Block-Based Distributed File Systems

Anthony J. McGregor

July 1997

Abstract

Distributed file systems have become popular because they allow information to be shared be between computers in a natural way. A distributed file system often forms a central building block in a distributed system.

Currently most distributed file systems are built using a communications interface that transfers messages about files between machines. This thesis proposes a different, lower level, communications interface. This 'block-based' interface exchanges information about the blocks that make up the file but not about the files themselves. No other distributed file system is built this way.

By demonstrating that a distributed file system can be implemented in a block-based manner, this thesis opens the way for many advances in distributed file systems. These include a reduction of the processing required at the server, uniformity in managing file blocks and fine-grained placement and replication of data. The simple communications model also lends itself to efficient implementation both at the server and in the communications protocols that support the interface. These advantages come at the cost of a more complex client implementation and the need for a lower level consistency mechanism.

A block-based distributed file system (BB-NFS) has been implemented. BB-NFS provides the Unix file system interface and demonstrates the feasibility and implementability of the block-based approach. Experience with the implementation lead to the development of a lock cache mechanism which gives a large improvement in the performance of the prototype. Although it has not been directly measured it is plausible that the prototype will perform better than the file based approach.

The block-based approach has much to offer future distributed file system developers. This thesis introduces the approach and its advantages, demonstrates its feasibility and shows that it can be implemented in a way that performs well.

Acknowledgments

Like most thesis students I am indebted to many people. The most direct and extensive help has come from my friends and colleagues in the Computer Science Department, especially my supervisors Professor John Cleary and Professor Mark Apperley. I am particularly grateful to John for accepting me as a research student even though I am not working directly in his current field of interest. John has the ability to go directly to the heart of what is important and I have found his advice on my practical work and the drafts of my thesis extremely helpful. I am also greatly indebted to Mark for encouraging me to start and for creating the circumstances where I could finish. Mark also read and commented on the final draft of the thesis, a major task for which I am very grateful.

Thank you too, to Paul Lyons, who also read the final draft of my thesis and made so many helpful comments and corrections. I greatly appreciate your encouragement and friendship.

It has amazed me that, despite the long period that it has taken for me to get to this point, my family have consistently encouraged me and gracefully accepted the many times when I have said "Sorry, I need to work on my thesis today." For Luke and Meaghan it will be a new experience to have a father who is not working on his thesis. To Susan what more can I say other than "I love you" and thank you.

To YHWH, my Lord and friend, you have sustained me through this endeavor and also through the many trials that have happened along the way. Thank you.

To everyone who has contributed to this thesis through your encouragement and by doing some of what I should otherwise have been doing, especially Andrew, Jim, Lloyd and Murray, thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction to Distributed Systems

The pioneers of computing, Babbage[102], von Neumann[11] and Turing[228], thought of computers as processing machines. However, with the advent of magnetic disks, computers took on the additional role of data storage and organisation. As a consequence of the need to organise data the first file systems were built in the mid 1960s[241].

In the late 1960s and early 1970s the spread of teleprocessing extended the role of the computer to include the sharing of information. Although computers have become more diverse and have increased in power and complexity over time the role of the computer as a tool for data processing, storage and sharing has remained unchanged.

Perhaps the most significant change in computer systems occurred in the mid 1970s when improvements in VLSI technology led to the production of personal computers. Through the 1980s and 1990s personal computers have grown in functionality and popularity. While they have created new markets, including the large home computer market, personal computers have also taken over many of the roles of earlier centralised systems.

Of particular impetus in the development of the personal computer market is the low cost and independence of personal computers. These factors allow the end-user to take control of the choice, purchase and operation of the computer system.

Early personal computers offered much less processing capacity than large centralised systems of the time but had a higher performance:price ratio. Because of this inverted 'economy of scale' it was possible to make more computing power available to end users. This particularly impacted the use of graphical user interfaces. Although GUIs were in use from the early 1960s[210] they did not find widespread acceptance until the early 1980s when low cost CPU capacity became available through personal computers.[202]

Currently the size of the personal computer market and the ability for consumers to update equipment regularly means that personal computers contain the most powerful components available. Higher performance computers are now designed with the same CPUs and disks that are used in personal computers, but using multiple parallel resources.

While personal computers have improved the processing and data storage capacity of computer systems they have detracted from the data sharing role of computers. The centralised architectures of the 1960s naturally supported data sharing but the decentralised nature of personal computers works against it. This often leads to 'islands' of data, isolated from one another.

It is the goal of distributed computing to bring together the resources of a group of computers in a way that supports sharing. Tanenbaum and van Renesse put it this way:

> "A distributed operating system is one that looks to its users like an ordinary centralized operating system but runs on multiple independent central processing units (CPUs)."[219]

Although computer networks have existed since 1969[62] they do not, by themselves, reach very far towards the goal of distributed computing. Paradoxically, while the existence of a suitable computer network is a requirement for a distributed system, a key role of the system software is to hide the presence of the network. If the distributed system is to "look to its users like an ordinary centralised operating system" the end users and managers of the system should be able to ignore the presence of the network.

Although the goal of distributed computing can be stated simply it is not easy to achieve, particularly if performance, security and integrity are included as part of what it means to look like a centralised system. Distributed systems research has been active for more than two decades and many distributed systems have been designed and implemented. Despite extensive and long term research, operating systems in current use only meet the goal to a limited extent.

There are many different distributed system architectures but a distributed file system is pivotal to many. This is because the distributed file system is normally the primary means of sharing data. Ideally a distributed file system allows data to be accessed in the same way whether it is stored locally or remotely. This service is useful to user applications and also to other other operating system functions. As an example of the latter consider a distributed system that moves a job from a heavily loaded processor to a lightly loaded one. This task is easier if the file system makes the data needed by the job available at the new location.

This introductory chapter introduces distributed file systems and discusses the main issues that make a distributed file system different from a local one. The following sections contain a discussion of the main problems that should be addressed by a distributed file system architecture. These issues are: transparency, security, integrity, scalability, openness, portability, heterogeneity, and performance.

1.1 Transparency

In line with Tanenbaum's definition of a distributed system (see page 2), most distributed systems authors place the need for transparency over all other goals.

Coulouris *et al.* list the following areas where the distributed nature of a system might be visible[56]:

- information location and access methods
- replication
- failure
- migration
- performance
- scaling
- concurrency

Transparency pervades many of the other problems that a distributed file system architecture needs to address. It is necessary, for example, to make data highly available (see below) but this must be done in a way that maintains transparency.

A few authors point out that transparency is not always desirable[27][174][28]. In some cases, they argue, an application programmer can take better advantage of a distributed system if transparency is not provided. Birman and Joseph give the example of a group of robot drills boring a set of holes[26]. In their example the group gets the job to do as a whole but each drill has some independence. What is required here, according to Birman and Joseph, is explicit communication and synchronisation, not a system that hides the presence of separate processors. The ISIS toolkit[28] provides mechanisms aimed at meeting these needs.

1.2 Availability

If a distributed file system is implemented using more than one computer system it has the potential for fault tolerance. If one computer fails others can, if suitably configured, take over the workload of the faulty system.

If the only computer that holds a copy of some data fails, that data will not be available even if other parts of the system are fault tolerant. To increase the availability of data it may be stored on more than one computer. This is known as replication.

Despite the possibility of fault tolerance many distributed file systems have lower availability than file systems implemented on centralised computer systems.

