checked as part of the HRPC access. This ensures that when a file is read there will be no misinterpretation of the data format of the file, although there is no guarantee that implicit constraints on the information contained, or the higher level abstractions to be placed on the data, will be understood. This still requires agreement between accessing applications.

To describe the file record type the user constructs a Courier IDL description [Xerox81] which has been extended with FILETYPE and access is made by standard utility programs on a whole file basis, or by reading or writing records under program control. Files held in WCS are immutable, which is useful for such shared information as it will not become visible until it is in some consistent state. Text names are used to identify files but clients of WCS may construct their own access structures by using low level primitives which use file identifiers. There is currently no description of access control, which is essential for controlled sharing of objects, but it is noted that the environment is particularly cooperative, and hence social pressures can be used to replace strictly enforced access control mechanisms.

4.3 The Cedar File System

CFS [Schroeder85] is, like WCS, designed to provide permanent storage for archive and for sharing. The local disc of each workstation is expected to provide the principle file space for a user's personal files. The form of sharing that CFS is particularly designed for is that of a software subsystem, a set of files which gives a consistent snapshot of the state of some application under development. A DF file describes which versions of each immutable file make up a description of the state; by writing the DF file last the snapshot is made atomic.

The advantages of immutable files are that problems of concurrent access are largely eliminated, since a file does not become visible until the creator has finished writing the file. From then on any number of readers may be permitted without checking since no further updates can occur.

It is possible to specify that the highest version be read and this depends on the history of the files; references to versions in DF files are by explicit number. Although the need for locking on a file is eliminated by the use of immutable files, there is still the possibility that the creation of a new existing version should be based on the latest versions of a number of other files. To provide this consistency a multi-file lock is needed.

When a file which is part of a software sub-system is being modified, it is cached on the local disc of a workstation after the appropriate DF file has been used to bring over all the files in the software sub-system. The implementation of this exploits the immutability of files by only setting up entries in the local filing system name space as attachments to the remote file, without reading the contents of the file until they are actually needed. This design arose from experience of DF files when used on XDFS [Sturgis80] in XDE (Xerox Development Environment). There, the actual file contents are read onto the local disc at the point where the DF file is accessed. This leads to a long delay in setting up a software sub-system which is more conveniently spread over subsequent file opens. Unnecessary work is avoided under the Cedar scheme for files that are not used are not transferred; under the XDE scheme the contents of each file are transferred from the file server.

The immutability of the file guarantees that it will not have changed between referencing the DF file, creating the attachment, and reading the contents onto the local disc.

Version numbers are used to label modified files and this gives a nearly unique name; if the highest version is deleted then a version number may be reused. It is possible to specify files by creation time and this is typically used by tools with the version number giving a more convenient form for the user.

Many of the novel features of CFS are driven by the unusual environment in which it is used. This environment excludes any heterogeneity of systems, so conflicts of naming semantics, data representation or operational semantics are not issues of relevance.

4.4 LOCUS

LOCUS [Popek81] [Walker83] [Weinstein85] [Kline86] is a distributed operating system which presents users and application programs with a view of a single computing system that spans multiple machines in a local area network in a transparent manner. By making the features of distribution transparent it aims to make it possible to access both local and remote resources through the same interface. There is replication of data and an emphasis on the reliability and availability of the system without decreased performance, even when network partitions occur.

Although the LOCUS system runs on a variety of hardware there is homogeneity at the software level. Separate nodes run the same operating system and hence share common models of many components of the system, such as processes and semantics of operations, and so can easily pass information relating to such items between themselves. In the case of the filing system any kernel can update an object if it resides on that machine, or it can request the update to be made at a remote site, trusting that the kernel on the remote site will perform the operation correctly, and that it will not allow any information to leak out to unauthorized sources.

A solution to the problem of each type of machine needing different load modules for different hardware has been implemented. When a name is looked up in the file name space it may be resolved to give a hidden directory which contains a number of files, one per class of hardware. For example, /bin/rm names a hidden directory; this is a new class of object in the filing system, distinct from ordinary directories. This directory is searched to find a file name that matches the hardware being used and it is this file that is loaded and run. A user will not see /bin/rm as a directory but as a file; there is an additional interface that enables files in hidden directories to be accessed. This provides a way of using the same name from different hardware to access binaries with the same functionality. The interface through which the object is accessed depends on the use to be made of it; when installing a new version of a program it is necessary to have a different view of the name space from when a program is being loaded. Since this must be done for any binary files for programs that run on the various hardware, not just those in system directories, all users will need to be aware of the access methods for hidden directories.

One of the ways in which increased availability of LOCUS is achieved is by allowing work to continue even in the presence of network partitions. Protocols are run on each node to establish the members of a partition component and to merge components when contact is restored. In each component there is a current synchronization site for each filegroup, so components act as independent LOCUS systems, although with the same arrangement of mounted filegroups. When contact is restored between two components there is a need to resolve any conflicts that arise between objects that have been updated, created or deleted while the partition existed.

LOCUS identifies four classes of object in these circumstances; mailboxes, directories, databases and other files. In the case of directories and mailboxes the limited number of operations, with simple semantics, that can be performed means that many conflicts can be resolved, for example a directory entry being created in one component, but not in another, results in the new entry being propagated everywhere. However, this is not guaranteed to be the correct decision. Suppose

an entry has been created in one partition because a program running there read a piece of information in a file. Suppose further that this program runs in the other components during the partition, but that the file has previous been deleted in those components and that if the file does not exist then the program does not create a new directory entry. Then, when the network partition is removed, the rules for propagating changes to directories indicate that the new entry should be propagated. However, this is not correct. The file containing the information used to make the decision is not propagated, indeed may be deleted during partition component merger, as it has been deleted in all other components.

If a conflict arises that can not be resolved simply, for example two entries created with the same name, the user is mailed to announce the conflict and is expected to resolve it manually. Any conflict of files is handled in this latter manner. Databases that have conflicts are reported to the database manager in case it can resolve the problem. If this can not be done a mail message is sent to the user.

This method enables work to be continued during a partition. It makes the assumption that work in different components, which is not part of the same transaction, is sufficiently independent that the existence of the partition will not lead to a conflict of interest. Mechanisms exist so that applications may be notified of the existence of a partition. Thus, cooperating actions can be placed in a transaction and may take action particular to their common objective. However, there is a reliance on knowing which activities might result in conflict. There is no serialization or atomicity between an activity that is a part of a transaction and one that is not.

4.5 The ROE Filing System

An example of a network wide filing system, one of whose objectives is to allow for heterogeneity of computing systems that use it, is the ROE filing system [Ellis83]. Heterogeneity exists at the level of hardware and at the level of the operating systems that are run. While allowing for experimentation into file dis-

tribution and replication, ROE also hopes to present a single, coherent file system to its users without requiring the users to be aware of the location or replication of files.

A new abstraction, called a Roefile, is introduced which gives a textual name to a collection of file identifiers for the copies of the file or directory. This collection can change with time so that the copies used to give the replicated Roefile may be varied. Consistency of these copies is achieved by use of weighted voting with the assignment of votes being used to indicate the quality of service provided by the storage site in terms of availability. By giving more votes to the more available sites the overall availability of the Roefile is increased.

Directories are not locked as this would limit the concurrency of operations; instead the directory entries are the unit of the voting algorithm although all the entries in the same directory have the same number of votes at any given site. Updating directories requires a read quorum to determine the current version, although the read quorum may not consist solely of up to date copies, and a write quorum taken from those copies in the read quorum that are current. This permits directories to be accessed over a long period of time by a user without excluding changes during this period; the user will always see the most recent version of the directory. When an update of an entry is carried out the old data, which is to be replaced, is also sent so that it can be checked with the actual state in case an update has occurred between collecting the write quorum and performing the change. All these techniques are aimed at increasing the availability of a replicated directory.

Bulk data transfer is achieved by a uniform file transfer protocol, used both for files which are part of ROE and those which are not part of the network file service.

4.6 The TRIPOS Filing System

As well as providing an example of a system that uses only network facilities to provide its users with all permanent storage, TRIPOS has also been used for

work detailed in chapter 7 on changes needed to a client operating system to enable it to function in the environment described in chapters 5 and 6. Therefore a more complete description is given. Two forms of filing system are described; the original distributed system based on network file servers and a more efficient form where a specialized front end to the file servers is used to provide a filing system designed to support TRIPOS.

TRIPOS [Richards79] is a message passing operating system that runs on a number of mini and micro computers. A small kernel supports allocation of memory, processes, which are known as tasks, and message passing between tasks. There is no differentiation of tasks into user and system, but there are a number of tasks that are part of the standard system that provide the system facilities. Machines capable of running TRIPOS are attached to an extended Cambridge ring network, with the machines being allocated on demand from a processor bank containing a number of types of machine. TRIPOS is written in BCPL [Richards80], as are many of the application programs run on these machines.

Message passing is simplified because of the single address space for the machine; an area of memory supplied by the sender, called a packet, is passed by linking it into the work queue of the receiving task. The packets are required to have the first field available for use in linking into this queue with the second field giving the task number of the destination. After this there is no compulsory format, although most tasks expect the conventional format detailed in figure 4.1. The packet is used for both the request and the reply. The field **type** gives a function code, enabling the receiving task to demultiplex incoming packets.

When a packet is transmitted the kernel changes the field indicating the destination to be the task number of the source so that the packet is in a format that can be sent back as a reply. The reply packet is the packet sent with the two result fields, previously undefined, now set.

4.6.1 Naming

The filing system is an arbitrary graph with a number of named roots. One such file name might be **sys:c.bcpl**, where **sys:** gives the root for the name lookup, **c**

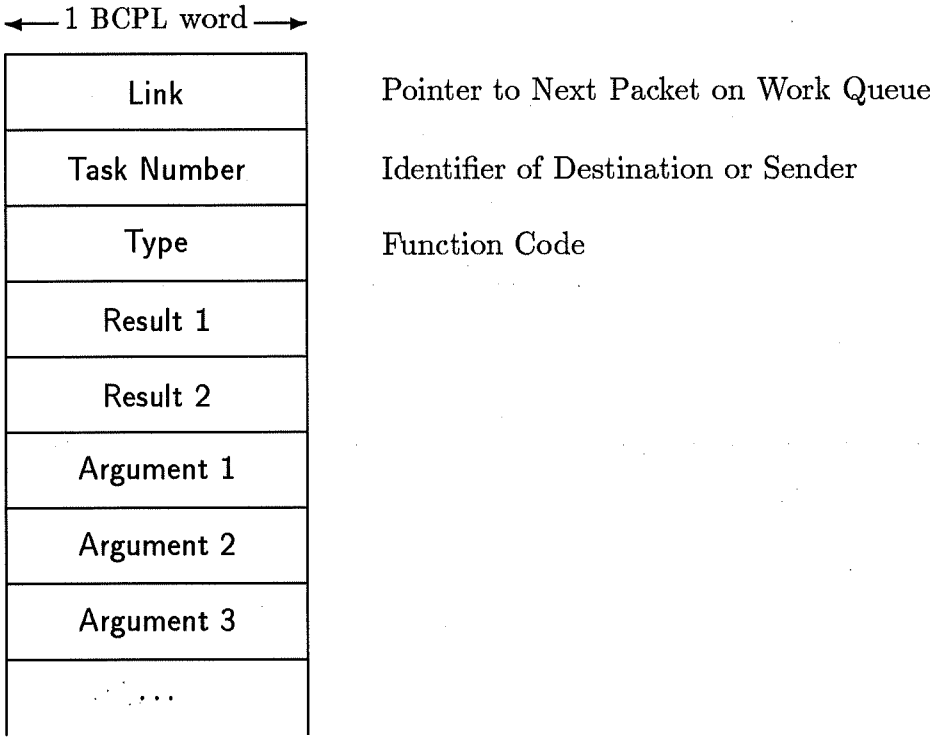

Figure 4.1: Conventional format of TRIPOS packets.

is the name of a directory in that directory and bcpl is a file in the sys:c directory. When a machine is loaded with TRIPOS for a given user, initialization of the machine includes setting up a number of roots to the filing system. This is carried out before the user may make any use of the machine because creating these roots requires refinement of objects that give greater access to the filing system. Such roots include sys:, the root of the filing system; t: the user's directory for temporary files; home:, the user's home directory, and any special accesses granted to this particular user.

With each root there is an associated access for the user to the filing system object named, which is taken as the initial access for the start of path name lookup. In practice the rights on these roots for each user change very slowly and, although it is possible for a name to convey different rights from different machines for the same user, it is usual for a name to carry the same rights for a given user from any client machine. A user may create additional roots as short names for commonly used objects but the accesses of these can be no more than could be obtained by using a path name from a root given during the initial setting up of the machine.

The problem of heterogeneity of the hardware on which TRIPOS runs is tackled by having different roots to the filing system for each of the classes of hardware. Thus, sys: refers to different objects depending on the hardware the system is running on. Then, system commands are picked up from different directories even though the same name is used. Users must make their own arrangements if a similar effect is to be achieved for their own programs.

4.6.2 File Servers

Permanent storage is managed by file servers [Dion80] [Dion81] which provide a universal file server interface [Birrell80] of files and indices named by fixed length UIDs. From the primitives provided clients may build the filing system structure of their choice. There are three types of object seen by a client of a file server :

- *Normal Files* : A normal file is just a container of data, accessed as a sequence of 16 bit words.

- *Special Files* : A special file is a normal file which has the additional property that a sequence of updates can be made atomic. Such a file can be used to hold data which is always self-consistent; should a crash occur before a special file has been committed then the state is reset; if it occurred after the sequence of operations had been committed then during recovery the updates are completed. TRIPOS does not provide special files to the user but uses them itself to implement directories. A user can access the facilities of special files by sending requests directly to the file server, bypassing the usual TRIPOS interface to the filing system, but the user is then responsible for the safety of the directory structure.

- *Indices* : An index is a list of UIDs on which operations are atomic. To create a new file or index, the index that the new UID should be stored in is supplied; this ensures that the object is not created and left unattached.

Objects are deleted by making them unreachable from the root index. At some later time the resources used by these objects will be reclaimed during garbage collection if they have not already been released by noticing that the reference count has reached zero. Only UIDs held in indices on the file server are considered during garbage collection; if any are stored in files or outside the file server then the user or application is responsible for ensuring that a copy is in an index as well to preserve the object named.

Concurrent file operations can be controlled by opening a file before performing a sequence of operations. The **open** operation returns a new, unique, temporary identifier for use as an interlock which can be supplied with later requests to access the file or index; any other **open** requests received while the interlock identifier is still active will be subject to a multiple reader or single writer policy. Any request that arrives, not specifying an interlock, will be enclosed in a notional **open** and **close** pair to create a temporary lock.

4.6.3 Experience with Cambridge File Servers

The file servers rely on each client to manipulate the filing structure correctly and perform any locking necessary. This means that each client must have a

complete copy of the code that implements the filing system; this is a limitation on small machines. Access control is advisory, for any user must be able to get the UID of the root that defines which files are reachable in the TRIPOS naming structure. From here a malicious user can get the UID of any file or index and hence change its contents. However, the user does not have access to the other filing systems that may exist unless explicitly given a suitable UID.

The scope for caching is restricted by the memory size of the machines used for TRIPOS; this restriction also leads to large numbers of small sized data requests rather than more efficient large block requests. However, there are also limitations on what may be cached because of the design of the file server. The results of path name lookup can not be cached for, at any time, another TRIPOS system may make some alteration to the filing system to invalidate the cached results. To validate the cache is as costly as performing the operation again. All operations must use the full locking scheme if they need to guarantee the state of some part of the filing system. Locks can be used but it is highly undesirable to hold these locks for any longer than necessary and additional file server request are needed to manage them.

The lack of facilities for caching to improve the performance of the file servers means that performance is acceptable with a few machines but as the number of TRIPOS systems accessing a file server increases the performance degrades rapidly.

The primitives provided by the file server allow many different filing systems to be constructed. However, access to files in a filing system requires an implementation of the interface in the accessing machine and a suitable UID for start of any path name search. Thus, access to each filing system is not through a single, uniform interface and requires trust between filing systems because a UID needs to be passed from one system to the other; unfortunately UIDs give full access to the object named so, if it were an index, it would allow full access to all sub-objects. However, it is possible to share just files if the design of a filing system allows arbitrary UIDs to be embedded in the naming graph as objects, then one operating system may pass a UID to another so that one file may be part of two filing systems. UIDs are held in the user's machine when it is running TRIPOS so

security of other filing systems is compromised if UIDs are passed to TRIPOS.

4.6.4 The Filing machine

The principle objective of the TRIPOS filing machine [Richardson84] is to improve the performance of TRIPOS systems working in this environment by reducing the number of file server operations. This is achieved by caching and removing the need to lock at the file server.

The TRIPOS filing machine contains the only copy of code needed to perform file server operations and it acts under the assumption that all accesses to TRIPOS files and directories will be passed to it, rather than being performed by the client directly. This enables the filing machine to operate a cache of files and directories. The cache removes the need to go to a file server at all if the data is already in the memory of the filing machine. The cache is read-ahead and write-behind. Data can be read in advance by performing a larger file server operation than the client request needed in order to have data in the cache before the client makes a later request; this takes the access time of the file server off the critical path for many of the requests received. This works particularly well since most file access is sequential. Data written to a file is buffered and written back asynchronously, with synchronization occurring only when the file is closed; at this point a reply is not sent to the client until the data is on disc at the file server.

Another optimization based on the assumption that the filing machine is the only direct client of the file servers is that locking can be performed in memory in the filing machine, without needing an interlock to be created at the file server. This again reduces the number of file server operations that are performed.

Although the TRIPOS filing machine increases the performance of TRIPOS systems there can only be one such machine in a network, which limits the scale of distributed system that can be supported. With increased numbers of client TRIPOS systems the filing machine becomes a bottleneck. To distribute its functions would require communication between the various filing machines to manage the cache and locking.

Entry Rights

D	bit	This entry may be deleted.
A	bit	The access bits for this entry may be altered.
U	bit	This entry may be updated.

File Rights

R	bit	The file may be read.
W	bit	The file may be written.
E	bit	The file is executable.

Directory Rights

C	bit	New entries may be created in this directory.
V	bit	V access is granted to the directory.
X	bit	X access is granted to the directory.
Y	bit	Y access is granted to the directory.
Z	bit	Z access is granted to the directory.

Figure 4.2: Access rights in TRIPOS

To work with the filing machine all TRIPOS systems have the normal file server handler task replaced by one that translates packets from applications into filing machine requests. This is simple because one request by an application generates one filing machine request, except for a few operations that are performed internally in the file handler of the client machine. By replacing the original file server task by a file handler task for the filing machine which accepted similar, but not identical, requests the form of the TRIPOS filing system was maintained. Many programs required no source code changes to work in the new system; changes to the interface were mainly due to a more sophisticated access control mechanism made possible because client machines no longer needed file server UIDs to read objects.

4.6.5 TRIPOS Access Control

The access control mechanism provided by the TRIPOS filing machine is derived from that found in the CAP computer [Wilkes79a]. A directory entry has an access matrix associated with it of 4 rows, named V, X, Y and Z. In each row there are

two bit patterns which represent the access rights, the first pattern refers to the entry itself and the second to the object named, which is either a file or another directory. See figure 4.2.

The access permitted is the union of those rows of the matrix which a user is allowed; this in turn is decided by the VXYZ bits in the entry naming the directory under consideration. The access for an entry is written as a list of the rows in the order VXYZ. The defaults are /DAURW/ARW/RW/R for files and /ACV/ACX/CY/Z for directories. The initial access is determined from the root name given as part of the path name. The E bit is unused in TRIPOS because it is not possible to distinguish between reading and loading a file.

Conventionally, V is thought of as the user, X as close friends, Y as other friends and Z as the default access. A study of the actual use made of access control shows that few entries deviate from the defaults and that only very exceptional use is made of the X and Y bits. Only a tiny number of programs use the access control functions of the interface or interpret the access control bits. These are all system provided utilities such as directory listing commands and the command to alter matrix entries.

5 The Design of a Filing System

The facilities provided by a distributed system are not constant and both hardware and software components will change as the old is replaced and as additional equipment and applications are introduced. Such hardware and software can be difficult to incorporate if a sufficiently flexible model of working is not adopted.

By acknowledging the problems of heterogeneity and the need to allow for growth at the design stage the work of administrators and maintainers can be reduced. Benefit is brought to the users of the system by a more systematic approach to the computing environment and from the speed at which new demands can be met by introducing new hardware and software. Therefore, one of the goals of this work is to present a system whereby new operating systems, file servers and application programs can easily be integrated into an existing system.

5.1 Resource Utilization

Two important considerations in the design are that client software should be able to utilize resources efficiently, giving users good performance, and that it should be possible to access a wide range of facilities without requiring the users to have a detailed knowledge of the underlying system. Unfortunately these may conflict with the objective of easing integration. By placing functionality in a server, changes will only need to be made in that server; however a bottleneck may be created. If we distribute the function to, say, the various types of user machine then there are more places for updates to be applied. For example, we might either have a cache as part of each file server or have clients caching files. In the former case clients are simpler and every client benefits from the caching but all the clients accessing one file server use one cache so the caches may become

overloaded; in the latter case each client is made more complex and a change in policy in the system may require changes to every client.

If there is one system, rather than a series of smaller systems loosely connected together, it is easier to provide some services. An example of this is electronic mail which is becoming increasingly important in many environments. It is a more useful tool if there is one mail system covering all users, rather than a separate system for each operating system in use. Mail can then be delivered to a user's preferred machine, and a user is not required to log on to each machine running an independent mail system.

5.2 Problems of Heterogeneity

Every operating system has its own model of a filing store and, while it is possible to identify many concepts that are common between systems, some fundamental differences remain.

5.2.1 Importing Software

New application software, not explicitly designed to work in a particular environment, can take a substantial amount of time to adapt to the local system. One large group of applications, which it is highly desirable to be able to use with ease, are those designed to work on existing, stand-alone operating systems. We would ideally like to have each such operating system provide its local semantics to support the running of existing application software in an unchanged form. The kernel translates between the local semantics of the stand-alone system and the network facilities to support file access. Localizing the changes to the kernel, rather than requiring modifications to all existing programs, will reduce the total work needed to move to the new environment. A benefit of this would be that programs supplied without source code could still be used.

5.2.2 Developing New Software

As well as being concerned with facilitating applications to be brought into the system, we must also consider new software constructed specifically for this distributed environment. There are features of the environment which client operating systems do not normally provide so we must allow special software to run which is designed for the distributed environment, for example a mail system, and for software to enable maintenance and repair to be carried out. To facilitate this, an operating system should provide additional operations which give direct access to the functionality in the distributed system. These operations do not apply translation from the client model of the system to the network model. Using this interface an application program can work directly in terms of the network model, and can, for example, specify file names with the full range of characters allowed by the network filing system as described in 5.2.4.

5.2.3 Data Representation

Once we allow two different systems to access the same filing system there are new classes of problems. While some systems see a file as being a sequence of bytes, numbered zero to the upper bound, others see a file as a sequence of records. Even among two systems both viewing a file as a sequence of bytes there is a problem of the representation of multi-byte objects such as 32 bit integers or floating point numbers; similarly, where two systems view a file as a sequence of records, there is a problem of interpretation of the contents of the records. Two applications accessing a shared file must also have the same high level understanding of what is represented, placing the same abstractions on the data.

One solution is to have a standard for the representation of data with network wide formats for the primitive types, such as integers, and provide for the construction of new types from these (for example [ISO85a] [ISO85b]). However, sharing between systems, while still being desirable, is not the most common use of the filing system. It is to be expected that a file will be used by the creating system most of the time. Imposing a standard may result in translation costs being incurred whenever a file is transferred across the network. The size of the file may

also be significantly increased if the information is stored in a form that allows for a wide range of data types. There is no complete solution; data descriptions can only convey the nature of the file's contents in terms of basic types and the construction rules. User defined types can not be completely described because they may involve implicit constraints within the data items forming the value of a type. A complete description of such constraints is not possible. Sharing of data will always require agreement on the interpretation.

It is important to give the local semantics for file access to an application, both in access method and file format, and therefore files should be transferred to the client machine as a unit, rather than providing operations to manipulate parts of them. Once under the control of the local machine a file can be interpreted as an unstructured sequence of bytes or as a sequence of records, whichever is appropriate to the local operating system. When another application on a different operating system is handling the file it is responsible for interpreting the information in the correct manner. It will need to be aware that the file was created on another operating system but, as described above, this is necessary in order to interpret the data. Whole file transfer allows for any file structure to be stored in a file server in a convenient manner, without the file server needing to understand that structure. Interpretation of the sequence of bytes is left to the clients. In many cases it can be expected that a standard representation of data, such as ASN.1, will be used for files that will be accessed from many operating systems. However, the decision not to use such a standard does not preclude sharing or the use of a specialized convention for reasons of efficiency.

5.2.4 Naming

The naming of objects in a filing system is complicated by differences in the sets of characters allowed to appear in a component name, different choices of component separator and differences in the permitted structure of the directory graph. With care it is possible to choose names that are universally accessible, for instance, by always choosing components that consist of letters, and by not creating components that differ only in the case of the characters in some part of the name. We have assumed that separators are from the set of non-alphabetic

characters; it is unusual to have the separator as, say, 'X'. This principle of satisfying the majority when unable to present a universal solution is the only course left open if we are to attempt to cope with heterogeneity. By specifying a path name as a sequence of components, rather than a single string with separator characters embedded in it, each operating system can use its usual separator leaving the application unchanged.

A simple solution to the problem of naming files across client systems is to enforce a set of rules that allow a large majority of operating systems to name all files. This contradicts the objective of providing local semantics to application programs which might make use of facilities which violate the network rules. For example, the UNIX *make* utility, used to maintain groups of related files which make up one or more programs, will look in files '*makefile*' then '*Makefile*', if no other file is specified, to find a description of the dependences of the group of files. This utility is so widely used that changing any aspect of it would have a great effect on users.

Therefore, it is not correct to enforce a network standard for file names. Programs may view the filing system as if it were the client's usual filing system. As programs which share data must already agree on certain features beforehand they can also agree on a file name which is acceptable to both of them. If sharing is not planned for, use of the extended interface of the operating system may allow an application to use a particular file name, but this is restricted to applications designed in the local environment. For two arbitrary applications, disagreement on file names will preclude sharing unless symbolic links are used to resolve the differences.

5.3 Features of a Distributed Filing System

A distributed environment generates further requirements for a filing system over and above those of a filing system in a stand-alone computer. These arise from the separation of the components involved and the new modes of failure that are possible. It is usually assumed that the failure of a subsystem in a stand-alone

computer will lead to the shutdown of the whole system which will be brought up again following the normal bootstrap procedure. In a distributed system this is not necessary, and is undesirable, because the separation of the components means that it is possible to continue operation even in the event of failure of some subsystem. Separation isolates the failure and clearly defines the error states that can arise.

5.3.1 Transactions

Transactions provide a useful method of overcoming the failure of components by the property of atomicity of an action, serializing it with respect to other transactions and causing the changes to all occur or all fail. But any transaction system has costs associated with it. Any action that is to be made atomic must incur a cost in the commit protocol used to implement the atomic property and since the protocol must use some form of permanent storage to record the intended action this expense can be great. In a local area network the rate of failure of components is low so the cost of a transaction system is great compared with the frequency with which it is necessary to activate its recovery procedure. In a filing system efficiency is an important consideration so that the benefits of any facility must be proven if they incur significant cost.

Another feature of a transaction system is that there is a system wide recovery policy built into the transaction system. While this property gives a clean model of working it is not necessarily optimal or correct for all applications. It is the application that defines the sequence of events that occur, and only the application can be considered to know the intended effect. To impose the recovery policy of a transaction mechanism on an application may be undesirable when the recovery mechanism does not make use of some higher level relation between actions that can be used to simplify recovery. The intermediate, temporary files produced by a compiler can be recalculated these and, being temporary files, there is no need to clear these files up in a careful manner as they can simply be deleted, or even just left for reuse. A long running simulation program may wish to produce checkpoint files only at times when it is convenient to do so when, say, the simulation is in some state where the amount of data that needs to be checkpointed is considerably

smaller than at other times, and the simulation itself will restart from the last checkpoint taken.

To some applications, especially those imported from another system, the concept of transactions will not be understood so that, although any filing system action may be atomic, this is of no use to the application as it can not cope with a transaction abort in a graceful manner. There is a large investment in software and in any system it is desirable to be able to import software and run it unchanged. This is a case where the cost incurred by a transaction mechanism brings no benefits. As an example consider a user editing a document in a system that has a transaction mechanism at the level of a filing system action; if a two phase commit protocol is used then an action can take arbitrarily long to commit if the master site fails after it has decided to commit or abort but has not announced this fact. The user is left waiting for the master site to reboot and continue which may take some time. In this case the user may prefer to start again from a checkpoint file produced by the editor, rather than wait, but the intended output file will not be accessible as it is locked by the blocked transaction.

For these reasons it is inappropriate to have a transaction mechanism as part of the filing system. If an application requires the features of transactions then it is able to construct them but if it does not wish to incur the cost associated with such a mechanism then the filing system may be used without transactions.

5.3.2 Location Independence and Location Transparency

In the workstation model a user will most frequently be using the machine in his office. However, local area networks can span physically distant areas, such as two buildings, by the use of optical fibre links so that, as well as wishing to be able to work from his office, a user may also wish to be able to access his files from other machines on the network. In the processor bank model the machine allocated to the user will not necessarily be the same for each session so that here too we must allow files to be accessible from any machine, subject to access control. This mobility of users can be made easier by having the textual names of objects in the filing system independent of the location where the name is used. Location

independence is also important in that it allows application software to access permanent data in the same manner from wherever it is being run. By making text names independent of the source of the request the writing of application software is made easier.

The filing system will need to be able to reconfigure when new equipment is introduced and when failure occurs in a component used to provide the service. We do not wish to tie the location of an object to some particular node in the network because the location is deducible from the text name used to access the object. Location transparency is the property whereby the name supplied to the filing system to identify an object does not indicate the location where the object is stored.

5.3.3 Mutable Files

To support the various models of filing systems to be found in a heterogeneous network it is appropriate to have mutable files. Most operating systems support mutable files and it is simple to use such a system for immutable files by avoiding operations that update a file, rather than write a new one. The converse is not true. If a filing system, providing mutable files, is to be constructed on top of a storage service that only provides immutable files it is extremely difficult to provide the correct semantics in an efficient manner. Suppose a file has two, different names, that the immutable file system has version numbers and that if no version number is supplied then a suitable default is provided. For reading or writing this default is the highest existing version number. Then consider the case where a program writes to the file using one of its possible names. A read operation using the same name will see the changes by letting the default version number be used. However, access using the other name will not because a new version number under this name has not been created. Version numbers apply to the textual naming level; the lower level UIDs are not visible.

5.3.4 Concurrency Control

In a model of working where complete files are read or written in indivisible operations concurrency of a writer and several readers or of several writers is not a problem in that the state of the file is guaranteed to be consistent. This is similar to the case where files are immutable. However, all problems are not eliminated; to decide what changes to make to a file a client may wish to read the contents of this and other files, perform some computation or other extended action such as seeking input from a user, then write a new, updated copy back to the filing system. Should another client attempt to access any of these files then that client must only be allowed to read information, and not to alter one of these file's contents. For, if it did, then the first client may be basing its decision of what changes to make on out of date information.

There are some applications that rely on the characteristics of some particular operating system in the case where more than one program concurrently accesses a file; this makes them very difficult to port to other systems. For example, in the UNIX operating system, where locking is only advisory, it is possible to read a file that is being written and see the new data appear. This requires the programs to be operating on top of the same kernel so that they share kernel resources such as the block cache. This is a feature that can not be transferred to a distributed, heterogeneous environment because of the new failure modes and because of the dependency on the particular semantics of the stand-alone operating system.

5.3.5 File Server Access

Various facilities have been provided by file storage servers, such as control of concurrent access to file server objects, accounting of space used, atomicity of actions and support for constructing a hierarchical name space on top of the server's facilities. In a system that contains many types of client it is to be expected that many types of file server will exist. As the system grows, new types of file storage server will appear which provide a more cost effective solution to providing access to bulk permanent storage than those available when the system was first assembled.

If we are to provide an integrated service for filing in such a system then some facilities of the more sophisticated file storage servers may not be usable, and facilities of simple servers may need to be enhanced to give the required guarantees of security and reliability. Consider a server that provides accounting of its resources; while the information may be useful to a network wide accounting system this can not be assumed to exist for all servers.

In previous distributed systems clients have accessed file servers directly. When there is a homogeneous collection of file servers this is convenient and efficient; when we admit many and varying file servers new complications arise. If we try to avoid modifying the file servers in the network then every client would need to be able to engage in every protocol that the file servers used or accept a restriction on the servers it could access. Further, the access model, with its authorization and access control, would be specific to each type of server.

5.4 A Network Directory Service

To enable sharing to exist between all possible systems on the network while at the same time giving protection against malicious or accidental threats there must be a boundary between the client machine, whose hardware and software can not be assumed to be in any known state, and the filing system. Therefore the filing system can not reside in any client machine where such guarantees can not be obtained. The alternative is to provide system resources to implement the service; as the filing system is central to any computer system it must be reliable and give good performance under load. These features are found in distributed services that use a number of server machines to provide their functionality.

If we are to allow many types of client machine to operate in a single filing system we must ensure that each client is not required to trust every other to maintain the filing system's internal structure. There must be a controlling authority outside of any client that ensures that the integrity of the filing system is maintained. A single directory service provides an interface that maps requests from any of the client machines to actions in the filing system, ensuring the arguments

passed are correct and that the client has the authority to request the operation. Further, the service conceals the location of files on file storage servers from the clients, so that it is possible to reconfigure without disrupting the clients, and each client is simplified. A directory service centralizes the functionality of the filing system so that every file storage server does not need to be able to understand every other file storage server's interface. Replication of this service is important to improve reliability and to increase efficiency. A single such service operating on the network ensures that sharing between systems is possible since there is one filing system for all operating systems.

5.5 Client Model of File Operations

The principle technique to ensure that operations on a file perform in the same manner as they would on a stand-alone system is to transfer a complete file on which to perform operations. The file can then be viewed in the normal way according to the operating system's local model and applications will see the same semantics and file structure as in the stand-alone case. Another advantage of this method is that the total work done by the filing system in transferring the file is reduced in the case of the whole file being accessed.