Leslie Lamport is quoted as having said:

> "A distributed system is a system in which the failure of a computer that one has never heard of can make it impossible to get work done."[136]

There are three main reasons why fault tolerance is difficult to achieve in a distributed file system:

- The network pervades the system and a failure of the network can stop all communication and consequently make all non-local resources unavailable.
- Replication of data carries a performance penalty and this reduces its usefulness. If data is not replicated and the system that stores that data becomes unavailable, read operations on the data cannot be serviced.
- Some faults might disrupt the behaviour of the system but not terminate it. In general a fault might cause any of the problems that can be caused by a deliberate attack. Such faults are known as Byzantine faults after the Byzantine generals problem defined by Lamport *et al.* [119].

1.3 Integrity

A distributed system must address the same integrity needs as a centralised system but there are two factors which make this more difficult in a distributed environment. First, the presence of more than one computer system introduces new ways for a distributed system to fail. Second, the distributed nature of the system makes it difficult or impossible to gain a consistent view of the whole system. These issues are discussed in this section.

As an example of a new failure mode consider the information that a Unix[1] file system and an application using the file system keep about each other. The application refers to a file by using a file descriptor handle (which was given to it, by the file system, when the file was opened). The file system maintains a file descriptor which includes the file identifier and the current read/write position in the file. Notice that both the application and the

[1]Unix is a trademark of AT&T Bell labs.

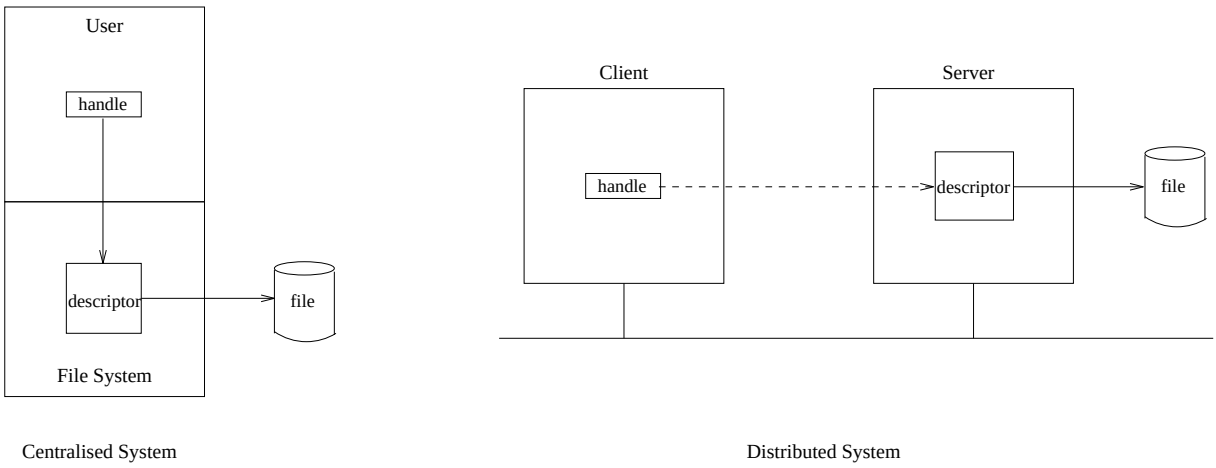


Figure 1.1: File Handles and Descriptors

file system keep information about the open file (the file handle and the file descriptor respectively).

If a Unix file system is implemented on a centralised system, both the application and the file system will lose this information in a system crash. If, however, the two are part of a distributed system and reside on different computers the failure of one will leave the information kept by the other intact. If the client fails then the server will indefinitely maintain a descriptor that cannot be used. If, on the other hand, the server fails, the association between the file descriptor handle and the file will no longer be valid. It might refer to no file or to another file. If the file descriptor does become associated with another file, later file accesses will be invalid. These are new problems not found in the centralised system.

Some distributed file systems such as NFS[141][194] are implemented in a way that does not require state information, like file descriptors, to be kept by the server and client. Nelson *et al.* [152] point out that statelessness has a negative impact on the performance and semantics of NFS and suggest that some state information needs to be kept at the cost of more complex error recovery.

A second integrity problem in distributed systems arises from the difficulty of obtaining a consistent view of the system. If the data of interest is spread over a number of computers and is changing on each system independently it may be impossible to implement some algorithms that are successful in a centralised system.

As an example consider the problem of deadlock[68] detection. Deadlock can be detected by constructing a 'wait-for graph' (see figure 1.2). The nodes of a wait-for graph are resources and processes. The arcs represent locks held by processes or those required for

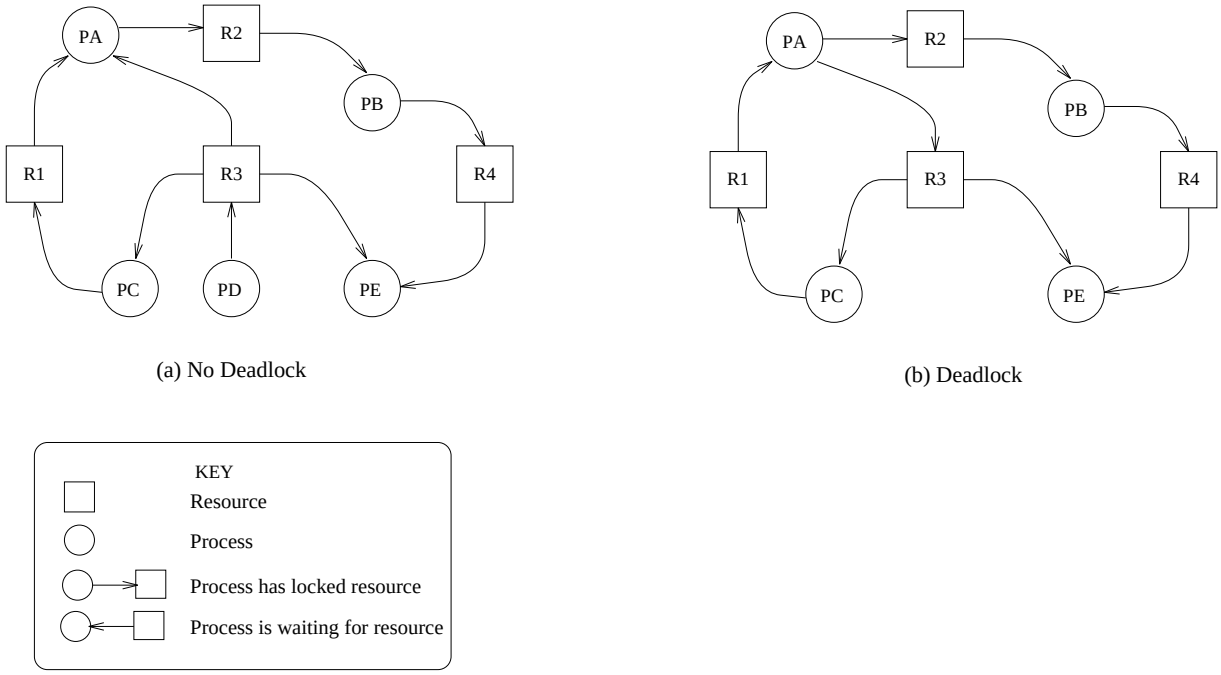


Figure 1.2: Deadlock detection using Wait-For Graphs

the process to proceed[99]. If there is a cycle in the graph then a deadlock exists.