5.5.1 File Access

There are now three parties involved in accessing a file; the user machines, who are clients of the filing system, the file storage service that provides permanent storage, and the directory service which exists to

- give the client the view that there is one global filing system
- provide uniform access control
- control concurrent access.

If a client had to deal directly with file servers these properties would be lost or would not be uniform. The solution is for client requests to go to the directory

service where operations such as path name lookup are performed, but for the reply, if it is in the form of data from a file, to come from a file server to avoid the data being copied through the directory service.

Consider a client opening a file on behalf of an application program. The sequence of operations could be that the client sends a request to the directory service to lookup the file name; because it controls access the directory service must contact the appropriate file storage server to inform it that the client may retrieve the file at the present time. The directory service could return some temporary authority to the client who then contacts the file server. This has four messages on the critical path; a request/reply to the directory service and a request/reply to the file server to fetch the file.

The length of the critical path can be reduced; a three way protocol can be used which proceeds as follows; the client contacts the directory service to request the file contents, supplying some port for the data to be sent to. The client is willing to receive data on this port from any file server. The size of the port name space and the short length of time the port is waiting give some assurance that the data path is secure against accidental interference. The directory service looks up the path name to yield a UID and then determines which file server the contents are currently stored on. It transmits a message to that server to tell it to send the data to the client. The file server then sends the data as instructed. See figure 5.1.

This leaves three messages on the critical path; client to directory service containing the request, directory service to file server, instructing the file to be sent, and file server to client containing the data. This latter message may consist of a number of network packets. Replies to these messages are also sent to confirm success or transmit an error status but these are not on the critical path when the action is successful. They allow for more information to be returned as to the nature of any error other than just a timeout.

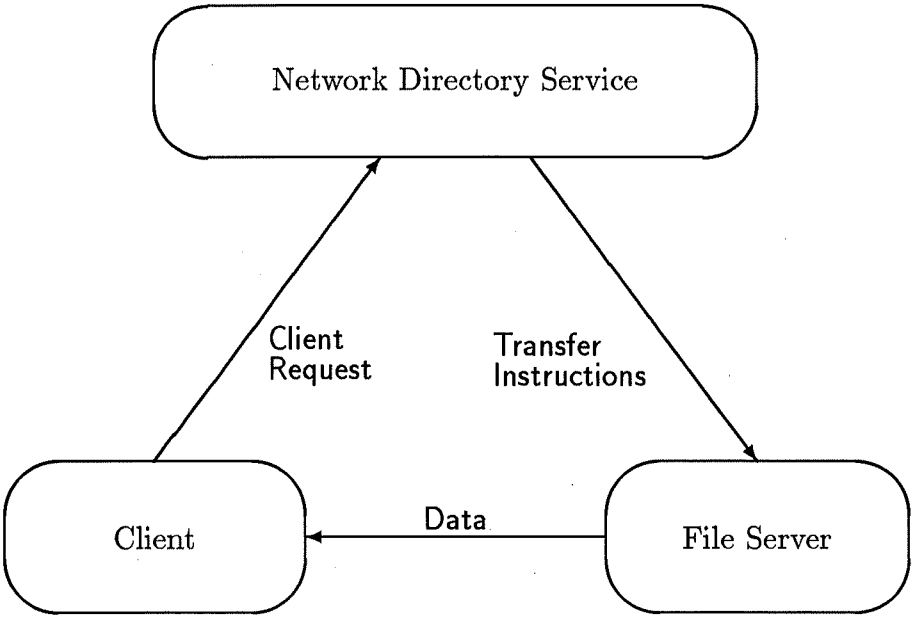

Figure 5.1: Critical Path for Fetching Data

5.5.2 Reading

When reference is first made to a file the operating system will issue a request to transfer the file from the filing system to the local machine. If it has local disc then the file can be placed there, enabling caching of the file for later accesses. A more sophisticated implementation could implement the above model of operations involved in accessing a file for the first time but use an optimization to allow the application to start to process the data as soon as the first part of the file arrives. The latency incurred by the application involved in touching a file for the first time is then reduced. Much file processing consists of sequential reading of data so that the improvement noted above is useful in the most common case of file access.

5.5.3 Writing

When a file is written the data is initially placed on local storage and then transferred to the file server when writing is complete and the file is closed. This can be used to remove the file transfer time from the applications critical path, especially if the client machine has local disc, for the contents of the file do not

need to be sent when the file is closed if there is a locally stored, permanent copy. The operating system can attempt to send the file repeatedly if it loses contact temporarily with the filing system without causing the application to hang. In the case of serious failure of the filing system, causing it to become unavailable for longer periods of time, recovery may need human intervention to decide whether to continue without network filing facilities, or to wait until the directory service has been restored.

For a file that is to be updated, rather than created and written, the contents are transferred as in the case of reading a file, and returned as in the case of writing a file.

5.5.4 Client Caching

Caching in memory of information from disc has been shown to be beneficial in stand-alone systems for increasing the performance of the filing systems [McKusick84]. In the case of distributed systems there are also large gains to be made in caching files [Schroeder85] [Richardson84]. It is important to allow clients to make use of their local resources to enable caching to be performed. In a system with whole file transfer a copy can be retained after it has been closed and written to remote storage in case it is accessed again. Subsequent access will require validation of the cached copy but it will be shown that this only requires communication with the directory service, and not with any file storage servers.

There is an important class of files where caching is especially beneficial. These are the files that contain commands invoked by the user. The pattern of use of these files is heavily orientated towards reading rather than writing; they are rarely changed and users may not be concerned if they do not have an up-to-date version if, for example, a change means a performance improvement or an extension to the facilities provided. In these circumstances the old version will continue to work correctly. Therefore, it may be acceptable to cache these files and use them without checking that they are up-to-date. The introduction of new versions of system supplied commands would then require human intervention but the frequency of such events is such that the increase in performance outweighs

the extra work for administrators.

5.5.5 Exclusion Locks

The design provides one kind of lock, a token which is an exclusive lock that stops other clients from writing the file. The holder of a token is guaranteed that no other valid token is associated with that file so that to process a file with locking the client sends a request to the directory service to issue a token specifying the file, reads the file, performs any desired computation, then writes the file back. If it wishes to freeze the state of other files then it can take tokens out on them as well; this will only be needed by applications making use of the distributed environment because imported applications which look at several files rely on the operating system on which they run.

To perform the action of opening a file for writing the operating system can request a token from the directory service, although it could be done at a later time, such as at file close. However, obtaining the token at file open time enables errors to be passed back to the application if, for instance, the path name specified is invalid, and the operating system wishes to pass this condition back to the application. Should the operating system use asynchronous write-back then obtaining the token just before sending the file to a file server does not allow it to tell an application if an error occurs. Even if write-back is synchronous then the work that has to be performed to create the file will now be thrown away.

5.5.6 Cache Servers

The use of whole file transfer puts a requirement on clients of the filing system to be able to hold a copy of a file on behalf of the application that reads or writes it. For machines which have a local disc such space is available but, for economic reasons, it may be desirable to have discless machines on the network and these do not have the facilities to provide the space needed for file storage.

The solution is to incorporate these discless machines in a sub-system which presents itself to the filing system as having sufficient resources to handle the whole

file method of operation. The additional facilities are provided by a service which allows access to disc space which is shared among those clients which do not have local permanent storage; sharing of this cache service still enables the economic criteria that lead to discless machines to be met.

Clients which need to handle files in small sections, because of memory constraints, will generate many small file access requests rather than a few large requests. It has been found that increasing the size, and hence reducing the number, of file server accesses can improve performance [Richardson84] by moving work away from the bottleneck of the file server. Network protocol work can be a significant proportion of the time taken by a data transfer where the transfer size is small, leading to the CPU in the file server being the bottleneck [Lazowska86]. An advantage of the cache server is that small clients do not send their requests to a file server; instead the cache server handles these and itself participates in whole file transfer to and from the network filing system. By making each cache server specific to the operating system that it serves the interface can be tailored to that operating system, reducing the size and complexity of the code in the client and the work it needs to perform.

5.6 Data Transfer

The problem of access to servers to fetch and store data is complicated by the wide range of access and transfer protocols used. One approach is to rely on the clients to implement each protocol they might need. However, this is unacceptable because of the large amount of work needed, $O(n * m)$, where n is the number of types of client and m is the number of types of file server, for each of the n clients will need m protocols implemented. A similar amount of work will be needed if each file server is modified to use a protocol understood by each client.

The alternative of a network model of file server access will mean that each client need only implement one standard protocol; work will still need to be done on servers to introduce them into the network but now there is no need to modify every client for each new file server. The total amount of work is now $O(n+m)$, as

each file server and each client need only have one protocol implemented. The lack of disruption to existing servers and clients upon the introduction of new clients or servers make this scheme even more attractive. Changes to file servers are necessitated by the use of the three way access protocol so requiring the transfer protocols to be changed in each file server does not generate additional software that must be modified.

5.7 Summary of Design Features

- By transferring complete files, rather than having the client operating systems manipulate files in place on the file server, it is possible to have the local semantics of the operating system for access to that file.

- A network standard for the representation of data in the filing system is not enforced to avoid overheads of translation where these are not necessary; sharing is most common between machines or users running on the same operating system and in this situation translation of the data is not needed. File storage servers provide a single file abstraction that can store the bits in the file and applications are responsible for correct interpretation of the data.

- The naming conventions of operating systems differ. To give compatibility for existing applications it is necessary to allow any characters to appear in a file name. By using a sequence of strings, rather than a string with separators, as the representation of a file name in the interface to the directory service no restrictions are placed on the client operating systems. Applications which wish to allow for the possibility of data being shared by other systems will have to restrict the characters used in a file name.

- The kernel of a client operating system should provide a compatible interface to that which would be seen if accessing the local filing system so that applications can be run unchanged. In addition, the interface should be extended to allow access to the network filing system interface for the use of applications designed to run in this environment.

- The functions of file naming and directory manipulation are separated from the storage of the information. A single directory service provides the operations for client operating systems to use to access files in the network filing system. For efficiency and for reliability this service should be replicated.

- Locking is required to allow a client or application to freeze the state of a file while it is being written or while a calculation is being made on the basis of the file's contents. Locking is not needed to avoid inconsistency within one file because the whole file is read and written as a single, indivisible operation.

- To read a file a client contacts the directory service to resolve the path name when the file is opened; the file is then sent from the file server to the client. To write a file the client optionally obtains a lock on the file when it is opened. When writing has finished the file is sent direct to the file server.

6 A Prototype Directory Service

This chapter looks at an implementation of the ideas that have been discussed. Of necessity the implementation is influenced by the environment in which it is to operate; however, the general principles it employs are applicable to a wide variety of network architectures.

6.1 Environment

6.1.1 The Cambridge Model Distributed System

The Cambridge Model Distributed System (CMDS) [Needham82] replaces the concept of a central mainframe with a series of computers attached to a local area network. It uses a processor bank model, that is, machines are allocated on request when a user has some computational need, rather than permanently providing hardware to users.

6.1.2 The Network

The CMDS is built around a number of Cambridge rings [Wilkes79b] connected by bridges [Leslie83]. From the point of view of an application accessing the network, the fundamental unit of data transfer is the basic block. It allows for the transmission of between 2 and 2048 bytes of data from one ring station to another.

The observed error rate is approximately 1 bit in 10^{11}, which is about 1 bit in every 10 gigabytes. The maximum bandwidth from one station to another varies with the size of the ring but is about 800 kilobits/second for a 3 slot ring; it is a feature of the ring that there is a guarantee of a station obtaining some bandwidth at all times.

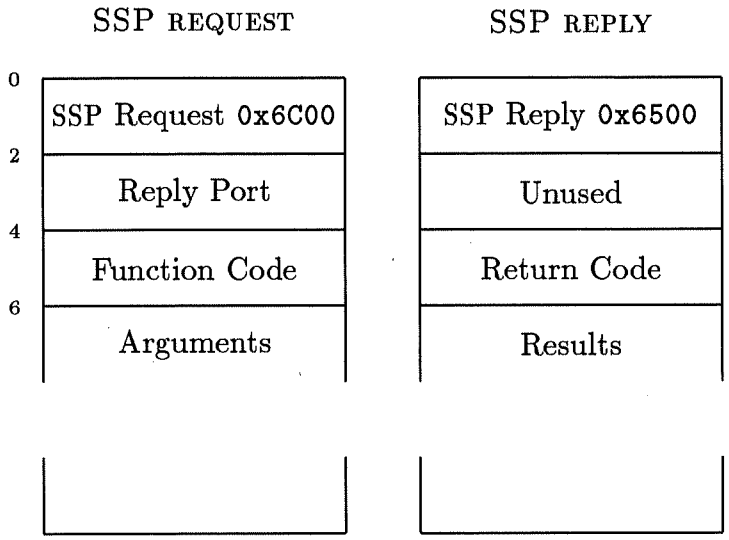

Figure 6.1: Basic Block Formats for the Single Shot Protocol

Protocols

On top of the basic block protocol there are a number of other protocols. The Single Shot Protocol (SSP) [Ody79] is commonly used for interaction with ring services. In this, a single basic block is sent to the service. The block contains a reply port, to which the server should direct the results, and a number of arguments. The first argument is a 16 bit number giving the function code to demultiplex various types of request that might arrive on a particular port; other arguments can be in a format decided by the service. The reply is another basic block with an unused 16 bit word and a return code followed by a number of results of any format.

To establish a virtual circuit there is the OPEN protocol which is used to establish a block stream between two applications. An OPEN block is a basic block, with a port number for the reply, a function code and a number of application dependent arguments. The receiving application sends an OPENACK block back that contains a return code, port number for the block stream and a number of results from the application.

Other protocols exist, such as a byte stream protocol (BSP) [Johnson80], which

offers flow control, acknowledgements and error recovery built on top of a block stream, and two RPC protocols (see section 6.1.5).

Extended Ring Networks

Bridges are not able to transparently copy the low level mini-packet, which are used to construct basic blocks, from one ring to another because there is a limit in the architecture of the ring to 256 addresses. There is a nameserver on each ring at a well-known address. When a name is looked up by some machine wishing to send a basic block, the nameserver entry may indicate that the required service is served by a machine on another ring. Suppose that it is on an adjacent ring, that is, there is just one bridge involved. The nameserver instructs the bridge that a route is to be set up and the bridge allocates one port on the senders ring for the outward basic block, and one port on the receivers ring for the reply. The sender is supplied with the ring station number of the bridge and the port allocated on its ring as the results of the nameserver lookup.

The contents of the basic block sent from the sender to the bridge port on that ring are copied onto the receiver's ring, except that the reply port is altered to be the port allocated by the bridge on the receiver's ring. The bridge inspects the header of the basic block to see which protocol is being used to enable it to determine what action to take. When a reply is sent the bridge copies the block to the original sender's ring; if it is a reply to an OPEN then the port number in the block is altered to that of a bridge port allocated for the rest of the conversation. Except for this last port, deallocation occurs after first use. All ports are deallocated if no block is received on them for some timeout period.

A similar process is involved if the receiver is more than one bridge away. Bridges will set up routes through to the next bridge until the correct ring is reached.

6.1.3 The Processor Bank

The processor bank contains a number of types of machines ranging in size and power. The principle machines are Motorola 68000 systems. Allocation of

a machine to a user or another machine is achieved by sending a request to the resource manager [Craft85] for a system to be loaded into a machine. The request includes the properties desired, such as the amount of memory or presence of special devices, and a machine that has at least these attributes is allocated and loaded with system image. Once the machine is allocated there is no guarantee as to the contents of the machines because the user is free to change the system that is running and because there is no secure memory in the machines.

6.1.4 The Mayflower Operating System

One of the operating systems that can be loaded is Mayflower [Hamilton84]. It was designed primarily to support research into RPC mechanisms and has subsequently been used to construct some servers using a high level language with support for concurrency and RPC. Mayflower provides a simple kernel that controls memory allocation, access to devices, such as the ring interface, and support for a number of application programs each running in a domain. Each domain can have multiple processes with synchronization within a domain based on the use of shared memory.

6.1.5 Concurrent CLU and Remote Procedure Call

Mayflower is a preferred language system; that is, while it is possible to write programs in any language, the interface to the kernel is based around the needs of one language. The preferred language is CLU [Liskov81], which has been extended [Hamilton84] to include lightweight processes running in a shared address space and a language level RPC mechanism.

The remote procedure call mechanism provides two styles of call; the *exactly-once* protocol, which ensures calls are executed precisely once in the absence of machine failures or long term communication failures, and the *maybe* protocol, which passes low level error conditions back to the caller. Such conditions might be a timeout of a response packet or congestion of the receiver's RPC mechanism. These soft errors are handled by the *exactly-once* protocol but unrecoverable hard errors, such as call binding errors, are still passed back to the calling code as CLU

exceptions. It is not a transparent RPC mechanism and there is a different syntax for local procedure invocation and remote procedure invocation.

It was found to be more appropriate to use the *maybe* protocol in the construction of the directory service because of the additional control over error detection and recovery. The *exactly-once* mechanism does not detect the failure of servers sufficiently quickly, since it could not distinguish a crashed directory server from one that was temporarily congested, except with a large number of retries over an extended period of time. The nature of the application, with the detailed knowledge about the remote procedure being called, meant that a better detection algorithm could be used.

By basing communication between servers on an RPC mechanism a degree of network independence is achieved. Many low level details are masked, leaving only more general characteristics, such as occurrences of timeouts due to the callee failing, rather than transient errors of the ring.

6.2 Data Structures

The most important task of the directory service is to map textual names to UIDs. Several methods are available. In CFS the full text name is looked up in a B-tree [Comer79]; lookup consists of searching for the name and retrieving the UID held there. However, in the directory service we wish to have data spread over many servers, with a server not needing to hold all information relating to the name space. Therefore some way to divide the name space between servers is needed.

6.2.1 Directories

The scheme adopted was chosen to be close to the user's model of the directory graph by using the concepts of directory and file. By using a directory, a concept used by the clients of the service, it is possible to make use of the locality of reference in names looked up. It is to be expected that a user or application will

name objects close together in the sense they are in the same directory or share a common parent. As will be seen, after one access to a directory that directory is available for further accesses without entailing any further network action.

A directory has times for its creation and last update and also a list of directory entries, which are pairs of component name and link to the named object. The link contains the access control information and a discriminated union which gives the type, one of file, directory or soft link, together with the UID in the case of a file or directory, or the name in the case of a soft link. The access control information should not reside with the object named because there might be two links to the same file or directory for which a user wishes to have different access controls. Suppose that access control is held as part of a file or directory in order to remove the need for links. Checking of access permission would require knowledge of the directory and component being used to reach the object. A change of component name would need a change both to the directory and to the object itself which are held independently on permanent storage. Thus, a multiple object atomic update must be carried out. The need for this is removed if links are held with the directory entry; only the directory used to name the object needs to be updated for a change in component name.

If access control is associated with links there arises the problem of controlling access to the root of the directory graph. Since the root is well-known there is no link to be followed to access it and so no access control can be applied. This situation does not arise in filing systems which have a tree structure because access information can be held with the object, and not as part of the directory entry. The solution adopted is to have a root to which anyone could have read access and which never required updating. The root contains just one entry to a directory which, to the user, appears to be the actual naming root of the filing system. This may be thought of as starting the path name lookup on a well-known link; by placing that link in a directory the replication management of the well-known link is unified with the management of other directories. The directory viewed by a client as the naming root will be referred to as the root and the actual, read-only root will be the meta-root. A benefit of this scheme is that the UID of the

meta-root can be well-known to the start up code that the servers of the directory service execute to initialized themselves and it provides a solution which does not require special consideration for access to the naming root.

6.2.2 Files

At an abstract level a file is an object with a number of attributes, the most important of which is the data associated with that file. The management of these attributes is divided between the directory service, for items such as timestamps and size, and the client, for the data contained in the file. The file servers give a permanent representation of these attributes, but without any responsibility for the consistency or accuracy of the information contained.

A file is split into two parts, a file header, which contains the directory service information and the UID of the contents, and the data, for the client managed attribute. This separation of a file into two parts allows the directory service to manage file headers in a similar manner to directories, using the same replication scheme and update policy. Clients can be given access to all the data part which can conveniently be held in a file server file avoiding the need for a hidden header in the file server. A potential disadvantage of this scheme is the number of objects that must be read in order to access a file. Caching of file headers reduces the overheads of access.

6.3 Replication of Directory Service Information

Replication of data in the directory service is essential to give sufficient performance and to give this important service increased reliability. The data items replicated and managed by the directory service are directories and file headers.

6.3.1 The Cache

The scheme used is based on caching read-only copies of objects in directory servers and having a primary site on some server controlling all updates. As will be

seen the primary site is the first to reference the object and that reference causes the only read from disc. A distributed locking scheme (section 6.4) ensures that all subsequent references from other servers cause a copy to be transferred from the primary site to the new cache site. This is based on a number of assumptions :–

- The frequency of update is much lower than that of lookup, particularly for directories that are widely shared.
- Efficiency is an important criterion.
- The rate of failure of the network and of the hardware used for the directory service is low.

The central role of the directory service in the distributed computing environment means that it is important to have an efficient implementation. In deciding which one of several schemes to use in a particular situation it is acceptable to increase complexity of coding and maintenance if this leads to better performance.

As the expected rate of failure of servers is low it is desirable for the cache management scheme to have low overhead in the normal case of servers being accessible as and when they are needed. To perform an update operation it is assumed that the primary site is available to perform coordination of the change until contact with that site fails. At this point the operation enters a recovery phase to cater for the loss of that server.

A directory which is closer to a root of the naming graph will have a higher ratio of read operations to write operations. If a directory is accessed by many of the directory service's clients then it is likely to be closer to the root of the naming graph. A change to such a directory is seen by many clients so that there is less disruption to the name space structure if these directories are kept from frequent change. This is apart from efficiency considerations.

Therefore, it is desirable to replicate most widely those directories that are most heavily used and to choose a scheme that has a low cost for read operations, performing cache management on update operations. If, for a particular directory,

Operation	Number of Messages Sent Between Servers	
	Invalidate On Change	Validate On Lookup
Lookup	0	1
Change	$1+(N-1)$	1
Mean Cost	$P(Change).N$	$P(Change)+P(Lookup)=1$

where N is the number of sites with cached copies.

Figure 6.2: Costs of Cache Management Schemes

$P(Change)$ is the probability of an operation being a request to make a change and $P(Lookup)$ is the probability that the request is to read the object then the least cost scheme (see figure 6.2) is invalidate on change if

$$P(Change) < 1/(Number\ of\ Cached\ Copies)$$

This suggests an invalidate-on-change scheme, where a cached copy is always up to date, since this criterion will be met for most directories. If a write operation is performed it must occur at the primary site which invalidates all other cache entries. A new copy is not sent to all the sites in order to reduce the latency of the write operations and to allow for rebalancing of caches. If the directory is read again by a cache site then a new copy will be fetched; if not, then the memory space may be used to cache some other object.

The expected low rate of failure of servers and of the network means that any replication scheme should aim to minimize costs that occur on every operation, even if this increases the cost of recovery. A lightweight detection scheme means that an operation is as cheap as possible in the normal case of no failures occurring during its execution.

6.3.2 Grain of Replication

When considering the cache we need to consider three units:

- The unit of transfer: this is the size of the items that are exchanged between servers when a site wishes to copy an object into the cache from another

server.

- The unit of storage: objects are stored on file servers to give a representation that persists across partial or total failure of the directory service.

- The unit of replication: this is the size of object that can be addressed in the cache. The unit of transfer will be a multiple of this since a fraction of a replication object transferred could not be stored in the cache.

The unit of replication is a filing system object, directory or file header, and, because clients operate on this, it is convenient to also make it the unit of storage. Making the unit of storage smaller would mean that to read an object a number of file server operations would need to be performed, rather than sending one request across the network. Writing objects would also need a number of file server requests and these would need to be performed atomically. A unit of storage that is larger than that of replication would result in several objects needing to be written, and hence unavailable for other updates, when one is changed. As we are allowing an arbitrary graph structure it would be difficult to choose the grouping of objects into the unit of storage.

The scheme employed performs transfers between servers much more frequently than between a server and permanent storage. This is advantageous because accessing permanent storage across the network is much slower than accessing the memory copy of a cooperating directory server.

The unit of transfer must be a multiple of the replication unit. However, it may be more efficient to transfer several cache objects when only one is requested in the hope that later accesses to the requesting server's cache will be successful in finding the appropriate object because it was copied in as part of some earlier transfer operation. Transfers of large numbers of filing system objects will cause the total transfer time to increase, delaying the client request that caused the transfer to happen. This effect will be important if it is expensive to determine which objects to place in the transfer. Noting that locality of reference is to be found in objects named by clients, transferring the objects named by components of a directory may help to increase the cache hit rate.

There is no guarantee that these child objects are in the cache of the server holding the parent directory, or even that they have been read in from disc. Transferring only candidate objects already in the cache of the primary site server was rejected in the prototype because the cache hit rate of server will be high due to the large number of read operations, as compared with update operations. Under a scheme of just transferring the object requested the additional delay caused by checking the cache of the primary site for the presence or absence of candidates was not justified.

6.3.3 Object Management

A primary site update scheme is used with each server devoting the majority of its free memory space to a cache of directories, file headers and information relating to uncached objects. The high cost of reading from a file server, especially as caching there will not be beneficial if there is caching in the directory service, means that every effort is made to obtain a copy of a directory from another site rather than access disc. A cache slot is keyed by UID and can be in one of four states:

1. *Primary*, where this server holds the primary copy for this object.

2. *Cached*, where this server has a cache copy of the object but it is not the primary site for it.

3. *Known primary site*, where the server has a record of where the primary site is but currently has no copy. This state occurs for any object which has been read by some server but no copy exists at this site.

4. *On disc*, where the object only resides on permanent storage and there is no primary site. This state of a cache entry is indicted by the absence of an entry; it is not confused with a non-existent object because a UID for the object has been found.

When an object is referenced which does not currently have a slot in the cache then an election (as described in 6.4) is held to ensure that two sites do not

simultaneously attempt to become the primary site. The winner of the election may then read the object from the file server and it announces the successful completion of the file server operation during the release phase of the election. Every site marks its cache as *known primary site* except the new primary site. Thus, all sites will know where the primary site for any object is, if that object has been read from disc, in the absence of failures.

If, however, a site didn't know the primary site for an object, which is held at some server, then, when it calls an election for the object, the attempt to take out a distributed lock fails with the return indicating where the primary site is. This is used when a new server is brought into the service. Instead of attempting to get its cache into some consistent state before starting it is allowed to operate with an empty cache. This is filled up as it calls elections on objects already active.

When an object is referenced for a read operation by a client at some server the cache is checked to see if this site has a copy or is the primary site. If either of these occur, then the reference can be satisfied immediately, otherwise if the cache slot is marked *known primary site* a call is made to the primary site to get a cached copy. Should the cache have no entry for this object an election is called to obtain a distributed lock to read the object from disc.

Operations that cause an object to change must be performed at the primary site. When a request is sent by a client to a server that is the primary site the object is updated, written back to disc and all other cache sites informed that they should invalidate their caches with respect to this object. At this point a reply is sent to the client. If the request is sent to a site having a cached copy then the request is forwarded to the primary site where the operation is performed as before. The cache site also updates its cached copy in the anticipation of subsequent accesses by the client who caused the update.

6.3.4 File Server Support

This cache strategy minimizes the number of file server operations that need to be performed. A directory is only read from disc when there is no currently

available primary site, either because the object has never been referenced or because the primary site has failed. In a distributed system file server operations are often much slower than the equivalent disc operations in a stand-alone system and they can become bottlenecks. By avoiding the need to go to file server the directory service is not impeded in this manner.

An ideal file server to support the directory service would be designed to perform operations with as little latency as possible. There will be more write requests than read requests so caching at the file server will not be beneficial; the directory service is already doing the necessary caching.

The size of objects is small, comparable to the size of a disc block, so attempting to read ahead significant amounts of data will rarely increase the throughput of the file server. Only caching of data relating to the location of disc objects would bring worthwhile improvement. The reading and writing of objects contained in one disc block with as little delay, preferably atomically by writing the object to disc then changing the information indicating its location, will dominate the pattern of requests sent.

6.4 Elections

In holding an election we wish to efficiently and reliably nominate just one site to perform an action, in this case read an object from disc and become the primary site. This synchronization between the servers is achieved by acquiring a distributed lock on the object before performing the operation, with the address of the primary site being passed to all other servers during the release of the lock. An election provides the method of ensuring mutual exclusion on the operation of becoming the primary site.

Unlike locking in a system with shared memory, distributed locking is costly because of the communication between the sites involved, and hence in the delay before the critical region can be entered. Therefore a method is needed that requires few messages in the case of no contention on the lock, that is, no other

server is currently in the region and no other server is attempting to acquire the lock. It is required that servers obey the convention that the lock is obtained before becoming the primary site but this can be ensured since the servers trust one another as to the code they are running.

6.4.1 Algorithm

Each server is expected to know the network address of every other server currently providing the service from the resource location methods used in the network. There are two phases to the acquisition of a lock. In the first each server, including the one holding the election, is polled in the order defined by some ordering on network addresses. Under the current implementation this is achieved by polling in increasing ring station number. There are two possible replies a server can make to a poll request; it can agree after adding the UID to its list of currently locked objects or it can reject the lock request. It is not enough to know where the primary site is to reject a lock request since a failure of the primary site may have caused the election to occur and another server may be attempting to recover by reading the object in from file server.

If a request is made to a server that has already voted for some other site to become the primary site, indicated by the UID being on the list of locked objects, then the call is blocked until the first caller releases the lock. The request is then rejected and the name of the server that releases the lock returned; this is the server that has become the primary site. This strategy will not lead to a deadlock because the servers are contacted in a predefined order; if two elections start at the same time then one will block at the first server they have in common on their list of known servers. The assumption of no network partitions and a slow rate of failure means that the lists of servers in any machines calling an election will have some servers in common.

To allow for failure of servers during the election a call to a server will timeout if a reply is not sent. The choice of the length of the time allowed must allow for a call that blocks because some other server has called an election on the object at the same time. Should a server attempt to call an election when there is already a

primary site, such as might occur after a timeout, then the election will fail when the current primary site is contacted.

After an initial period the frequency of elections will become low as many directories are cached from disc into the directory service.

6.4.2 Multicast

The Cambridge ring has no facilities for broadcast or multicast. Should a network have either of these features then alternative election algorithms may be possible. The scheme outlined uses the order of contact to ensure that a second election for the same UID will be blocked early at some server. With multicast the order of contact is undefined but obtaining a majority of the votes cast by servers would guarantee mutual exclusion. In the rare case of three or more servers all calling an election and none obtaining a majority backing off for a short, random length of time and retrying would avoid deadlock.

6.5 Failure Tolerance

An important feature of distributed systems is their ability to maintain a service provided by a group of machines in the presence of network failure or node failure.

It is not necessary to have a general recovery scheme that can recover any operation or sequence of operations because there are only a small, fixed number of operations, which occur in short sequences, that cross server boundaries. The additional overhead of such a general scheme does not bring any benefits. A scheme that is specific to an operation being carried out is more appropriate because it can be more efficient by using semantic knowledge of the operation being performed. The added complexity of the operations and increased difficulty in programming with no general recovery support are less important considerations because the directory service is a system facility that is heavily used.

6.5.1 Node Failure

A fail/stop model with independence of failures of nodes on the local area network is used. That is, a crash of one node does not change the probability of failure in any other node and a node does not produce any effects observable to other nodes by, say, transmitting corrupt data, before the processor stops. A node is seen to be functioning correctly or to be stopped.

While these assumptions provide a reasonable model for hardware the assumption of independence of failure for software must be more carefully examined. Should a piece of software fail due to some sequence of events then at least some of the sequence is likely to be retried at another server. If a request from a single client was sufficient to generate the failure sequence then the second server will fail. Such an error is easy to reproduce and hence to correct; software errors which occur because of the interaction of more than one client request may not be so simple to identify because the individual requests may complete successfully under different circumstances. When failure sequences need to be longer and require several client requests there will be a lower correlation between any two failures due to software to the point where independence provides a sufficiently good model.

If we extend the model to allow processors to generate incorrect messages and send them across the network then variations of Byzantine agreements [Pease80] [Lamport82] [Chor85] are necessary to ensure that messages are correct and that faulty processors are isolated. The additional costs of such techniques is high and the frequency of such complex failures low so that the decrease in performance is not balanced by the increased reliability.

6.5.2 Network Failures

There are a number of classes of communication errors that can occur in local area networks; each class is detectable and may be coped with different ways.

At the level of the unit of transfer across the network there are bit and burst errors where one or more bits of the unit are changed or lost. By the use of parity checks or cyclic redundancy checks it is possible to provide an interface where a

block of data is either successfully received, with the contents being correct, or it is rejected, with the contents not being passed to higher level protocols.

Block errors include loss and duplication of blocks although the contents are correct. These errors are limited in single networks but once we allow for several networks to be connected their frequency increases.

Both these classes of error are handled where necessary by RPC mechanisms. RPC provides a medium for communication that gives correct transmission of an arbitrary amount of data to the receiving machine or an error indication to the sending machine.

The last class of error that occurs due to the network is partition. In this the network becomes two or more separate networks where each node in a partition component can communicate with every other node in the component but not with any of the nodes in different partition components.

Partitions

Partitions cause serious problems when they occur; resources that are to be shared are no longer accessible to the whole user community and replicated data is difficult to maintain. Changes made in one component can not be propagated to another component so that decisions made may be based on out of date information. After the fault causing the partition has cleared then there is a problem of resolving updates of data, which were in different components, if these were permitted during partition.

The requirements of a scheme which copes with the possible occurrence of partitions of the network must be viewed in the light of their effect on the performance of the system. There is a balance between performance and availability. A partition is a rare event so that any costs needed to be applied at all times must be kept low. A partition may be transient or it may persist for a longer period of time, of the order of days. There may be a need for human intervention to deal with the problem, as in the case of an internetwork bridge failing due to a hardware fault.

If long term partitions can occur it is impossible to give the user a view of an error free network. Therefore, an application which makes use of the distributed features of the network will need to be aware of the possibility and existence of this mode of failure to take any measures required, appropriate to its use of distribution of the data.