In a centralised system the graph can be constructed as part of the lock allocation mechanism. However in a distributed system the resources and processes may be spread over a number of computers. When a lock is requested the construction of the wait-for graph requires collection of information from each system. It is possible that, while the information is being collected, other systems might request or release locks and change the graph. This can cause false detection of deadlock. Bernstein and Goodman elaborate on this problem and give some potential solutions[25].

1.4 Scalability

Distributed systems have the potential to increase in scale as the workload grows. If the number of users increases more computers can be added to the distributed system to support the increased workload.

However, no existing distributed system is able to be scaled up indefinitely. Scalability limitations arise when some component of the system must be involved in a fixed proportion of all work. Many systems are limited by the capacity of a single network. For example, SWALLOW[183] uses a network broadcast to notify storage sites of updates.

Systems which have been implemented on a large scale have the simplest semantics. For example Andrew[100], one of the largest distributed file systems, does not support coherence; it is possible for a user to read out-of-date information from a file.

Although indefinite scalability is probably not possible, some design techniques scale better than others. If a distributed file system is to scale well it should spread its workload as evenly as possible amongst all the resources available. Most distributed file systems do not spread their workload well. Many use the workstation/server model which requires all access to a collection of files to be processed by a single machine. Even high performance systems such as Bullet[233] are often designed this way. Although potentially one could balance the workload amongst a group of servers by noting which servers and files are heavily loaded and moving files to balance the load it has been found that this is difficult to do in practice[245][8].

1.5 Openness

In his "Operating Systems" paper in Communications of the ACM, Shaw contrasts open and closed systems:

> "Most earlier systems were *closed* systems in that a fixed set of services was supplied and new services could not be added easily. Many newer systems are *open*, offering extensibility features to users; at the extreme, an open system is just a basic kernel."[201]

Although openness is a desirable feature in a centralised system[122] it is more important still in a distributed system. Distributed systems are intended to bring together the computing and data resources of independent systems. Part of the reason end-user computing is popular is that it provides the flexibility for individual users to choose the type of system that most appropriately meets their needs. For a distributed system to be successful, it must make extra services available to its users without requiring them to forfeit the ability to make individual choices about the type and operation of machines they use.

1.6 Heterogeneity

The desire to meet the range of end-user needs with specialised operating system components leads to the goal of openness. It also leads to the need to support different

machine architectures within a single distributed system. The main issues in supporting heterogeneity are data representation and executable selection.

Computers vary in the way they represent data items in memory. A typical problem of this type is with the byte ordering of integers. In a non-distributed system files are explicitly moved from machine to machine and consequently representation issues need only be addressed during the file transfer or when data is imported into an application. In a distributed system, however, there are more points of interaction between systems and representation issues may need to be addressed at many points in the system's design.

A problem also arises if a file containing an executable program is used by a computer with a different machine code. This might occur in a distributed system if a single file system name space is implemented on a distributed system that supports different machine code architectures. For example Unix systems store executable code in the `/bin` directory. Implementation of a single name space will mean that there is a single `/bin` directory and all architectures will access the same executable files, even though they need to be different.

1.7 Security

Centralised systems can rely on the physical security of the hardware to stop unauthorised access to data. If the hardware is secure, the problem of limiting access is reduced to the problem of authenticating users and maintaining access control information.

By contrast, the hardware of a distributed system is not normally secure. It is likely to be physically dispersed around an organisation and may be the responsibility of a number of different managements. It may cover more than a single organisation. In addition the network, which is at the heart of the distributed system, is vulnerable to attack.

Voydock and Kent[238] classify security violations into three groups, according to the way security has been violated:

- unauthorised release of information
- modification of information, and
- denial of resource usage

Attacks come in many forms. Satyanarayanan[198] identifies the following security risks in a distributed file system: remote login attack, Trojan horse, stand-alone boot, eavesdropping, unauthorised access to backups, external release of security information (e.g.

a password), interactions between users on the same workstation, software faults, generating excessive network traffic, excessive server resource usage, use of workstations by unintended users, and cryptanalysis. This list is not exhaustive.

Although there is a significant body of distributed system security research, very few implemented distributed file systems address security in a thorough way. Most authors agree that building a secure system requires the use of data encryption as proposed by Needham and Schroeder[149][150]. The main reasons for the limited use of encryption are:

- Correct algorithms are difficult to design. The original Needham and Schroeder algorithm, published in Communications of the ACM, contained an error reported by Denning and Sacco[66].

  Needham and Schroeder comment:

  > "... [cryptographic] protocols such as those developed here are prone to extremely subtle errors that are unlikely to be detected in normal operation."[149]

- Unless special hardware is employed, encryption has a performance penalty.

- Encryption is less flexible that most security systems implemented on centralised systems. Reed and Svobodova[183] note that encryption only provides a single level of security. Potential users either know the key and can read *and* write the data or they do not know the key and cannot do either?[2]

1.8 Portability

For a distributed system to be generally useful it must support a wide range of applications. The most successful way of achieving this is for it to support a popular existing operating system interface. The Unix interface[187] is the most common interface supported by distributed file systems. LOCUS[174] and CHORUS[192] are examples of distributed file systems that support the Unix interface.

Interfaces designed for centralised systems often have shortcomings when used in the distributed environment. A number of distributed systems have extended the Unix interface to include more functionality. For example the Deceit[204] distributed file system addresses fault tolerance by extending NFS to support multiple storage sites for files. To do

[2] A solution to this problem is proposed in Chapter 5. See 'Security' on page 84.

this it extends the Unix file interface to allow the user to select the appropriate number of replicas.

Several aspects of the Unix interface are difficult to implement in a distributed file system. These are:

- The fork/exec procedure for starting a new process is inefficient in a distributed system[220].
- Unix allows a single file pointer (which specifies where data is to be read from or written to a file) to be shared between processes. This is difficult to do if the processes are on different computers[174].
- When a Unix file name is deleted, the data in the file must be retained while any users have the file open. This can be difficult to implement in a distributed system where the server and clients have only limited knowledge of each others state.
- The structure of a directory is visible to the application program. This limits flexibility[174].
- Some blocks are modified very frequently. For example the last access time of the root (/) directory changes every time any file is accessed.
- Some calls (e.g. stat) are used very frequently because they are cheap to implement on a centralised system. However these same calls can be expensive to implement in a distributed system[222].
- The strict consistency model is hard to implement in a distributed file system[126].

Despite these problems most distributed file systems support the Unix file interface or an approximation of it.

1.9 Performance

It is reasonable to expect that the large total processing capacity of many distributed systems should lead to a high performance system. In many respects this is the case. For example the CPU processing capacity that can be dedicated to a user interface is large compared with that available from a centralised system.

However in other areas distributed systems sometimes fail to deliver the expected performance benefits. In their ACM Computing Surveys paper Tanenbaum and van Renesse say:

> "There has been built more than one system requiring the full computing power of its machines just to run the protocols, leaving nothing over to do the work."[219]

There are two main reasons why it is difficult to achieve good performance in a distributed system:

- Data transfer, that in a centralised computer system takes place over the computers internal or IO bus, must occur over a network. Typically the raw transfer rate of a computer network is much lower than that of a bus. In addition many CPU cycles are required to implement the network protocols.

  The overhead of general purpose protocols has lead some distributed system researchers to believe it is necessary to built protocols customised to the needs of the distributed system being implemented. LOCUS[174], Swallow[183] and Amoeba[217] are some of the many distributed systems with their own custom protocols.[3]

- The synchronisation and consistency mechanisms required to coordinate the activities of the individual systems reduce performance.