Without allowing for the occurrence of partitions the filing system can not safely provide any level of service. The infrequency of partition does mean that the level of such a service can be low in terms of performance, compared to when partitioning is absent, and may not allow some clients to perform some operations. This can not be avoided; if a client wishes to read a file which currently only resides on a file server in another partition component then there is nothing that can be done except either wait for the restoration of communication with the file server or abandon the operation. Abandoning the operation is more appropriate if the partition is expected to last for some time but waiting and retrying is a more useful policy if it is a short term occurrence. The level of service may not be sufficient for some clients; even when operations can be performed the response of the directory service may be reduced because of an increase in clients using servers in a component when those clients would have directed their requests to servers which can not be contacted at present.

There are several styles of policy that the directory service could adopt when a partition occurs.

1. Stop all changes to objects in the filing system. This effectively stops any useful work being done since the filing system becomes read only.

2. Allow the existence of a primary site in each partition component. This allows the most work to be done while the partition exists but there is the possibility of inconsistences arising from updates in two components.

3. Allow update operations in the component containing the primary site before the partition but only read operations in other components able to access a copy. There is still a possibility of decisions being made on out of date

copies of data but there is no possibility of conflicts on restoration of communications.

4. Allow read or update operations only in the component containing the primary site. The difficulty here is the detection of the partition's existence because it could occur at any time. It would be necessary to check that no partition existed on every read operation. The caching algorithm has attempted to avoid any contact with the primary site during read operations for efficiency.

5. Weighted voting algorithms. These increase the availability of the filing system but at the cost of increased overheads on every operation, whether a partition is in existence or not.

The third scheme allows some form of working while a partition exists and avoids increasing the overheads of operations at all times. The others require checks to be made, even on read operations; a partition can only be noticed at the point where a remote procedure call is attempted from one server to another. The prototype directory service does not cope with the existence of partitions.

Detection of Partitions

One of the difficulties that any distributed system encounters is deciding when a partition exists. There is a difference in the actions that should be taken between an occurrence of a partition of a group of nodes and the complete failure of the same group due to, say, a power failure. This is because the group of nodes in question may continue to provide a service in its component of the partition in the first case but in the second it will not. The first time that nodes which are not part of the the group will be able to differentiate the two cases is when contact is restored; however they may need to decide before that how to recover from the fault.

There is no test that can be applied to separate these two situations so the alternative is to have an empirical test based on the known and expected failure characteristics in the network and of the servers. By assuming independence of

node failure and its low probability we may assume, say, that if two or more nodes become uncontactable then partition recovery should be used, if it is only one node then crash recovery is started. There is a very low probability that two nodes will fail in an indistinguishable interval of time so that it may be regarded as less likely than the existence of a partition. A human administrator may, at a later time, instruct the directory service that what was originally classified as a partition should now be regarded as failure of those machines. This will be necessary because a long term partition could have existed, followed by failure of nodes in a partition component at a later time. It is not possible to reverse the decision that a group of nodes had failed when, in fact, they formed a partition component.

6.6 Concurrency and Locking

It is necessary to have a token in order to write the file it is associated with. Once a token is granted by the directory service there is no need for the client to contact the service again; it can go straight to the appropriate file server. The file server is supplied with the token when the token is issued by the directory service. There is a request and reply from the client to the directory service during which a request and reply is made to the appropriate file server.

By including a sufficiently long random number as part of the token it is possible for the file server to check this against the number it would expect a token to contain. If they agree then the write-back request can be allowed to continue. This technique is particularly suitable for authenticating tokens because their lifetime in the system is not long. A random number of 31 bits, which is never zero, is currently used. If a client wishes to freeze a file, as in section 5.3.4, but does not have write permission on it then the directory service invalidates the authenticating random number by setting it to zero.

The file server must also provide the directory service with an operation to invalidate a token. It is the responsibility of the directory service to notify file servers if a client fails.

An alternative to the four way exchange of messages to issue a token is to use a three way exchange similar to that used to read files from the file server. However, there is no gain in this situation. It was necessary for the file server to contact the client to arrange the transfer of the file because it is unacceptable to route the file's contents through the directory service. When issuing a token the client must still wait for the directory service since the directory service manages tokens. Therefore, no reply can be sent to the client until the file server has replied to the directory service and, at that time, the directory service, as well as the file server, is in a position to send the token to the client.

6.7 An Example

As an illustration of the methods employed in a directory server to perform operations in a fail-safe manner, without employing a general transaction mechanism, consider the creation of a new directory. Write (a,b,c) for a sequence of components naming the object to be created, so (a,b) is the parent directory. Suppose that the client making the request sends it to a server that is not the primary site for (a,b) and that the parent already exists. Only one object can be created in an operation; it is an error if (a,b) does not exist. Figure 6.3 outlines the sequence of events.

The process of creation consists of five stages; checking the request, creating the new object, attaching it to the naming graph, invalidating the caches of other servers and updating the initiating server's cache copy. The first, second and last occur at the server (C) contacted, the others at the primary site (P) for (a,b). For simplicity it is assumed that the objects (a) and (a,b) are cached; had they not been then name lookup would result in a copy being obtained from the primary site or being read from disc as described in 6.3.3.

After decoding the request and validating the session identifier (see 7.1.2), the name (a,b) is looked up by server C to find the UID of the parent and to check that this request is allowed to be executed on behalf of the user or application program. Next, the server C creates the new object and writes it to disc with

Actions at C	Actions at P
Receive Request	
Validate Request	
Create New Object	
Lookup (a,b) to find P	
Call P	
	Check Component
	Update P
	Write to Disc
	Invalidate Other Caches
Reply to Client	

Figure 6.3: The Sequence of Events to Create a New Directory

reference count set to 1. Server C is to be the primary site for the newly created directory, but other servers are not informed yet in case the operation does not succeed, due to either failure of the sites involved or some condition preventing the object from being attached to the directory graph.

Since server C is not the primary site for the parent it can not update the parent to attach the new object to the directory graph. The fact that it is not the primary site can be determined by looking at the primary site address held in the cache copy of the directory. Therefore C issues an RPC to P including the UIDs of the parent and of the child. On receipt P retrieves the parent directory from its cache by using the UID supplied. Passing low level identifiers for the objects involved avoids the need to resolve path names at both servers. P checks to see if the component is in use; this check must be made here to allow for the situation where P has just created a component of the same name in response to another client's request or an RPC from another server. The commit point for the operation is passed when the parent has been successfully written to disc. P can inform all sites that there is a new object, passing the UID for (a,b,c), and that all cache copies of the parent, except that at C, should be invalidated. On the return of the RPC to C the copy of the parent is updated, since this client might access

the parent again; this is only an optimization as the copy is identical to that which would be obtained by calling P for a new copy. Finally, C can reply to the client to say that the operation has completed successfully.

There are several error conditions that can occur. If P finds that it is not the primary site for the parent directory then P is a rebooted server running on the machine of some failed server which was the primary site of the parent and the parent is currently without a primary site (for otherwise a server recovering from the loss of the original primary site would have notified C to alter C's cache state). In this case P returns an error and C recovers by attempting to become the new primary site for (a,b) by holding an election. If P is a rebooted server with the parent in memory as the primary copy then P called an election for the object; further, the election finished between C checking its cache to find P is the (old) primary site and C sending the RPC to P, for otherwise C would need to get a copy of the parent to perform the lookup. It does not matter that P is a new incarnation of a server as the disc representation is always up to date and update can proceed as before.

Should P fail before the commit point then no change has occurred to the permanent storage representation of (a,b); when the RPC from C to P fails C will hold an election and read the object from disc in the old state which is the same as that already in its cache. If P fails after writing the parent but before the RPC returns then the object read in will have a component c created and C will discover this. By having C check to see that (a,b,c) does not exist before making the RPC c can only exist when C recovers from P's failure if the operation succeed or there was another operation concurrently executing with that from C.

Should two servers C1 and C2 both attempt to perform operations that create (a,b,c) and P fails then it is possible that, say, server C2 in recovering will see a new component c created, whether it existed because of C1's operation or its own. It is the responsibility of the clients to coordinate actions that might result in conflict by, say, obtaining an exclusion lock; it is not the responsibility of the directory service because the name graph should not be used as a form of communication between users or applications.

If C fails at any time then the client must contact a new server and check the state of (a,b). It is guaranteed that either (a,b,c) has been successfully created or no change has occurred.

Although the actions that must be performed to execute the creation of (a,b,c) are many and complex there is least use of slow access to permanent storage. The writing of the parent is used to signal commit of the operation and should failure occur then no damage has been done to the filing system.

7 The Client Model

This chapter describes the filing system as seen by the clients in the distributed computing environment as well as how files are transferred between a file server and a user's local machine. The objective throughout is to have a structure that allows efficient implementation of the services required at the client. An implementation for a personal workstation operating system is detailed. In this chapter the term user is used for either a workstation or one of the people logged on to a shared machine.

7.1 Client Interface

There are 3 classes of operations provided by the directory service :–

- *Object Operations.* Creation and deletion of directories and deleting, storing and fetching files. Symbolic links have not been implemented in the prototype.
- *User and Group Manipulation.* To help with access control lists the directory service introduces the notions of user and group. Maintenance of the user and group data is performed by administrators.
- *ACL Manipulation.* Changes to the rights associated with an object.

Before the directory service can perform a request it must authenticate it to guarantee the identity of the sender beyond reasonable doubt. What is considered to be reasonable doubt will depend on the degree of security required.

7.1.1 Authentication

In any distributed system there are difficulties of authentication because of the separation of the client from the service being accessed. A malicious or faulty

program may attempt to present itself to the service claiming to be acting on behalf of any user. The vulnerability of the network to tapping or interference makes authentication in this environment difficult; in a secure kernel the presence of shared memory and hardware support results in different solutions.

There are two issues involved; authentication, which ensures that a message actually comes from who claims to be the sender, and authorization, which ensures that the requester has permission to perform the operation or access the resource. The problem of verifying the identity of the sender is general to a distributed computing environment and it is assumed that the environment provides a solution to this.

7.1.2 User Sessions

The second issue of authorization is simplified in the directory service. Any user may attempt to perform an operation but the result of the operation may either be a refusal because of access control restrictions or the user's intended result. All the information of the rights of a validated user are contained in the data held by the directory service; there is no need to go to some outside service.

To avoid the need to check the identity of a user for each request received, in the manner defined by the local environment, there is a session for each user. To start a session a client machine contacts the directory service with sufficient information to validate the user; the directory service authenticates the information and replies with a session identifier. By including a sufficiently long random number in this identifier it is possible to reduce the probability of the identifier being guessed to any desired low level.

All operations performed by the user will contain this session identifier to give an efficient validation of the identity of the requester. It is assumed that the kernel of a shared machine is trusted so that it will manage a user's session identifier correctly. This is reasonable since the kernel will manage the user's data when it is held on local disc or in memory and is being trusted not to release it to other users on the same machine. The kernel, rather than the user, must manage the

session identifier, so that applications can be run unchanged. If a cache server is being used to extend the facilities of a discless machine then the cache server will manage the session identifiers.

When a user starts a session the infrastructure of the distributed computing environment must provide the location of the directory service either by use of a nameserver, where the service name can be resolved into a machine address, by broadcasting a request for directory servers or by supplying this data as part of starting up a user's machine. This is the general problem of resource location in a local area network.

The client machine should be supplied with, or be able to deduce, the address of the most lightly loaded server; in a network where several LANs are connected by network bridges it may be advisable to ensure that the directory server is on the same LAN. From this point all user requests from this client machine will be directed to this server. The nature of the cache in the directory server means that it is sensible to allocate users on the same machine and clients running the same operating system to the same directory server unless loading dictates otherwise. The locality of reference generated by this allocation strategy will increase the efficiency of the caching algorithm.

The only point where a client machine needs to be aware that the service is implemented on a number of servers, rather than just one, occurs when the server being accessed fails. Failure within the directory service, except failure of all servers, is handled within the service. The client will need to find a new server, or servers, and create new sessions for the users. Sessions exist only to avoid full authentication on each user request and carry no state other than the session identifier. The only effect is a delay while a new session is established and longer response times for operations because the cache at the new site will not hold the objects previously accessed at the old site.

7.1.3 Object Operations

Two types of object exist in the directory graph, directories and files. Directories can be created or unlinked from the directory graph or can be used for path

name look up. It is not possible to delete an object, that is, ensure that the object no longer exists, but only to ensure that one possible name for the object has been removed. In an arbitrary graph there can be many paths, and hence many names, for an object; following back pointers would be required to locate all names for the object. The time and space needed for this is not justified; the unlink operation is sufficient.

Unlinking Objects

Reference counts kept with objects enable early freeing of resources on permanent storage and in the distributed cache; if unlinking an object results in that object having a reference count of zero then the object can be deleted from permanent storage. As in any directed graph structure there is a problem of a cycle becoming detached from the graph; although the objects in the cycle are unreachable from the naming root of the graph, none of them have a zero reference count. This only occurs with directories; a reference count on a file header is sufficient to determine when unlinking of a file causes it to be unreachable.

The resources used by these cycles must be reclaimed sometime. Garbage collection will find all unreachable objects and enable reuse of the resources allocated to them. Indeed, garbage collection is needed to clear up resources allocated but not reachable that were left by failure of a server. Garbage collection (for example [Garret80]) can be initiated when the service is not being heavily used, such as when backup of permanent storage is taking place. The rate of creation of these unreachable objects will be sufficiently low in normal use that garbage collection will only be needed infrequently.

Restructuring the Graph

A user may wish to change the name under which an object is held, that is, a change is to be made in the structure of the directory graph. However, it is not possible to perform an arbitrary renaming operation; if the old and new names are in the same directory, that is, only the last component is to be changed, then this operation can be performed atomically because only a single object, the parent

directory, is involved. Moving an object from one directory to a different one can not be done atomically because two directories must be changed and written back together. In keeping with the principle of providing primitives for clients to use to build their own model of file operations, the rename operation is only valid if the new and old names identify the same parent by the same path. Allowing naming of the parent by different paths was rejected because of the lack of clarity. If two seemingly similar rename operations were performed one might succeed and the other fail because, in the first, two dissimilar path names in fact currently name the same object while in the second they did not. This could not be determined by simply looking at the operation's arguments.

A client operating system can perform the more general rename operation of moving an object from one directory to another by combination of the link and unlink operations. This will not be atomic unless additional action is taken outside the directory service. By only applying the rename operation to the atomic case in the client interface it is made clear as to the difference in the two cases.

Listings of Directories

As well as changing the structure of the directory graph another common operation is to look at the contents of a directory. The information supplied includes the time of creation, a list of components in that directory and the access rights on each link. If the size of files in the directory is required then additional information requests will have to be sent to the directory service since a file's size is a property of the file, not of the name used to access it.

In a distributed computing system there can be a loss of some information because of the distribution of objects in the filing system. In a stand-alone system with one copy of each object all operations are directed to a single instance of the data representing the state of the object. When replicated there are multiple copies of the data with operations being performed on one copy without reference to other copies. In the case of the design described the operation of reading an object does not cause a record of the operation to be propagated to all objects. Therefore, there is no "last read time" for an object that can be obtained to be

supplied with the directory listing. The loss of this information is a result of the gain in efficiency of the replication scheme.

7.1.4 Users and Groups

The unit of access control is the *user*. It is usually associated with some person but there could also be one user for a number of people, as is the case with the UNIX user **root**, or a number of user identifiers for one person which convey different access rights, for example a person who works on several projects. A *group* is a list of users who are its members.

The operations needed are to be able to create and destroy users and groups and to be able to add and remove users from groups. Most of these operations must only be allowed to be performed by administrators of the system, for example to control who may be a user of the system. In the case of users it is also necessary to arrange that authentication of their identity can be carried out by whatever means the network infrastructure provides.

7.1.5 Access Control List Operations

Operations on access control lists – add member, delete member, change access of member and list members with accesses, where a member is a user or group – are most often performed by programs invoked directly from the terminal for the purpose of maintaining the access structure of the directory graph. It is unusual for an application which is not primarily involved in manipulation of the access structure to deal with access structure at all; its only contact is in performing actions on file and directories where a request may be refused due to insufficient access rights.

The programs which are directly concerned with access control lists are commands issued by a user during an interactive session. Because of the variety of operating system models that might occur in the single environment provided by the directory service there will be conflicts of access model. When security is considered there must be guarantees provided as to the permissible access to data.

The single access control model applied to the whole filing system will not be compatible with the models of each client operating system. It is mainly the command programs directly invoked from the terminal that will need to be changed because other application programs do not understand the access model except in terms of being allowed or refused access to data.

The result of this is that it is possible to have an access control system that does not allow each client's model to be expressed; the programs affected are small in number and are system commands, rather than user written applications.

7.2 Transfer of Files

Clients deal directly with file servers to transfer files to and from local disc. There is a single, network-wide protocol. With just one protocol there is not an explosion of work to be done when a new type of client or file server is introduced into the network. Only the new equipment needs to be modified to use the network standard; all other equipment requires no changes to be made. This makes the introduction of new equipment faster and less disruptive to the original components of the distributed environment.

7.2.1 Protocol Requirements

File transfer will be a common operation and hence the protocol used to support it must be efficient while still providing a reliable communications link.. Conflict arises from the variety of file sizes that are to be found. Although the average file in a filing system is under 4Kbytes (see figure 7.1) there are many files larger than this that are frequently used. It has been found that the average file weighted by frequency of use is about 11Kbytes on a UNIX system [Lazowska86].