  Consider a distributed file system that replicates data at different sites to achieve some fault tolerance. Correct operation of a system which employs replication normally requires a read operation to return the data most recently written to the system, irrespective of the replicas involved in the read and write operations. This is provided by a coherence mechanism. The coherence mechanism might involve copying the data to all sites when a write operation occurs. It is likely that there will be extra overhead associated with reliably sending the data to all machines.

There is a loose relationship between performance and scalability. A distributed system whose design leads it to perform well is likely to scale well. This is not always the case however. For example some systems implement replica consistency using a network broadcast (CHORUS[192] and Orca[17] are examples). Network broadcast performs well

[3] In Chapter 6 it is shown that the careful use of a general purpose protocol stack (UDP and IP from the 'TCP/IP' suite of protocols) can meet the requirements of a distributed file system without losing the interworking abilities of a widely implemented protocol.

providing the system is lightly loaded but limits scalability. The total capacity of a system employing broadcasts system cannot exceed the lower of:

- the ability of the network to carry the broadcast traffic
- the ability of the slowest attached system to process the broadcast workload

This chapter has introduced the main issues that need to be addressed by a distributed file system. The following chapters cover the role and implementation of a block-based network file system including a discussion of how it addresses these issues.

Chapter 2 describes the block-based distributed file system concept and the reasons for studying it. Chapter 3 contains background material useful in understanding the block-based network file system implementation. The need for consistency in a distributed file system and the relationship between coherence and user level consistency is discussed in detail. Chapter 4 provides some examples of existing distributed file systems. Chapter 5 describes an architecture suitable for building a block-based distributed file system. It ends with a table showing the relationship between the features of the architecture and the problems presented in this chapter.

Some of the most important features of the architecture presented in chapter 5 can be developed and tested in isolation from other requirements. Chapter 6 presents an implementation of a block-based network file system (BB-NFS) undertaken to prove the key concepts of the block-based distributed file system architecture.

A major concern about the block-based approach is the need for more network traffic to maintain synchronisation above the block level. Chapter 7 contains the results of measurements of the implemented locking mechanisms and quantifies the overhead introduced.

The thesis concludes with a summary of the results of this work and discusses the most important research areas for further development of the block based distributed file system concept.

Chapter 2

Block-Based Distributed File Systems

Chapter 1 introduced distributed systems with an emphasis on distributed file systems. This chapter introduces the idea that is central to the thesis, the block-based distributed file system. The chapter begins by introducing a taxonomy of distributed file systems based on the level of abstraction at which communication between systems occurs. The taxonomy is presented to place the notion of a block-based distributed file system in context. Following it, the reasons why a block-based distributed file system is interesting to study are given. The chapter ends with a discussion of the most important issues that should be addressed in a study of block-based distributed file systems.

2.1 Level of Distribution

Transparent sharing of data, despite the (potentially) different locations of the user and storage, is the primary goal of a distributed file system. To be able to share data requires communication. In a distributed system communication can be divided into two functions

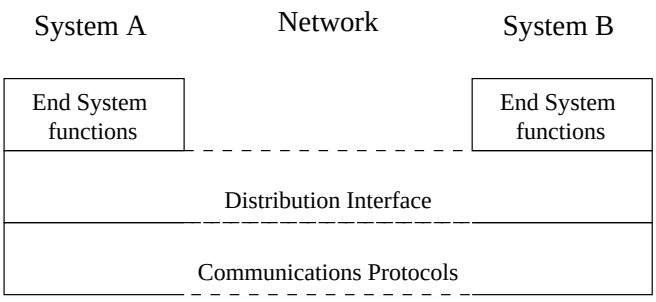


Figure 2.1: Distributed System Communication Functions

(see figure 2.1).

- The first function provides the reliable movement of data between machines. This ability is provided by the hardware and software of a computer network. Typical examples include the TCP/IP[54] and the lower layers of the OSI model[63]. Data communications has been, and continues to be, the subject of extensive research with the main aims of improving data rates, reducing latency and making management easier. See Tanenbaum[216] or Halsall[88] for an introduction to the field.

- The distribution interface in a distributed system makes use of reliable data transfer to share objects of some kind between computers. In this thesis the term 'distribution interface' is used to refer to this service within the distributed system. Implementing the distribution interface requires the cooperation of both computer systems, as coordinated by a protocol. Above this level operations are structured independently on each system without the need to consider how communication occurs between systems.

  For example, many distributed file systems have a distribution interface that supports reading data from a file. Above the level of the distribution interface a request for file data can be made without concern for the low level interactions between the local system and the system that stores the data.

  The distribution interface may be a concrete interface implemented as a distinguishable part of the distributed system. Alternatively it may be abstract, aiding in the discussion of the system, but not identifiable as a part of an implementation.

Distributed systems, like all operating systems, can be considered as a number of hierarchical layers. The distribution interface can be integrated into a distributed system at different levels of the storage hierarchy. The following sections show the different levels at which the distribution interface can be implemented.

Application Level

The highest level at which the distribution interface can be implemented is the application. In this case applications make use of data communications directly. This provides a high degree of flexibility but does not provide network transparency for the applications programmer. In a sense this is the trivial approach. The network services are not enhanced by the operating system. The socket mechanism, available in most implementations of the Unix operating system, is an example.

Application Level Objects

Some systems, particularly object oriented distributed systems, provide the applications programmer with a mechanism to make objects known and accessible across a number of systems. By making object methods accessible from a remote system the data encapsulated by the object is indirectly available. Clouds[60] is an example of an object based distributed system.

File Level

Most distributed file system distribution interfaces include the notion of a file. Andrew[100], Coda[112], Ficus[184] and FUSS[146] are all examples. In these systems, file operations are provided to higher layers by the distribution interface. The implementation of the distribution interface supports these file operations and is tied to the particular file abstraction supported.

Block Level

While many successful systems have been built at the application and file levels there are some difficulties with each. A block-based distribution interface supports only block[4] operations.

The block-based approach is central to this thesis. Chapter 6 describes BB-NFS which is a distributed file system that uses a block-based distribution interface. The interface itself is described in detail in Chapter 6, beginning on page 92. The main operations supported by the interface are reading and writing blocks. When a block is read a lock can be requested that guarantees exclusive access to the most recent version of the block's data.

Appendix A includes details of many other distributed file systems Although none of these systems employed a block-based distribution interface, a number are similar in some respects. The remainder of this section presents three other distribution interfaces that have some block-based aspects and their relationship to the block-based approach presented in this thesis.

[4]The terms *block* and *page* are often used interchangeably. They may refer to either a fixed sized unit of data transfered to or from a disk or a fixed sized subunit of the virtual memory address space. In this thesis the term block is used for the former and page for the latter.

Mixed

Many distributed file systems have a file or object level distribution interface but support block-by-block reading and writing. NFS[141][194] and the OSF Distributed File System (also known as Episode)[125] are examples of distributed file systems that operate in this way.

Remote Disk

Some early distributed file systems used a block-level distribution interface without locking. Remote Virtual Disk (RVD)[46] is an example of this type of system. These systems are simple to implement and support file systems which provide transparent access to information stored on a remote disk. However they only support a single client using each disk or read only file systems.

The limitation to a single client comes from the lack of consistency mechanisms. Even if the file system interface offers no consistency mechanisms to its users (as is the case with Unix) the file system itself requires a consistency mechanism for the protection of file meta-data. Consider the bit map of available blocks that most file systems maintain. This bit map is itself stored in disk blocks. If two clients attempt to modify the bitmap at about the same time there is the potential for the lost update problem[5] with respect to the modified bit map.

The simple remote disk distribution interface is not sufficiently powerful to avoid this problem. The key problem addressed by the block-based distributed file system implementation described in Chapter 6 is the identification and management of parts of the file system code that can cause the lost update problem in file meta-data.

Remote Disk Plus Locking

The distribution interface used in BB-NFS extends the functionality of the remote disk distribution interface to include locking facilities. Locks can then be used by the file system to ensure that updates to the file meta-data are applied one at a time and the lost update problem avoided.

One other distributed file system uses a similar approach. The VAX Cluster File

[5] The lost update problem is also known as the concurrent update problem and is described in Chapter 3 on page 37 and also in[25].

System[83] has a network disk distribution interface but also uses a distributed lock manager[205] to maintain consistency of file meta-data and to provide a consistency mechanism to file users.

The Vax cluster file system differs from the work described here in two respects:

- The distributed lock manager is not used at the block level. Instead it is applied to whole files, disk volumes and devices. When meta-data that relates to a single file is updated a lock is applied to the whole file. When the block bit maps are updated the the whole volume is locked. This coarse-level locking defeats some of the advantages of a block level interface.
- The Vax cluster file system uses customised hardware to support sharing. This limits its application.

2.2 Motivation

There are a number of advantages that might be gained by implementing a distributed file system on a block-based distribution interface. These are discussed in the following sections.

Fine grain

The fine granularity of the distribution interface means that the distributed file system has more flexibility when replicating and migrating data.

Coulouris *et al.* raise the following concern:

> "There are some features not found in current file services that will be important for the development of distributed applications in the future:
>
> *Support for fine-grained distribution of data:* As the sophistication of distributed applications grows, the sharing of small data items will become necessary. This is a reflection of the need to locate individual objects near the processes that are using them and to cache them individually in those locations. The file abstraction, which was developed as a model for permanent storage in centralised systems, doesn't address this need well. ..." [56]

By moving the distribution interface below the file level it is possible to implement fine

grained location and replication mechanisms while maintaining the widely used and understood file abstraction at a higher level.

Flexible

There are four competing models of data storage in distributed systems. These are the persistent object approach (as in Eden[4]), the one level persistent storage model (as in Apollo Domain[124]), the distributed database model (as in Cedar[37]) and the distributed file system model discussed in this thesis and typified by NFS[194]. There continues to be research in all these areas although research into the file system model appears to be particularly active at this time after less activity, in favor of other models, in the mid 1980s to early 1990s.

Although this thesis is primarily concerned with the application of a block level distribution interface to a file system, it is likely that the same interface could also be used to support the other approaches. This could allow them to share a common block level storage system and possibly to interwork through this mechanism.

Open

As noted in Chapter 1 (page 8), openness is an important characteristic of a distributed system. A block-based distribution interface allows the possibility for there to be more than one type of file system using the distributed block service. Although these file systems could each implement their own file abstraction they could also use a common storage pool and would not require preallocated storage resources. The common storage pool could also provide a basis for interworking the mechanisms.

The openness offered at the block level by a block-based distributed file system can be provided in addition to other forms of openness. If openness at the block level is not sufficiently powerful further openness can also be implemented at the file system level as is done in non-block-based systems. (See Chorus[9] for an example).

Idempotent

Block operations are naturally idempotent. That is they have the same effect even if repeated. For example, if a protocol error causes a block to be written twice instead of one the result is the same. This makes the distribution interface simpler and more robust.

In some existing distributed file systems correct operation, particularly in the presence of network failures, has been sacrificed for the sake of a simpler protocol (and consequently better performance). Juszczak explains how the lack of idempotence in NFS can lead to incorrect operation[107].

Simple Abstraction

The block level abstraction is simpler than the file level abstraction. This simplicity is balanced by more complex processing at the client. For example the BB-NFS distribution interface has 6 operations compared with NFS's 18. Many of the NFS operations are more complex that the block-based operations. (The NFS distribution interface is described in [141].)

Because of the simplicity of the block level distribution interface client requests require fewer server CPU cycles to process. In studies of the performance limits of distributed file systems (e.g. [100] and [197]) the availability of server CPU cycles is usually found to be the factor limiting distributed file system performance and scalability.

When discussing RVD (a block-based distributed file system that has no concurrency support and can therefore only support read only file systems) Treese notes:

> "Another effect of the block-level service is that a single VAX 11/750 server can support 75 client workstations with quite acceptable performance."[226]

More General Caching and Migration

Most existing distributed file systems enforce serialisation of file meta-data by managing all updates from a single site. This makes caching and migration of file meta-data difficult. Despite this, file meta-data exhibits locality of reference and is a good candidate for caching or migration[199].

Wang and Anderson[242] note that directory caching in existing distributed file systems is limited or ineffective and quote Sprite[152], NFS[194] and Andrew[100] as examples. Srinivasan and Mogul measured the network traffic after caching in Spritely NFS and found that more that 50% of the remote procedure calls were for directory operations[207].

A block level distribution interface does not make assumptions about the usage of the blocks it provides to the file system. Consequently it is free to cache and migrate all blocks,

not just data blocks. This does not mean that updates to file meta-data no longer need to be serialised. Serialisation in a block-based distributed file system must be managed by the software above the distribution interface. However the interface between systems occurs at a level where caching and migration can be addressed in a more general way and the solutions adopted are applicable to all blocks.

In most distributed file systems clients have spare capacity while servers form a bottleneck[121]. 'Serverless'[6] operation, at the file level, is a goal of xFS[8]. File meta-data is also distributed over a number of cooperating workstations by introducing an extra level of indirection in the file name mapping. Providing migration at the block level allows the principles of serverless distributed computing to be applied at a finer grain than a file. It also allows it to cover all file system blocks, including meta-data, with the one mechanism. Fine granularity is particularly important if the server supports some very large files, as might be the case with a database system.

Integration with Virtual Memory

Distributed virtual memory[156] (as first used in Accent[181]) continues to be an active area of distributed systems research. A block level distribution interface gives the potential to integrate the distributed file system and the distributed virtual memory systems.

Both virtual memory and the file system store data on disk. In many systems, including Unix, the two are implemented separately, making use of different disks or disk partitions. Competition between the file system and virtual memory is not always constructive[1] and some systems integrate the paging activity of virtual memory and the file system. Sprite[152], for example, does this by using the file system for paging. V[48] takes the opposite approach and maps files onto virtual memory.

Both approaches suffer from the limitation that they use an abstraction designed for one purpose to support a different function. For example if the file system is used for paging, the virtual memory system must pay for the ability for files to change size even though it does not need this facility.

A distributed block service allows problems common to both distributed virtual memory and the distributed file system (such as caching and consistency mechanisms) to be solved once without requiring one higher layer to use the other.

[6]The term serverless is a poor one because there are really many computers acting as servers. What is avoided is a single, centralised server. With respect to a single interaction between computers it is still convenient to consider the participants under the heading client and server.