Small files will fit inside one or two ring basic blocks so a protocol that does not set up a circuit but just sends the data without entering into any initiation of the transfer will be more efficient. For larger files, where many basic blocks are involved, there is a need to allow the receiver to allocate resources before the

File Size	UNIX		TRIPOS	
	Number	Cumulative	Number	Cumulative
0	683	0.7%	292	1.2%
256	13110	14.4%	3667	16.5%
512	10190	25.1%	2389	26.4%
1Kb	14665	40.5%	2989	38.8%
2Kb	15031	56.2%	3198	52.1%
4Kb	13980	70.8%	3176	65.3%
8Kb	11293	82.6%	2973	77.7%
16Kb	8371	91.4%	2688	88.8%
32Kb	4694	96.3%	1497	95.1%
64Kb	2240	98.6%	693	97.9%
128Kb	733	99.4%	274	99.1%
256Kb	293	99.7%	150	99.7%
512Kb	112	99.8%	39	99.9%
1Mb	55	99.9%	27	100.0%
> 1Mb	46	100.0%	5	100.0%
Total	95496		24057	

Figure 7.1: Distribution of File Sizes

contents are sent. By calling ahead to set up a connection between the sender and receiver it is possible to reduce the amount of protocol data in each block subsequently sent. There is more room for the file's contents, which reduces the number of blocks sent, and the receiver can prepare to receive the file without having resources permanently allocated for the transfer of larger files

7.2.2 Network Considerations

In developing a protocol for transferring files the characteristics of the network involved are important. They influence the amount of work that must be done in order to reduce the chances of an undetected error in the data to acceptably low levels and the style in which this is done. The objective is to maximize the expected throughput of data from sender to receiver.

Networks error properties that should be considered are those of the likelihood of packets being lost, reordered, duplicated or spurious packets being introduced.

Corruption of packets can most easily be eliminated by a checksum that associates a tag value that is calculated from the contents of a network packet. The receiver then calculates the expected value of the checksum and compares this with that provided. If the value calculated and the value supplied are the same it is assumed that no corruption of the packet has occurred; by choosing the length of checksum to be sufficiently long we can reduce the chances of undetected error to below the desired probability.

Packet reordering and duplication can be identified by placing a sequence number in each packet sent, indicating its correct position in the sequence of packets. From the sequence numbers the receiver can recreate the correct order and throw away any packets that are received twice.

Packet loss can only be detected in general if there is a length of time after which it is reasonable to deduce that the packet is not going to arrive. This timeout period will depend on the network. If we know that reordering of packets does not occur then a jump in a sequence number indicates that there has been loss of a packet.

If the probability of some kind of error happening is large then checking individual packets, acknowledgement of correctly received packets and requests for incorrect or lost packets to be resent are actions to be put in the protocol. However, the error rate in most local area networks is very low so that a simple check on all the packets received may be sufficient.

The Cambridge ring does not reorder or duplicate basic blocks but packets may be lost if, for example a receiver is congested. Therefore, a simple total count of the packets received and a checksum on the correctness of the data will detect all errors that might occur.

7.2.3 An Adaptive Protocol

File transfer is achieved by a strategy that allows for a single SSP or a pair of SSP's for small files and for a ring OPEN protocol for large files. The receiver is listening on a port that is either well-known, in the case of file server, or passed

from the client, via the directory service, to the file server in the file fetch request. The function code of the first block can then be used to differentiate the various policies. These policies are chosen to minimize the number of network messages that are necessary. It should be noted that the presence of bridges means that the address of, for example, a file server is not constant as seen by a client on another ring and that the nameserver must be queried to obtain a route through a bridge.

The protocol is very lightweight in that it has no error recovery facilities and the receiver timeouts if insufficient blocks are received. The number of blocks to be expected can always be deduced from the first block received. The sender expects an acknowledgement that all the blocks have been correctly received by the other end. In the case of sending material to a file server this acknowledgement should not be sent until the file's contents is on disc or where it can be guaranteed to persist in case of a failure. This assures the sender that the file is safely stored and the sender can release resources used to hold the file.

Single Block

A single SSP block is sent with the function set to 1101 (See figure 7.2). There is a 16 bit checksum, space for a token for when a client is sending to a file server to write a file, the size of the file being sent, then up to 2028 bytes of data. Since an SSP block must be an even number of bytes long it is necessary to send the file's length; sending a shortened basic block only indicates the length to the nearest two bytes. The reply to the SSP is a standard SSP reply block with return code set to zero on success or an error indicator to show invalid checksum, invalid file length or illegal token.

Double Block

The first packet of a double block file is the same as a single block file except that the function code is set to 1102. The reply to this indicates the port.that the next block should be directed to; this enables a well-known port on a file server to be kept as free as possible of incoming blocks so that other machines sending material are not kept waiting.

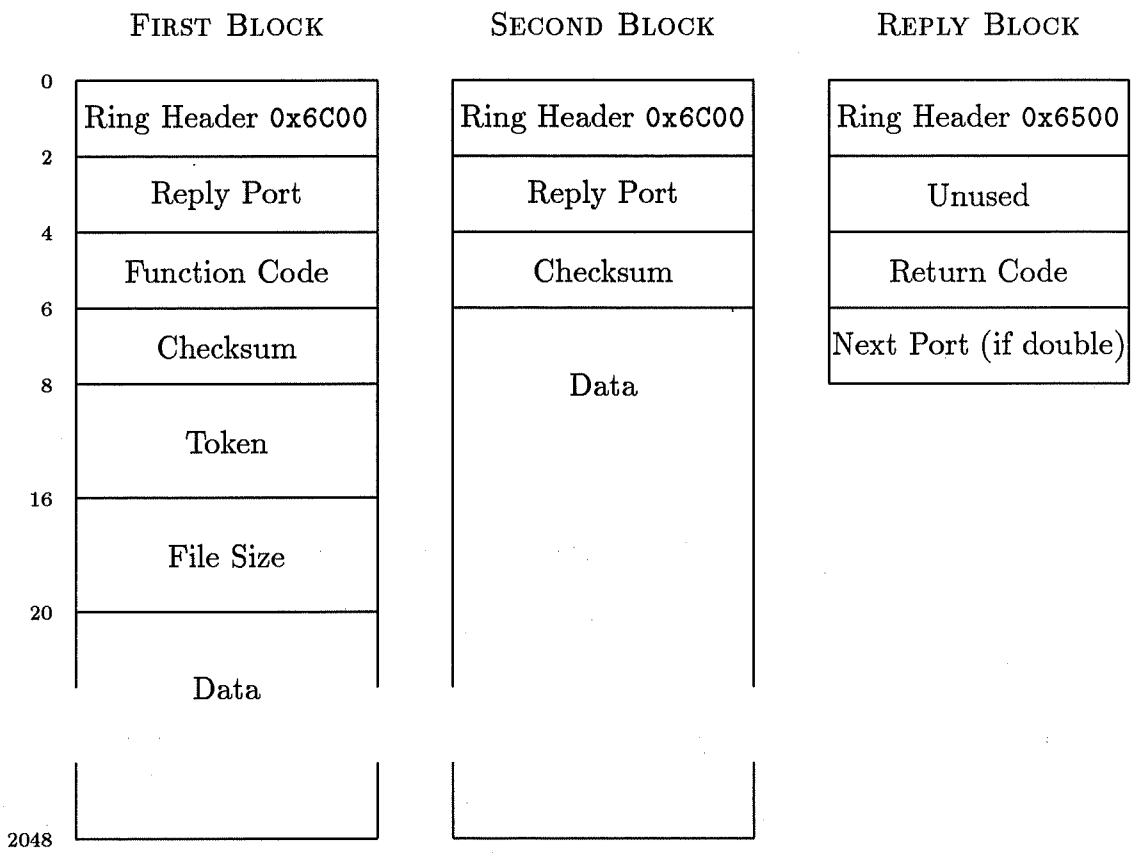

Figure 7.2: Block Formats for Single and Double Block Transfers

The second block contains a checksum and up to 2042 bytes, so that the two blocks together cope with file sizes from 2029 to 4070 bytes.

Multi Block

For files of size 4071 bytes or more then a basic block level connection is established between the sender and receiver. In the case where this goes through a bridge this avoids the setup costs of contacting the local nameserver on each block.

The first block, which is similar to that used in the single block protocol, contains a function code set to 1103 to distinguish from the two cases above. From the length of the file the receiver can calculate the number of blocks that will be sent.

Subsequent blocks are sent that contain a checksum and data, up to 2044 bytes

Basic Blocks Sent			
File Size	To Receiver	To Nameserver	Total
1 Block	2	2	4
2 Blocks	4	4	8
n Blocks[a]	$2+(n-1)+1$	6	$n+8$

[a] Other messages sent are not on the critical path.

Figure 7.3: Costs of File Transfer

per block. The receiver sends an SSP block to the sender when all the data has been received and safely stored. In order to perform this acknowledgement SSP the receiver must contact the nameserver in case the sender is on another ring and a route through one or more bridges must be set up.

A sender may choose not to use the double block form of the protocol and, instead, send a file of that length by the multi block form. This is useful when the application is writing a file on some client and the eventual length of the file is unknown. The first block must leave space for the protocol header, but this is the same for single and multi block forms. A subsequent block would need space left if the double block form will be used but not if the multi block form will be employed. By going straight to the multiblock form a client avoids the need to copy data from one buffer to another to reformat it.

7.3 Client Structure

As an example of the changes needed for a client operating system a modified version of TRIPOS has been produced. To provide binary compatibility with existing applications it is necessary to replace the file handler for the filing machine with one that has the same interface but which maps TRIPOS directory operations to the corresponding directory service operations and performs file operations on the locally stored copy. The TRIPOS filing system structure is a directed graph and the interface is at a similar level to the interface for the TRIPOS filing machine so this translation is simple except for access control which presents some

difficulties because the TRIPOS model is very different from that of access control lists.

7.3.1 A New File Handler

When a file is opened for reading it is transferred to the local machine synchronously so that the application must wait for the whole file to arrive before it can continue. This could be avoided by replying to the client as soon as the file starts to arrive, this indicates that the request to the directory server has been successful, but this would only be of significant advantage for larger files. For subsequent requests to obtain data from the file no external action is necessary as the whole file is cached at the client. A cache server would be necessary in a production implementation because of the memory limitations of the TRIPOS machines. This allows for small memory clients, freeing store for application use. The current implementation uses memory in the client for complete files and is therefore unsuitable for large files and for use other than as a prototype.

When opening a file for writing the file handler obtains a token from the directory service before replying to an application's request to open a file for writing. As data is passed by the application to the file handler it is copied into buffers with space allowed for the FTP protocol headers. To make this simple the double block method is not used as the amount of data is not known until the file is closed. Once the first block overflows a second block is allocated and filled, with no room left for a header. This is in the format for the multi-block method which has blocks consisting only of data after the first one. Whether a double block transfer will suffice is not known until the second block is full or the file is closed. At a slight loss of efficiency for certain sizes of files, a client machine never uses the double block form to send data to the file server although it will accept file transfers in this format. Since files are written less often than they are read and since only files in the range 2029 to 4070 bytes are affected, the loss is insignificant.

Access Control Mapping

The notions of refinement of access along a path and the passing of rights from one component to the next are the most difficult to reflect in the primitives of the directory service. It is possible to rely on convention in solving problems such as the character set used in component names because adhering to the convention is advantageous to the clients. Access control only restricts clients; more freedom is be obtained by bypassing the system. Thus, it is necessary to enforce the functionality.

As it is not practical to provide a system which is equivalent to that of every client system of access the only option is to cater for the majority of cases which will leave most programs still runnable in the new system, but require a few programs to be altered. This is more attractive if these few programs are commands, rather than large sub-systems or user written applications.

The proposed alteration is based on the observation that the main concepts used are those of 'user' and 'other'. By calculating the access to each object in the filing system when it is transferred to the directory service the effective owner can be determined and an access list for this user and for the default access can be constructed. By this method the actual use made of the filing system is retained even though the exact semantics are not preserved.

For a system such as UNIX or VMS the translation of the access control model is simpler because these systems do not present such an unusual model.

8 Conclusions

8.1 Work Carried out

A design for a distributed, network directory service has been developed for large, heterogeneous local area networks and a prototype implementation has been constructed. The work has shown that file sharing between diverse operating systems may be supported without compromising the efficiency of file access and that evolution of the distributed computing system does not need to cause disruption to existing components. The directory service uses the features of the network to give a reliable service that can survive loss of some of the servers implementing the service and, at all times, the service is fail-safe, since the directory graph is left in a valid state as actions are committed by a single write to permanent storage.

The filing system is central to a computing system so efficiency of the directory service is important; this is achieved by replicating the service. Each server is accessed by some of the client computers and uses a cache of directories to avoid network access; because clients remain bound to the same server, while it is available, the cache in that server will tend to hold those directories frequently accessed by programs run on the clients. The distributed cache is kept consistent by a primary site update policy. This policy is chosen because it gives the lowest expected amount of communication between servers, since a cache entry is up to date at all times. To decrease the total amount of work needed to manage the cache, read operations are made cheap, while write operations incur greater overheads. This reduces the total amount of work needed in cache management because reading of directories occurs more frequently than updating.

The TRIPOS operating system was chosen to show the design of a client kernel. The choice of this operating system was motivated by the unusual access control model employed; other systems that use access control lists in their local model of

access control are easier to map onto the facilities of the network model provided by the directory service because of the similarity of their access models to the network model.

8.2 Evaluation

Functionality

The directory service does not aim to impose a single model of file access on its client operating systems and hence on the users and applications. Instead, the functionality of the directory service provides flexible primitives that allow the clients to present their local model of file access to applications. A danger of this is that misuse of the primitives can obstruct sharing; for example, file names that only differ by the case of some of the characters can not be accessed by an operating system that equates the case in file names. However, the security of the filing system structure and the data contained in the file is not compromised because access control is enforced by the directory service and because operations to update and read the name space are not performed directly by clients. The interface to the directory service lies between that of an imposed filing model and allowing complete freedom of data access and of name space structures; it allows most models of filing but ensures that security of data and integrity of the name space are not compromised. The range of filing models that can be accomodated includes conventional computer systems; it does not provide a basis for specialized systems such as databases or object-based programming environments.

Distribution

The directory service makes no attempt to conceal the distributed nature of the file storage servers that make up the permanent storage in the network. However, the distributed nature of the service itself only becomes evident to a client when it can no longer contact the server that is handling its requests. If failure transparency or network transparency is required then the application is responsible for constructing this from the local filing model and from direct access allowed by

an extended kernel interface to the primitives of the directory service. Thus, the environment presented to the user is that of the local operating system together with the new modes of failure; the user is expected to understand the additional properties of the system that distribution brings.

Enforced access control means that commands that allow change of access must be rewritten and users are not guaranteed to be able to use the old command. In this way the directory service is visible to the user; this differs from systems such as LOCUS where every effort is made to present the view that the distributed system is one large machine with increased reliability, availability and performance.

Scaling

The directory service is designed to operate in a single local area network. Communication across the network and between the directory servers is expected to be sufficiently reliable to permit the use of the optimistic recovery policy of assuming that operations succeed until an indication to the contrary is found. There is no allowance for partitioning of the network; if this could occur then the design would need to ensure that it was impossible for there to be two primary sites for the same object.

The caching policy limits the scale of the service; when an object is updated all other sites that have the object cached need to be invalidated to fulfil the condition that all cached copies are up to date. The cost of this will depend on the number of replicas. Currently all sites, regardless of whether they have a cached copy or not, are contacted. This cost is especially important if an efficient reliable multicast is not possible, for then the invalidation costs are proportional to the number of replicas, and hence related to the load on the system and the number of servers.

The area that is most affected by the scale of the directory service is the election algorithm. Under the current scheme all sites are contacted to obtain a distributed lock. Even obtaining a majority of the sites does not affect the growth rate of the cost (which is $O(n)$, where n is the number of servers). The algorithm is based on the assumptions that communication through the system has a low failure rate

and that the cost of making a remote procedure call is the same between any two network nodes. These assumptions may not hold in wider area or extended local area networks.

Administrative Domains

The directory service is assumed to be operating within a single administrative domain; a server will proceed with operations requested by another server without any checks, trusting that the requesting server is authorized to make the request. The administrative authority is assumed to be the same for each server. In a larger scale network this is not guaranteed to be true; indeed, if we have two local area networks that have been joined, it is unlikely to be true. With this assumption it is acceptable to use an election algorithm that assumes the directory service is operating in a single local area network. The design of the directory service does not cover environments that have several administrative domains and hence its scale is bounded.

8.3 Review

This thesis has concentrated on the integration of existing systems rather than constructing a new model of file access for applications. This is motivated by the desire to be able to run software that already exists; the programmer's model used to develop that software must be maintained. At the same time it is recognized that, with the introduction of a new filing system to cover all machines in the network, there is a requirement for new software not to be constrained by previous models and to be able to use the full facilities that exist in the local environment of the new filing system.

A filing system provides permanent storage of information, a naming structure to be able to manage that information, and control of the sharing of data. The principal technique employed in the design is to separate the storage of data from the management of the name space by using a three party model of file access. A single, network-wide directory service exists to provide management of the name

space and to control access to data. File storage servers provide the storage of information; the wide variety of naming schemes and other facilities that are to be found in file server designs means that file servers can not be used for the naming role. The collection of clients transfer whole files to and from file servers using a standard network file transfer protocol that allows for a three way exchange of messages to access a file. The kernel of a client machine can then provide the local semantics of file access to application programs. It is the kernel that uses the functionality of the directory service to provide the local semantics of file names where possible.