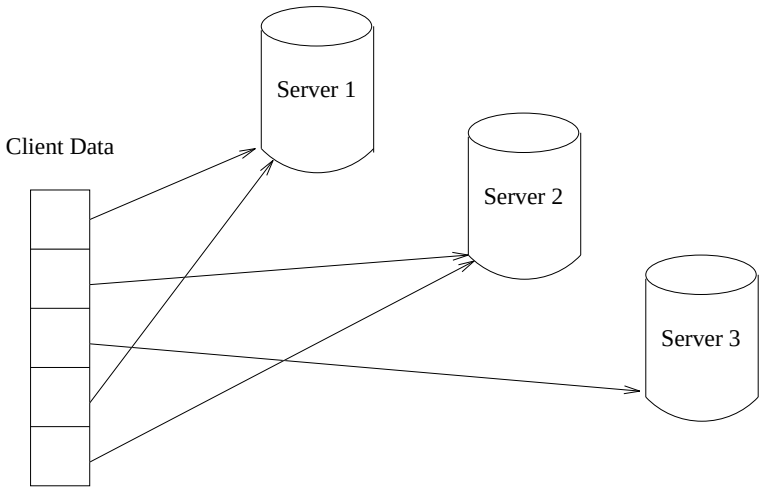


Figure 2.2: Server Striping

More Extensive Striping

A recent development in distributed file systems is the notion of striping. A striped distributed file system is the distributed analog of centralised RAID systems[168]. Instead of writing a file to a single server a striped system writes parts of the file to each of a number of servers (see figure 2.2). This spreads the load over multiple systems and allows performance that exceeds the capacity of a single server. See Zebra[92] for an example.

Because these systems stripe files they must manage file meta-data separately to ensure serialisation of updates. Zebra, for example, stores file meta-data on special file manager systems. This data is a critical resource and is not striped.

For reasons similar to those discussed under caching and migration (on page 21) a block level distribution interface allows all data, including meta-data, to be striped. This has the potential to make the benefits of striping available to meta-data as well as file data.

Simple Protocol Requirements

Most communications protocols are CPU intensive so simpler protocols will normally result in better performance and scalability. Distributed systems do not normally require the full functionality provided by general purpose protocols. The unneeded functionality and the introduction of extra layers of protocol adds overhead which increases communications latency. Achieving minimal communications latency is essential to good performance in a distributed system. Many authors (e.g. [217][109][211][174]) believe that the performance requirements of distributed systems cannot be meet by general purpose protocols includ-

ing the Internet (TCP/IP) protocol suite and the ISO OSI protocols . These authors argue for specialised protocols tailored to the distributed system they serve.

Protocol Needs

Because the block-based distribution interface is very simple, it can be supported by an even simpler protocol than most distributed system distribution interfaces. Some of the features found in general purpose protocols that are not required to implement the block-based distribution interface are:

Error Correction

Saltzer *et al.* [193] point out that low layers of a protocol stack cannot recover from all errors. As a consequence the higher layers must duplicate some, or all of the error recovery procedures provided by the lower layers. This is true for the block-based approach because the distribution interface protocol must recover from server or client crashes.

LANs and the newer high bandwidth WANs (such as ATM and SMDS) have very low, but non-zero, error rates. Correct recovery from those errors that do occur can be performed by the distribution interface protocol. The distribution interface layer must have a timeout mechanism to recover from system failures. This mechanism will also recover from communications errors providing the network detects and discards packets in error. Detecting and discarding packets with errors is normally provided by the hardware of a network interface and imposes very little overhead.

Virtual Circuits

Virtual circuits allow a network to more efficiently process large streams of data but require additional work at the start and end of the communication. The block-based approach is transaction-based not data-stream-based. This means that the overhead of connection establishment gives little gain and imposes an increased latency. In addition the state information required to manage open virtual circuits increases the complexity of recovery from failures.

Flow Control

Because the distribution interface is transaction based, interactions between the client and the server are inherently stop-and-wait. When the client makes a request of the server, the application needs the result of the request before it can continue. This is why low latency is particularly important in distributed file systems. If the file system has a single thread another request from this client will not be

sent to the server until it responds to the current request. This limits the flow of information and makes low level flow control mechanisms redundant. If there are multiple threads then more buffers at the server may be required. However the number of threads limits the maximum number of outstanding messages and inherently provides flow control.

Fragmentation and Reassembly

The exchange of blocks means that data is always transfered in fixed sized units that are able to fit in a single LAN packet for most current LANs. As a consequence, fragmentation and reassembly are not required.

Layered Implementation

The layered architecture of most communications protocols allows the implementor to avoid dealing with the full complexity of the communications system in a single piece of code. However, layering introduces additional overhead. Because the block-based distribution interface only requires a simple communications mechanism, little layering is needed in the implementation.

The simpler needs of a block-based distribution interface give two advantages. Firstly the removal of these functions allows a simple and very efficient protocol implementation. The second advantage is more subtle. As is demonstrated by the implementation described in Chapter 6 the simpler needs of the block-based distribution interface allow a standard protocol suite (the TCP/IP protocol suite in that case) to be used in an efficient way (see 'protocol' on page 106). This allows the internetworking benefits of standardisation without the performance penalties normally inherent in a general purpose protocol.

2.3 Questions and Difficulties

The most obvious difficulty when implementing a distributed file system on top of a block level distribution interface is the need to ensure that updates to file meta-data operate in a consistent manner. The following sections contain a discussion of this and other issues that might have an impact on the feasibility of a block-based distributed file system.

Concurrency Control

Most distributed file systems serialise updates to their meta-data by making a single entity responsible for meta-data updates. A block level distribution interface does not support this approach because all file system blocks must be treated in the same way. A block-based distributed file system must use another concurrency control mechanism such as locking[73] or an optimistic[117] approach. BB-NFS employs locking to serialise meta-data updates.

As explained in the sections above there are a number of advantages gained by placing the responsibility for consistent updates with the client system rather than a server. These advantages include a reduction in the server workload and more extensive caching, replication and migration.

Although it is clear that a locking scheme can be used to ensure consistency, two questions about the suitability of locking need to be answered. The first is what the performance implications of locking are. In particular the taking and returning of locks requires communication between the client and the server. It is not immediately obvious how much extra communication will be involved and what impact this extra communication will have on the performance of the file system.

The second question is whether locking can be implemented with a reasonable effort. This concern arises because concurrent programming, especially concurrent programming within the operating system kernel, is difficult. Many authors have commented on this (see, for example, [60], [174], and [148]).

Popek *et al.* write:

> "As real distributed systems come into existence, an unpleasant truth is being learned: the development of software for distributed applications is often far harder to design, implement, debug and maintain then than the analogous application written for a centralized system"[174]

Whole File Behaviour

A number of file system behaviour studies have shown that files are often accessed as whole units. Ousterhout *et al.* [161] carried out a very influential performance study. They analysed three Unix systems at the University of California at Berkeley. One of their findings was that about 70% of all file accesses are whole file accesses. There have been a number of followup studies by different researchers showing similar results, including[16] and [58].

These papers, particularly the Ousterhout paper, are widely referenced and are the basis for many decisions about distributed file systems. Howard *et al.* quote whole file behaviour as a guiding principle in the design of Andrew[100]. The Andrew Benchmark (described in [100] but also used by other distributed file system researches e.g. [208]) is based around whole file operations.

Although the block level abstraction provides block operations it does not prohibit whole file transfers. A file system can retrieve the whole file a block at a time. Retrieving a file a block at a time has different performance implications from retrieving it as a single operation. These implications are discussed under the headings fragmentation, latency and locking. This section concludes with a review of some more recent research that suggests that whole file behaviour may be an artifact of the academic Unix environment and is less common in commercial systems.

Fragmentation

It might appear that whole file transfers incur a penalty when implemented on top of a block-based distribution interface because the file must be transfered a block at a time rather than in one piece. It should be noted however data transfer is performed in small pieces in all distributed file systems. There are two reasons for this. Data transfer is limited by the size of the protocol data unit (PDU) of the underlying network. In the case of Ethernet[140] PDUs may be up to 1500 bytes. Many other networks have similar sizes.

The second reason for fragmentation is because files are stored on disks in small fixed sized sectors.

When a whole file transfer is requested as a single operation (as it is in Andrew for example) fragmentation (and reassembly of the fragments) occurs in at least three places:

- Sectors are read off the disk and assembled into the file. The file is passed to the communications system.
- The communications system fragments the file and sends it to the client as a stream of PDUs.
- The PDUs are received and reassembled into the original file.

If the file is not able to be stored and processed in memory it must be written to disk at the client and this will involve re-fragmenting it.

By contrast a whole file transfer implemented on top of a block-based distribution interface is able to read blocks from the disk and transmit them to the client without fragmentation or reassembly provided that a block can be transmitted in a single network PDU.[7] If the file is to be used in memory at the client the blocks must be reassembled into a single file. However if the file is transfered to the client's disk no reassembly is required. The difference between the approaches arises because the same abstraction is used for disk storage and communication in the block-based approach, whereas a conversion between formats is required in the whole file transfer approach.

Latency

Like fragmentation, latency occurs at a number of points in a file transfer.

The main points are:

- The time required to assemble and transfer the request to the server.
- The time required for the server to receive the request and to (possibly repeatedly)
  - request a disk block
  - wait for the disk subsystem to retrieve the block and place it in memory.
- The time required to formulate and transfer the reply to the client.

[7] This is the case for the implementation described in Chapter 6.

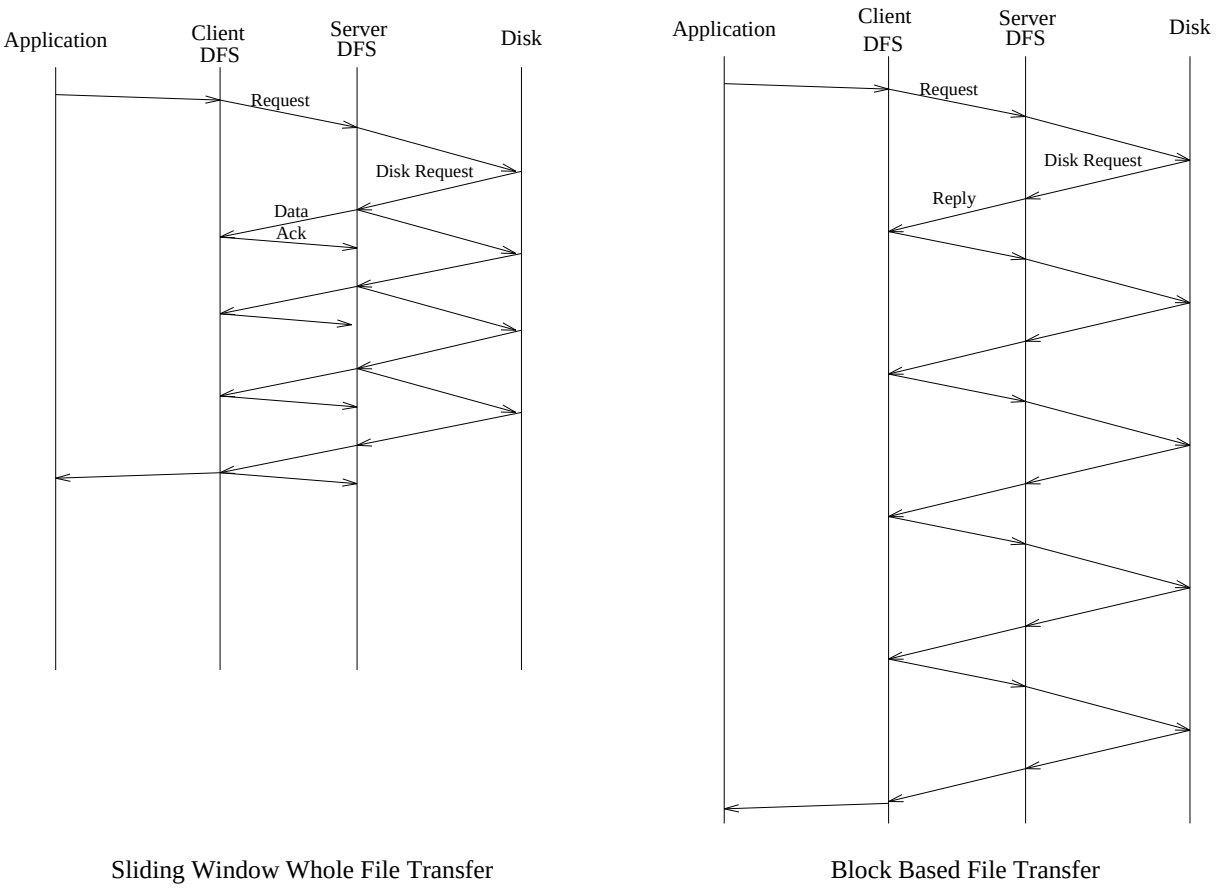


Figure 2.3: File Transfer Latency

There is not a simple relationship between these times and the total latency. Depending on the implementation some may overlap while others will be cumulative.

Figure 2.3 shows the way the latencies combine with a whole file transfer using a sliding window protocol[8] (a) and a block-based transfer (b). Various alternative implementations are not shown in the diagram including:

- **Whole file transfer using a stop-and-wait[9]protocol**. In this case there is no overlap between the acknowledgment and the subsequent disk request. As a consequence, the latency becomes similar to that of the block-based transfer.
- **Read ahead**: Many file systems fetch more than a single block at a time. If the file system supports multiple concurrent block requests, read-ahead allows a degree of overlap by requesting the next block before a reply has been received to the previous block request.
- **Overlapped IO**: If the implementation of the whole file transfer illustrated in

[8]Sliding window protocols are also known as busy-RQ protocols.
[9]Stop-and-wait protocols are also know as idle-RQ protocols.

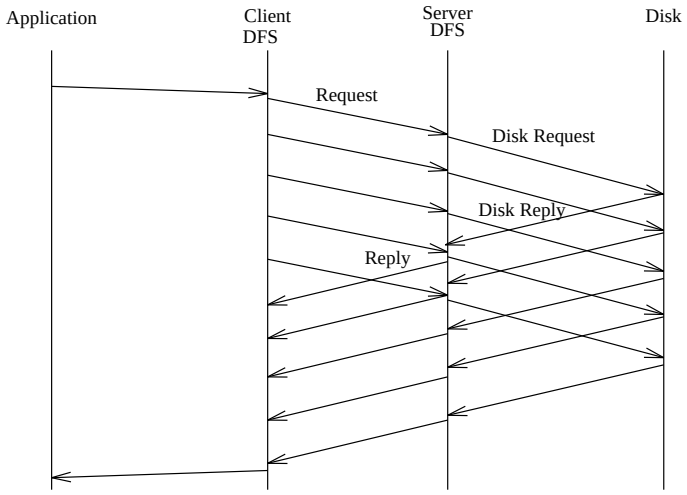


Figure 2.4: Block-Based File Transfer with Overlapped IO

figure 2.3(a) supported overlapped disk and network IO, much of the time between the end of one disk request and the start of the next could be removed.

In a similar way the latency between block requests might be removed when using a block-based distribution interface to implement a whole file transfer as shown in figure 2.4.