Sharing of information is an important feature of a filing system. It occurs between network nodes running the same operating system and between nodes running different operating systems and is not necessarily planned; access to a file may be required that could not be anticipated when the file was created. This leads to a network filing system, rather than a shared file store to supplement local filing.

To allow an application to run unchanged it is necessary for the full range of file names to be expressible in the directory service. Thus, sharing between applications running on the same operating system is supported, which is the more frequent case. Sharing across different systems can not be achieved so easily. Restrictions on file names in one system may render inaccessible files generated by another client. If there were restrictions on file names in the directory service then applications would not be able to run unchanged. By extending the operating system interface to allow direct access to the directory service, without the local translation of file names, an application written for this environment may name any file. In particular, symbolic links created by a specially written program may be used to allow an imported application to access data created on another operating system under an otherwise inaccessible name.

A network standard representation of data in a file is not enforced; this allows applications to avoid overheads involved in translation to and from the local data format and to avoid overheads of increased file size. This is orientated towards sharing of files by applications running on the same operating system. When a file

is accessed from another system it is the responsibility of the application reading the data, possibly in cooperation with the application writing the data, to be able to understand the file's contents.

One area that can not be addressed by the technique of providing facilities and allowing clients to construct local models on these facilities is that of access control. It is necessary to provide guarantees that data is not released or altered by some other client or application without permission being granted, directly or indirectly, by the creator. This results in a single access control method being enforced by the directory service rather than allowing client kernels to construct their own access model; this is because the directory service does not trust the clients. This may be considered to be a difficulty that is unavoidably in conflict with the objective of providing local semantics. However, consideration of the use that is made of access control shows that the majority of programs do not understand the full model of access control; only the granting or denial of access to a file is used. No manipulation or interpretation of the access control is performed. Programs that do handle, for instance, access lists are those invoked directly by the user, such as commands for listing a directory's contents or changing access rights. By accepting that changes have to be made to this small class of programs it is possible to integrate various systems into one, network-wide filing system.

A Prototype Design

In this appendix the design of the prototype is discussed; the data types used to model the directory graph are described and the principal algorithms are detailed. Illustrations are given as fragments of CLU [Liskov81].

- Pathname lookup : clients of the directory service name files and directories by pathnames. Therefore the operation of looking up a pathname is important. It uses the cache, whose entries are always up to date, in order to reduce the need to access other servers.

- Creating a new directory : this operation is typical of the operations performed by the directory service on behalf of the clients (see section 6.7). It does not use a general recovery scheme, but instead, uses a scheme tailored to the semantics of the operation. By carefully ordering the actions taken there is increased availability of the parent directory during the update.

- Obtaining a distributed lock : a distributed lock is needed whenever a file header or directory is read from disc. The election algorithm is implemented to achieve mutual exclusion among the servers.

A.1 Data Types

The main components of each server are two modules for controlling the distributed cache and the distributed locking, and four abstract data types making up the directory name graph. The directory graph is composed of nodes, of type *gnode*, which are a discriminated union of types *fnode*, which is a file header, and *directory*. Each directory has a number of named links to other nodes in the graph. The name of a node in the graph is a *path*. The term *object* is used to cover either a file or a directory.

gnode

```
rep = record [
    uid       : uid ,
    back_ptrs : array[uid]    % The reference count.
        % Information added when held in the cache.
    home      : address ,     % The address of the primary site.
    node      : object ,      % The contained object.
    lock      : lock ,        % Local lock for update operations.
]

object = oneof [ file : fnode , directory : directory ]
```

Figure A.1: The definition of gnode

The directory graph is held on disc as a collection of objects of type *gnode*, named by UID. Additional information is held with an instance of a gnode when it is held in a server's cache. See figure A.1.

The only information needed from the back pointers is the size of the array to give the reference count. However, for debugging purposes it is useful to be able to take any node and find out which directories reference it. The directory service itself makes no use of the UIDs held in the back pointer list. It should be noted that CLU arrays can be used as linked lists; they have operations to remove or add elements which change the length of the array. Hence, the length of the array is the value of the reference count.

The cluster *gnode* also includes the code for reading in the meta-root (see section 6.2.1) from disc during initialization of each server. Directory server initialization requires that the meta-root is placed in the cache. The UID of the meta-root can be built into the code as it must exist before all other objects. If it is not found then a new meta-root is created under the UID expected for the meta-root. Once the server has the meta-root in the cache it can process requests.

fnode

The file servers hold the contents of files and the transfer of the contents only involves the clients and the file servers. The directory service, however, manages

the file headers and these are held in the type *fnode*. A record is kept of the last time the file was written to a file server, when it was created and its current size. This information is kept up to date by receiving changes when a client uses a token to write a file.

directory

```
rep = record[ create_time   : time ,
              last_change   : time ,
              entries       : entries ]

entries = monitored_table[string,link]        % Safe hash table

link = struct[ acl    : acl ,
               item   : item ]

item = oneof [ file : uid , directory : uid ]
```

Figure A.2: The structure of a directory

An instance of a directory contains information similar to a file header, time of creation and last write, as well as a list of components. The components are held in a hash table that has safe operations for concurrent reading and writing; this means that read operations do not need to lock the whole directory. Write operations claim the local lock of the gnode before carrying out the changes. This is required because some operations, such as renaming an entry, must read the directory to ensure that the operation is valid before proceeding with the change.

The hash table is described in figure A.2. A component does not name an object because an object can not be a symbolic link; an *item* could also be a symbolic link in a complete implementation. The algorithm for pathname lookup would recursively call itself if it encountered a symbolic link.

path

The path data type holds pathnames as a sequence of component names. Operations are provided to lookup a name; this takes two forms, either looking up the name as given or looking up the name to find the parent.

```
cache_table = monitored_table[uid,state]

state = variant[ cached   : gnode ,
                 primary  : gnode ,
                 known_ps : address ]    % Address of primary site.

% Return the gnode with UID u.
% On return the object is in the local cache.

fetch = PROC(u:uid) RETURNS(gnode)
    % Test to see if there is a cache entry already present.
    s:state := cache_table$lookup(cache, u) EXCEPT
      WHEN not_found :
        % No cache entry - read from disc.
        RETURN ( become_primary_site(u) )
        END
    % There is an entry for this object in the cache.
    TAGCASE s
      % Already got a copy - return it
      TAG cached, primary(g:gnode):RETURN(g)
      % Haven't got a copy but know where the primary site is.
      TAG known_ps(na:address):
        g:gnode := CALL rpc_retrieve ( u ) AT na EXCEPT
          WHEN call_failed , not_primary_site:
            RETURN ( become_primary_site(u) )
            END
        <enter the pair (u, state$make_cached(g)) into the cache>
        RETURN(g)
        END
    END fetch
```

Figure A.3: The distributed cache

dlock

The acquisition of a distributed lock is structured as a gate [Cooper85]. A gate is a control structure that consists of two parts, entry and exit code, which enclose a block of code, the body of the gate. The entry code is executed before the body and exit code after. The compiler ensures that all possible exit paths, whether they are due to exceptions, return statement or premature ending of an enclosing loop, will reenter the gate to execute the exit code.

dcache

The part of the distributed cache held on each server is a hash table which

allows concurrent operations. Each entry is a mutable tagged union giving the current state of that entry; the entries have their UID as key. The principle operation is fetching a cache entry. If there is a locally available copy then this is necessarily be up-to-date and can be returned immediately; if, however, the primary site is known, but there is no copy in the cache, then an RPC call is made to obtain a copy. The call must cope with the case of the failure of the primary site, followed by the starting up of a new server. This would result in the exception `not_primary_site` being raised. Should this new server not be the primary site for the object then the requesting site calls an election to read in the object from disc.

A.2 Pathname Lookup

Operations that need to specify an object in the filing system do so by supplying a pathname as one of the parameters of the operation. This pathname is a sequence of components that enable the required object to be found. The internal representation of a path is a sequence of strings. Such a sequence is an immutable object; that is there are no operations that alter the sequence once it has been created. Lookup proceeds by obtaining the well-known root of the name graph; this is held by the module that manages the distributed cache in the form of the gnode of the meta-root. Once the directory has been obtained the lookup can start.

There is a slight difference between the requirements for the final component and the remaining initial components. Only the final component can be a file, the others must be directories so that the appropriate component can be looked up.

The first component is passed to the directory cluster to obtain the UID of the object that this component names by looking it up in the root directory. This object is either a file or a directory; if it is a file then the path is not well formed and an error is returned in the form of an exception; if the object is a directory then the UID is used to retrieve it from the cache. This operation will cause the directory to be read in or fetched from another site should this be necessary.

```
qs = sequence[string]

lookup = PROC(p:rep, u:user, access:access_type)
         RETURNS(object)
         SIGNALS(invalid_path,
                 insufficient_access,
                 not_found)
  g:gnode := gnode of the meta root          % Get the well-known root.
  d:directory := gnode$value_directory(g)    % Extract the directory.
  first_part:qs := qs$remh(p)                % Name of the parent.

  FOR component:string IN qs$elements(first_part) DO
      obj:object :=
          directory$lookup(d, component, u, search_access)
          RESIGNAL insufficient_access, not_found
      TAGCASE obj
        TAG file : SIGNAL invalid_path
        TAG directory(duid:uid) :
           g := dcache$fetch(duid)           % gnode of next directory.
           d := gnode$value_directory(g)     % Next directory.
           END
        END

%  Now do the last part of the path - this is slightly different
%  because it may be a file.

  obj:object := directory$lookup(d, qs$top(p), u, access_required)
    RESIGNAL insufficient_access, not_found
  RETURN(obj)
  END lookup
```

Figure A.4: Path name lookup

The newly found directory then replaces the initial directory and the search can proceed to the next component. This is repeated for all the components except the last. Once the parent of the named object has been determined a separate lookup which allows the object to be a file, as well as a directory, is performed.

A.3 Creating a New Directory

```
% p is the path name of the object "parent" in which the new
% directory is to be created.

d:directory := directory$create()          % Allocate memory.
child:gnode :=                             % Create the graph node.
    gnode$create_directory(d,uid$next())
% Set reference count to 1 and write to disc.
% This announces the existence of a new UID to all sites so that they
% have known_ps cache entries for it.
gnode$add_back_ptr ( child , parent.uid )
dcache$new_entry(child)                    % Place it in the cache.
% Attach it to the name graph.
directory$add_entry(parent, path$last_component(p) ,
                    object$make_directory(child.uid) ,
                    acl$default(u)) EXCEPT
  WHEN duplicate_entry:
    gnode$delete_from_disc(child.uid)
    SIGNAL duplicate_entry
    END
```

Figure A.5: Creating a New Directory

As an example of the techniques used by the directory service consider the creation of a new directory (figure A.5). Two actions must occur. Firstly, a new object is created and written to disc. This is achieved by creating a new directory within the server and writing it disc with the reference count set to 1. Secondly, it must be attached to the name graph. This is performed by the operation `add_entry` on the parent; the parent's name can be determined by taking all but the last component of the new object's pathname and this is used to obtain the directory `parent` before the child is created.

Updating the parent can only be performed by the primary site of the parent. Therefore, if this is not the primary site the operation is forwarded to the correct

```
add_entry = PROC(g:gnode, entry:string, obj:object, a:acl)
            SIGNALS(duplicate_entry)
  d:rep := <directory contained in gnode g>
  IF this is not the primary site for d THEN
    CALL rpc_add_entry(g, entry, obj, a) AT g.home EXCEPT
      WHEN call_failed, not_primary_site:
        % RPC to primary site failed or new server on old primary site.
        dcache$become_primary_site(g.uid)
        add_entry(dcache$fetch(g.uid), entry, obj, a)
        RETURN
        END RESIGNAL duplicate_entry
    % Keep cached copy valid by altering the copy in our hands.
    <update directory d>
  ELSE
    % This is the primary site.  Exclude all other update operations.
    USING lock$region(g.lock) DO
      <update directory d>
      % Write to disc.
      gnode$to_disc ( g )
      <single round of messages to invalidate caches at other sites>
      END % end of critical region.
    END add_entry
```

Figure A.6: Adding a New Entry

server. This leads to two cases; in the first the call to `add_entry` is made at the primary site and in the second the primary site must be contacted to perform the change as there is only a cached copy at this server. As in fetching a cached copy, the call to the primary site may encounter a new server running on the site of the failed primary site.

At the primary site, the update is performed under a local lock to exclude other operations updating this directory. When the change has been made the lock is released after the change has been propagated to all other sites so that any cached copies can be invalidated and the cache entries made into `known_ps`. As an optimization, if the update is initiated at a site with a cached copy, the copy is updated and reinstalled in the cache.

A.4 Distributed Locking

The code for the gate `wait` is the entry and exit code for critical regions that

```
wait = GATE(u:uid) SIGNALS(no_lock(address))
    voters:anet := anet$[]
    FOR na:address IN <known servers> DO
        IF <this server> THEN
            lock_uid(u)
        ELSE
            CALL rpc_lock_uid(u) AT na EXCEPT
              WHEN call_failed:
                <remove from list of known servers>
                CONTINUE
                END
            END  % If
        anet$addh(voters,na)          % Remember sites who have voted.
        END EXCEPT
          WHEN no_lock(who:address):
            % We lost the election.  Release locks without updating
            % the cache entry for this UID.
            FOR addr:address IN anet$elements(voters) DO
                IF <this server> THEN
                    release_uid(u,who,false)
                ELSE
                    CALL rpc_release_uid(u,who,false) AT addr EXCEPT
                      WHEN call_failed:
                        <remove from list of known servers>
                        END
                    END
                END
            SIGNAL no_lock(who)
            END

   KEY   % Perform the body of the critical region.

   % Release the lock everywhere; update cache to indicate known
   % primary site for this UID.

    who:address := this_address()
    FOR addr:network_address IN <known servers> DO
        IF <this server> THEN
            release_uid(u,who,true)
        ELSE
            CALL rpc_release_uid(u,who.true)
              WHEN failed: <remove from list of known servers> END
            END
        END
    END wait
```

Figure A.7: Distributed Locking

update an object in the directory graph. The keyword KEY separates the entry code from the exit code. Locks are obtained on UIDs and each site maintains a list of current locks with atomic operations of `lock_uid` and `release_uid`. In addition to the mechanism outlined in figure A.7, each site will put a timeout on the lifetime of a lock to cope with the failure of the server locking the UID. This is not shown in figure A.7 for simplicity but is implemented in the prototype.

If there is contention for a lock at some site, then the second attempt to lock on the UID is blocked until the lock holder releases the lock. Then any blocked sites are told that they did not succeed in obtaining the lock and are passed the name of the winner of the election. This is achieved by the signal `no_lock(winner of election)`.

The election proceeds by polling each site since there is no broadcast available on a Cambridge ring. Each site either agrees to the request by returning normally from `rpc_lock_uid`, or raises a `no_lock` signal. Any failed site is ignored. If the election is successful the body of the critical region is executed. On entering the exit code all locks are released.

There is one optimization; lock release takes a boolean argument indicating whether the cache entry for this UID should be invalidated because the object has changed. By passing this with the lock release an extra round of messages may be avoided.

Use of Distributed Locking

The procedure `become_primary_site` (figure A.8) is called when an object is to be read from disc. This occurs when the first reference to an object is made or when recovery from failure of a server is in progress.

If, during the election, a lock is not obtained then the eventual winner of the election will have a valid copy of the object in its cache. Therefore, a call of `rpc_retrieve` can be used to obtain a copy. A failure of this call leads to a recursive call of `become_primary_site`. This can not lead to blocking out a server which is attempting to read an object due to an infinite sequence of other servers

```
become_primary_site = PROC (u:uid)
                      RETURNS (gnode)
    USING dlock$wait(u) DO
        g:gnode := gnode$read_from_disc(u)
        <update cache table with (u,state$make_primary_site(g))>
        RETURN(g)
        END EXCEPT
          WHEN no_lock(who:address):
            % Lock contention.
            g:gnode := CALL rpc_retrieve(u) AT who EXCEPT
              WHEN call_failed, not_primary_site:
                % Just try again: we can hold elections faster than
                % sites reboot so this will not lead to livelock.
                RETURN(become_primary_site(u))
                END
            RETURN(g)
            END
        END
    END become_primary_site
```

Figure A.8: Reading an Object from Disc

starting up, then failing because rebooting takes much longer than it takes to hold an election.

B Prototype Implementation

This appendix gives details of the prototype implementation of the directory service. It was constructed in the Cambridge Model Distributed System [Needham82] [Bacon87], providing a collection of personal computers, which are allocated on demand. These are managed by the Resource Manager, one of a number of services that provide the infrastructure of the CMDS. The addresses of the standard services and of machines on the ring are found by sending requests to the nameserver, which resides at a well-known location. The nameserver maps the name of a service or a machine to the address of its current location by means of a small table, held in ROM, and a larger table held on a file server. The ROM table enables the larger table to be found during initialization. This allows some reconfiguration of the ring, but only of the fixed, standard services. It is not possible to create new services, or relocate existing ones, in a manner that is sufficiently convenient to allow dynamic reconfiguration while the system is in use. Permanent storage is available to applications in the CMDS through the use of file servers [Dion80].

A directory server runs on a 1 megabyte Motorola 68000 machine, running the Mayflower operating system [Hamilton84]. The implementation language of the servers is CLU, extended to allow multiple processes, concurrency control [Hamilton84][Cooper85] and remote procedure calls. One server resides on each machine currently providing the service. Synchronization and communication among the processes that make up the server is achieved by use of the shared memory.

B.1 Initialization

The setting up of the directory service is controlled by a program running on one of the machines in the processor bank. This controller acquires the necessary machines for the service, causing them to be loaded with Mayflower. It then initializes them so that the service can start.

Each server needs to know which other machines are participating in the service. Ideally, each server would register itself with a nameserver on the ring so that clients and other servers could discover the addresses of the servers currently providing the service. Unfortunately, the nameserver has a static, built-in table that maps a service name to a single machine address, which is not sufficient for the needs of the directory service. The requirement is for a service name to map to a list of addresses, and for this mapping to be alterable by sending messages to the nameserver. Therefore, the process of initialization of a server includes passing a list of other servers. Once started, the servers maintain this list among themselves, with any new servers created by the controller informing the old servers that they have joined the directory service.

Initialization of the servers occurs in two phases. During the first phase the code is loaded and the RPC mechanism started. This is done simultaneously to all the machines and leaves them in a state where they are ready to enter the next phase of initialization. The second phase, which occurs concurrently on all machines, causes the caches to acquire an entry for the meta-root. A list of servers and the name of the file server are sent to each machine. The list of servers is passed to the distributed locking module, *dlock*, and the name of the file server to the code managing UIDs on disc. Finally, the meta-root is located as in figure B.1.

Several possibilities may occur when locating the meta-root. If some other server has already completed its initialization process, there will be an entry in the cache indicating the current site; no further work is needed. If no entry is found an election is called to read in the meta-root from disc. It is necessary to check the cache once the lock has been obtained because some other server could be releasing a lock on the meta-root while this server is acquiring one. The first

```
init = PROC()

    r_duid:uid := <uid of the meta root.>

    % See if we can avoid locking to read the object.
    IF <in cache> THEN RETURN END

    USING dlock$wait(r_duid) DO
        % Ensure that it wasn't done during lock acquisition.
        IF <in cache> THEN RETURN END
        root:gnode := gnode$read_root()
        <insert into the cache as primary copy>
        END EXCEPT
          WHEN no_lock(who:address): END
    END init
```

Figure B.1: Initializing the Distributed Cache

check of the cache is only an optimization to reduce the probability of locking.

The election may well fail because some other server is in the process of initializing its cache with the meta-root, and hence has a lock on the meta-root. When this occurs the locking mechanism does not return until the initial election has completed. Therefore, the primary site for the meta-root is known and has already been inserted into the cache during lock release. No special code is needed for locking the meta-root, although the high probability of lock contention means that the algorithm will not perform efficiently.

In addition to acquiring the resources and initializing servers the controller also provides a command interface for testing the service by invoking operations and by monitoring the state of the caches in each server. Operations can be directed at particular servers, simulating requests from client machines. It is also possible to cause failures in a server.

Also resident on the controller machine is a process that provides the file server facilities of storing and fetching files using the FTP protocol detailed in chapter 7. This avoids the need to modify the file servers on the ring, while at the same time allowing clients of the directory service to use the whole file transfer scheme.

B.2 Performing an RPC

The procedure rpc (see figure B.2) encapsulates the directory servers' usual algorithm for performing a remote procedure call. For convenience all remote procedure calls take one argument and return one result; all errors are signalled by the single exception *fault*, together with an error code. These conventions are imposed by the language if a single procedure is going to encapsulate the RPC calling sequence. This removes similar code sequences from the various operations that occur between procedures. If the algorithm embodied in rpc is not satisfactory an operation need not use rpc, but instead provide its own algorithm for choosing the timeout and retries and execute the CALL statement directly.

```
rpc = PROC[arg_t, res_t:type](rp:r_proc, arg:arg_t, addr:network_address )
      RETURNS(res_t)
      SIGNALS(failed,fault(int))

    r_proc = REMOTEPROCTYPE(arg_t) RETURNS(res_t) SIGNALS(fault(int))

    attempts:int := 1
    WHILE attempts <= max_attempts DO
        RETURN ( CALL rp ( arg ) AT addr TIMEOUT 20 seconds ) EXCEPT
          WHEN soft_error(errno:int):
            % If remote site didn't even receive the call then
            % assume its dead.  All other soft errors are retryable.
            IF <failed to send call> THEN BREAK END
          WHEN hard_error(k:int):
            % No hard errors are retryable.
            SIGNAL failed
            END
        attempts := attempts + 1
        END RESIGNAL fault
    % Too many retries.
    SIGNAL failed
    END rpc
```

Figure B.2: The execution of a remote procedure call

B.3 Requests and Operations

Two implementations of the interface to the directory service are provided. One is a collection of remote procedures and the other is a process that decodes

incoming SSP requests. When a request is received it is passed to one of a pool of processes that is currently waiting for work to arrive. This ensures that use is made of the directory servers resources when some request is blocked, awaiting information from the file server or from another server, while at the same time giving a simple control structure.

The session identifier is used as a key into a table of known sessions registered at the server. In the prototype the key is, for convenience, the user identifier, but in a complete implementation this would need to be made secure by encryption or tagging with a random number from a large name space. To name objects in the directory graph, the clients pass absolute pathnames, that is the name is looked up starting at the root of the name graph. This yields the UID of the object that the operation is to be applied to.

Once the request parameters have been turned into UIDs the operation is performed. Any results or exceptions are passed back to the client; exceptions are turned into error codes if the request was received in the form of an SSP.

B.4 Testing

In testing the directory service it is necessary to cause failures of the servers to occur. To ensure these were in accordance with the fail/stop model it was necessary to be able to stop a server running at any moment. In the initial testing of the election algorithm this was achieved by reloading the operating system; as part of this process the resource manager causes the processor to perform a reset at any arbitrary moment. In the directory servers a faster method was employed. To simulate a failure an additional remote procedure call was provided that instructed the Mayflower kernel to stop scheduling the processes of the server and to unload the state and code. No opportunity is given to the processes to perform any clearing up actions.

To validate that the correct actions have been taken by the servers, the controller allows requests to be sent that return the current contents of the cache

of a named server. This, in conjunction with listing the contents of a particular directory, enabled the state of the directory graph and the cached copies to be discovered. It was necessary to have up to four machines, in addition to the controller, to allow for all possible exchanges of messages. Some operations, such as creating a new directory, involve two objects so two machines were needed to be primary sites for these objects. Further, one machine held a cached copy of the parent object and one did not. The final state of the operation could then be verified. In addition, tests were carried out that involved fewer machines, making one server perform more than one of the roles listed above, as well as tests that involved large numbers of concurrent requests.

C Client Interface

The Directory Service Interface

The following description of the operations available to clients of the directory service gives the arguments, results and those errors are specific to the operation. Other operations exist for the creation and removal of users and groups; these are available only to managers. Errors that commonly occur in the validation of the arguments have been omitted for clarity. For example, whenever a session identifier is required the error Unknown-Session-Id may be returned. Others include Insufficient-Access and Invalid-Path.

Add-Member	Add a new member to an ACL.
Arguments:	Path-Name:String, Member:String, Access:Int, Session-Id:Int
Results:	
Errors:	Unknown-Member
Delete-Member	Remove a member from an ACL.
Arguments:	Path-Name:String, Member:String, Access:Int, Session-Id:Int
Results:	
Errors:	Unknown-Member
List-Acl	Give the contents of an ACL.
Arguments:	Path-Name:String, Session-Id:Int
Results:	String
Errors:	
Create-Directory	Create a new directory and attach it to the naming graph.
Arguments:	Path-Name:String, Session-Id:Int
Results:	
Errors:	No-Parent, Duplicate-Entry

Rename	Change the last component of a name.
Arguments:	Path-Name1:String, Path-Name2:String, Session-Id:Int
Results:	
Errors:	No-Common-Parent, No-Rename-Of-Root
Fetch-File	Get a file sent (and possibly a token) from a file server.
Arguments:	Path-Name:String, Access:Int, Port:Int, Session-Id:Int
Results:	
Errors:	Object-Not-File, Already-Locked
Issue-Token	Create a new token for an object.
Arguments:	Path-Name:String, Machine:String, Session-Id:Int
Results:	Token
Errors:	Already-Locked
Create-Link	Form a new link to an object.
Arguments:	Path1:String, Path2:String, Overwrite:Bool, Session-Id:Int
Results:	
Errors:	Already-Exists, No-Such-Object, Path1-Invalid, Path2-Invalid
Unlink-Entry	Remove a link from an object.
Arguments:	Path-Name:String, Session-Id:Int
Results:	
Errors:	No-Such-Object, Invalid-Path
Register-User	Create a new session.
Arguments:	Name:String
Results:	Int
Errors:	Unknown-User
Add-User-To-Group	
Arguments:	New-User:String, Group-Name:String, Session-Id:Int
Results:	
Errors:	Already-Member, Unknown-New-Member, Unknown-Group
Remove-User-From-Group	
Arguments:	User-Name:String, Group-Name:String, Session-Id:Int
Results:	
Errors:	Not-In-Group, Unknown-Old-Member

References

[Bacon87] J M Bacon and K G Hamilton.
Distributed Computing with RPC: The Cambridge Approach.
Technical Report TR 117, University of Cambridge Computer Laboratory, October 1987.
To appear in Proc. IFIP Conference on Distributed Computing.

[Birrell80] A D Birrell and R M Needham.
A Universal File Server.
IEEE Transactions on Software Engineering, SE-6(5):450–453, September 1980.

[Black86] A P Black and E D Lazowska.
Interconnecting Heterogeneous Computer Systems.
In EUUG Conference Proceedings, pages 43–51, September 1986.

[Black87] A P Black, E D Lazowska, H M levy, D Notkin, J Sanislo and J Zahorjan.
Interconnecting Heterogeneous Computer Systems.
Technical Report 87-01-02, Dept. of Computer Science, Univ. of Washington, January 1987.

[Chor85] B Chor and B A Coan.
A Simple and Efficient Randomized Byzantine Agreement Algorithm.
IEEE Transactions on Software Engineering, SE-11(6):531–539, June 1985.

[Comer79] D Comer.
The Ubiquitous B-tree.
ACM Computing Surveys, 11(2):121–137, June 1979.

[Cooper85] R C B Cooper and K G Hamilton.
Preserving Abstraction in Concurrent Programming.
Technical Report TR 76, University of Cambridge Computer Laboratory, August 1985.

[Craft85] D H Craft.
Resource Management in a Distributed Computing System.
PhD thesis, Cambridge University, 1985.

[Dion80] J Dion.
The Cambridge File Server.
ACM Operating Systems Review, 14(4):26–35, October 1980.

[Dion81] J Dion.
Reliable Storage in a Local Area Network.
PhD thesis, Cambridge University, May 1981.

[Ellis83] C S Ellis and R Floyd.
The ROE File System.
Technical Report TR 119, Computer Science Dept., The University of Rochester, March 1983.

[Floyd86] R Floyd.
Directory Reference Patterns in a UNIX Environment.
Technical Report TR 179, Computer Science Dept., The University of Rochester, August 1986.

[Garret80] N H Garret and R M Needham.
An Asynchronous Garbage Collector for the Cambridge File Server.
ACM Operating Systems Review, 14(4):36–40, October 1980.

[Gifford79] D K Gifford.
Weighted Voting for Replicated Data.
In Proc. Seventh ACM Symposium on Operating Systems Principles, pages 150–162, October 1979.

[Girling82] C G Girling.
Object Representation on a Heterogeneous Network.
ACM Operating Systems Review, 16(4):49–59, October 1982.

[Hamilton84] K G Hamilton.
A Remote Procedure Call System.
PhD thesis, Cambridge University, 1984.

[ISO85a] International Standards Organization.
Encoding Rules for Abstract Syntax Notation One (ASN.1).
ISO TC97, ISO/DIS 8825, June 85.

[ISO85b] International Standards Organization.
Specification of Abstract Syntax Notation One (ASN.1).
ISO TC97, ISO/DIS 8824, June 85.

[Johnson80] M A Johnson.
Byte Stream Protocol.
Systems Reasearch Group, Cambridge University Computer Laboratory, April 1980.

[Kline86] C Kline.
Complete vs. Partial Transparency in LOCUS.
In EUUG Conference Proceedings, pages 5–14, September 1986.

[Lamport82] L Lamport, R Shostak and M Pease.
The Byzantine Generals Problem.
ACM Transactions on Programming Languages and Systems, 4(3):382–401, July 1982.

[Lampson71] B W Lampson.
Protection.
In Proc. Fifth Princeton Symposium on Information Sciences and Systems, pages 437–443, March 1971.
Reprinted in ACM Operating Systems Review, 8(1):18 – 24, January 1974.

[Lazowska86] E D Lazowska, J Zahorjan, D R Cheriton and W Zwaenepoel.
File Access Performance of Diskless Workstations.
ACM Computing Surveys, 4(3):238–268, August 1986.

[Leach82] P J Leach, B L Stumpf, J A Hamilton and P H Levine.
UIDs as Internal Names in a Distributed File System.
In Proc. of the Symposium on Principles of Distributed Computing, pages 34–41, August 1982.

[Leslie83] I M Leslie.
Extending the Local Area Network.
PhD thesis, Cambridge University, 1983.

[Liskov81] B Liskov, R Atkinson, T Bloom, E Moss, J Schaffert, R Scheifler and A Snyder.
CLU Reference Manual.
Springer-Verlag, 1981.

[McKusick84] M K McKusick, W N Joy, S J Leffler and R S Fabry.
A Fast File System for UNIX.
ACM Transactions on Computer Systems, 2(3):181–197, August 1984.

[Mullender84] S J Mullender and A S Tanenbaum.
The Design of a Capability-Based Distributed Operating System.
Technical Report CS-R8418, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1984.

[Mullender85] S J Mullender and A S Tanenbaum.
A Distributed File Service Based on Optimistic Concurrency Control.
In Proc. Tenth ACM Symposium on Operating Systems Principles, pages 51–62, October 1985.

[Needham82] R M Needham and A J Herbert.
The Cambridge Distributed Computing System.
International Computer Science Series, Addison-Wesley, London, 1982.

[Ody79] N J Ody.
A Protocol for "Single Shot" Ring Transactions.
Systems Reasearch Group, Cambridge University Computer Laboratory, April 1979.

[Pease80] M Pease, R Shostak and L Lamport.
Reaching Agreement in the Presence of Faults.
Journal of the ACM, 27(2):228–234, April 1980.

[Popek81] G Popek, B Walker, J Chow, D Edwards, C Kline, G Rudisin and G Thiel.
LOCUS: A Network Transparent, High Reliability Distributed System.
In Proc. Eighth Symposium on Operating Systems Principles, pages 169–177, December 1981.

[Redell74] D D Redell.
Naming and Protection in Extendible Operating Systems.
Technical Report MAC TR-140, Massachusetts Institute of Technology, November 1974.

[Richards79] M Richards, A R Aylward, P Bond, R D Evans and B J Knight.
TRIPOS - A Portable Operating System for Mini-computers.
Software Practice & Experience, 9(7):513–526, July 1979.

[Richards80] M Richards and C Whitby-Strevens.
BCPL - the language and its compiler.
Cambridge University Press, 1980.

[Richardson84] M F Richardson.
Filing System Services for Distributed Computing Systems.
PhD thesis, Cambridge University, 1984.

[Schroeder85] M D Schroeder, D K Gifford and R M Needham.
A Caching File System for a Programmer's Workstation.
In Proc. Tenth ACM Symposium on Operating Systems Principles, pages 25–34, December 1985.

[Sturgis80] H E Sturgis, J G Mitchell and J Israel.
Issues in the Design and Use of a Distributed File System.
ACM Operating Systems Review, 14(3):55–69, July 1980.

[Svobodova84] L Svobodova.
File Servers for Network-Based Distributed Systems.
ACM Computing Surveys, 16(4):353–398, December 1984.

[Swinehart79] D Swinehart, G McDaniel and D Boggs.
WFS: A Simple Shared File System for a Distributed Environment.
ACM Operating Systems Review, 13(5):9–17, November 1979.

[Walker83] B Walker, G Popek, R English, C Kline and G Thiel.
The LOCUS Distributed Operating System.
In Proc. Nineth ACM Symposium on Operating Systems Principles, pages 49–70, October 1983.

[Weinstein85] M Weinstein, T Page, B Livezey and G Popek.
Transactions and Synchronization in a Distributed Operating System.
In Proc. Tenth ACM Symposium on Operating Systems Principles, pages 115–126, December 1985.

[Wilkes79a] M V Wilkes and R M Needham.
The Cambridge CAP Computer and its Operating System.
The Computer Science Library, North Holland, 1979.

[Wilkes79b] M V Wilkes and D J Wheeler.
The Cambridge Digital Communication Ring.
In Proc. Local Area Communication Networks Symposium, Mitre and NBS, May 1979.

[Xerox81] Xerox Corporation.
Courier: The Remote Procedure Call Protocol.
Xerox System Integration Standard, XSIS 038112, December 1981.