- **Fixed Length Files**: In some systems files are fixed in length or immutable. If this is the case the disk management routines can be optimised to store the blocks of the file in a series of disk sectors that maximizes the transfer rate from the disk.

In summary, whole file transfer in a block-based distributed file system involves a different set of latencies from those found in whole file transfer implemented using a file level distribution interface. The total latency for either approach depends more on the implementation details than on the distribution interface. It should be possible to build implementations that have low latencies with either approach.

The relative length of the network and disk latencies are important in this discussion. Although the exact lengths of these latencies vary from device to device, disk latencies tend to be much higher than network latencies. This difference is likely to be exacerbated in the future because new technology is expected to decrease network latencies more than disk latencies. The more disk latencies outweigh network latencies the less significance the implementation of the distribution interface takes in performance. Based on a combination of measurement and calculation Dahlin *et al.* quote the latencies given in table 2.1[59].

Ethernet	6660μs
155Mbit/s ATM Network	800μs
Disk access	14,800μs

Table 2.1: Latency of System Components

Locking

Locking is often used in a distributed file system to support synchronisation or serialisability. If whole file behaviour is assumed, a file level distribution interface can support a single lock for the whole file. Because a block-based distribution interface does not include the notion of a file, it can not directly support the locking of a whole file in a single operation.

A file system could lock the whole file by locking all the blocks in the file. If a separate network operation was required to lock each block this could cause a significant performance penalty. However in some cases locking operations can be piggy-backed with data transfer. This approach is discussed more fully in Chapter 6 (Implementation). Block-by-block locking has the advantage that it can be applied to whole files or just the blocks of the file that need to be protected from concurrent access.

If necessary whole file locking can be implemented in a single network transaction even when a block-based distribution interface is used. This can be done by locking a block containing some essential file meta-data or by using the block locking mechanism to implement a set of locks specifically for this purpose.

Other Environments

From the preceding discussion it seems that implementation details play a significant role in the whole file behaviour of a block-based distributed file system. It is one of the aims of the practical work described in Chapter 6 to determine whether a simple distributed file system implemented on a block-based distribution interface supports whole file transfers well and, if necessary, to explore what steps are needed to achieve satisfactory performance.

There is, however, some evidence to suggest that the whole file nature of file transfers has been over-emphasised. The studies that suggest whole file transfer is dominant are based on the academic Unix environment. Biswas[32][31] studied file activity from two 'Fortune-100' companies and found that:

> "Random access files are, on an average, about three times larger in size than sequential files and account for a larger amount of file system activity." [32]

Whole file assumptions can limit the functionality of a distributed file system. In describing Andrew, Howard *et al.* point out that, because of its whole file nature, Andrew will not support distributed databases well[100]. If Biswas' results are widely applicable these limits make whole file transfer much less desirable.

Security

As discussed in Chapter 1 (page 9) security is more difficult in distributed systems than in centralised operating systems. One of the advantages of the block-based approach is the ability to implement serverless systems (see 'More General Caching and Migration' on page 21). Security is even more difficult in a block-based serverless distributed file system than it is in most distributed file systems because there are no centralised resources. Most distributed file systems rely on the security of a centralised resource as the basis for security in the distributed file system. Andrew, for example, enforces security by having physically secure servers and using authentication and encryption to secure communication with a particular user.[10]

The distributed file system architecture presented in Chapter 5 includes a proposal for implementing a secure serverless block-based distributed file system based on encryption and message digests.

Naming

Naming is the process of mapping one identifier to another. Usually a high level identifier, like a file name, is mapped to a low level one like a number that uniquely identifies the file. In a distributed system, naming and location transparency are often performed by the same process. That is, when a location independent name is translated by the naming system it produces a location dependent identifier.

In a block-based distributed file system it is desirable to have the block identifiers that are location independent. If this can be achieved migration of blocks becomes much easier. Unfortunately naming mechanisms intended for use at the file level are unlikely to be suitable at the block level because they will not perform well. It might be acceptable to

[10] See the description of the Andrew File System in Chapter 4 (beginning on page 53) or [198] for more detail.

perform several disk and network transactions as part of a file name translation that only occurs when a file is opened. However if even a single disk transaction is required to map each block name to its low level identifier there will be a serious performance penalty.

The use of caching and/or hints[123] appears to have the potential to address this problem. Naming is discussed further in the architecture of Chapter 5.

Support for Heterogeneous Systems

A block-based distributed file system builds its meta-data structures in the blocks stored by the block-based distribution interface. The precise structure will depend on the nature and implementation of the file system and will not be compatible with another file system.

It might be expected that this will limit the ability to support a range of computer architectures within a single distributed system. However this is not the case. A distributed file system that uses a file level distribution interface must convert between the local file representation and the representation supported by the distribution interface for all distributed file operations. This can also be done in a block-based system if an implementor chooses. Conversion to a common file structure can be performed by all systems using a block-based distribution interface. This will achieve the same degree of interworking as is possible with a file based distribution interface.

Other interworking options, which lie between the extremes of no conversion and converting all file accesses to a common format are also available in a block-based systems. These include:

- Conversion to a common structure only for files that are to be transfered between dissimilar systems.
- An ISO OSI presentation layer[105][104] style negotiation of the transfer format. This would allow similar machines to select a convenient structure rather than being limited to just a single one.

2.4 Summary

The discussion in this chapter has given a number of potential benefits of the block-based approach and highlighted some areas of concern. On careful examination, most concerns can be met using similar mechanisms in the block-based approach as in other distributed

file systems. This is not surprising because the higher level mechanisms are, for the most part, still employed in a block-based system.

It was noted, however, that careful implementation may be required to reduce the latency that repeated stop-and-wait block transfers might introduce. The issue of concurrency control requires further discussion and is the topic of the next chapter.

Chapter 3

Consistency and Synchronisation

Chapters 1 and 2 introduced distributed file systems and explored the problems and potential benefits associated with a distributed file system implemented on a block-based distribution interface. This chapter addresses a key component of a block based distributed file system, concurrency control. Concurrency control is required for the file system to maintain consistent data structures. Chapter 5 draws on this material in its description of the architecture of a block-based file system.

Consistency control is necessary because it is desirable to keep multiple copies of information in a distributed system. The simplest distributed systems do not keep copies but provide access to data stored on other computers by extending existing mechanisms to use remote data. In many cases this approach is unsatisfactory because of the delays imposed by accessing data via the network. In V, for example, it takes nearly 4 times as long to read a remote 1kb block as it does to access a local block[48]. If a single byte is required, remote access is two to four *orders of magnitude* slower than local access[18].

Simple remote access may also be unable the meet availability requirements. As noted earlier (see 'Availability' in Chapter 1, page 4) the cooperation of multiple computers via a network introduces additional points of failure in a distributed system. This can lower the availability of the service offered by the distributed system. Finally, simple remote access does not scale well because all operations must be processed by the server.

These problems can be reduced by caching, which is used to improve performance, or by replication which is used to improve reliability. Although caching and replication both entail keeping copies of data they differ in motivation, storage location and duration and are usually addressed separately in distributed systems (see table 3.1).

When more than one copy of a data item is stored the problem of maintaining consistency

Similarities	Differences
Multiple copies of data are kept.	Replication is long term while caching is short term.
Consistency mechanism required.	Caching raises performance, replication often lowers it.
Introduces new failure modes.	Caches are hierarchical, replicas may be of equal status.

Table 3.1: Caching and Replication

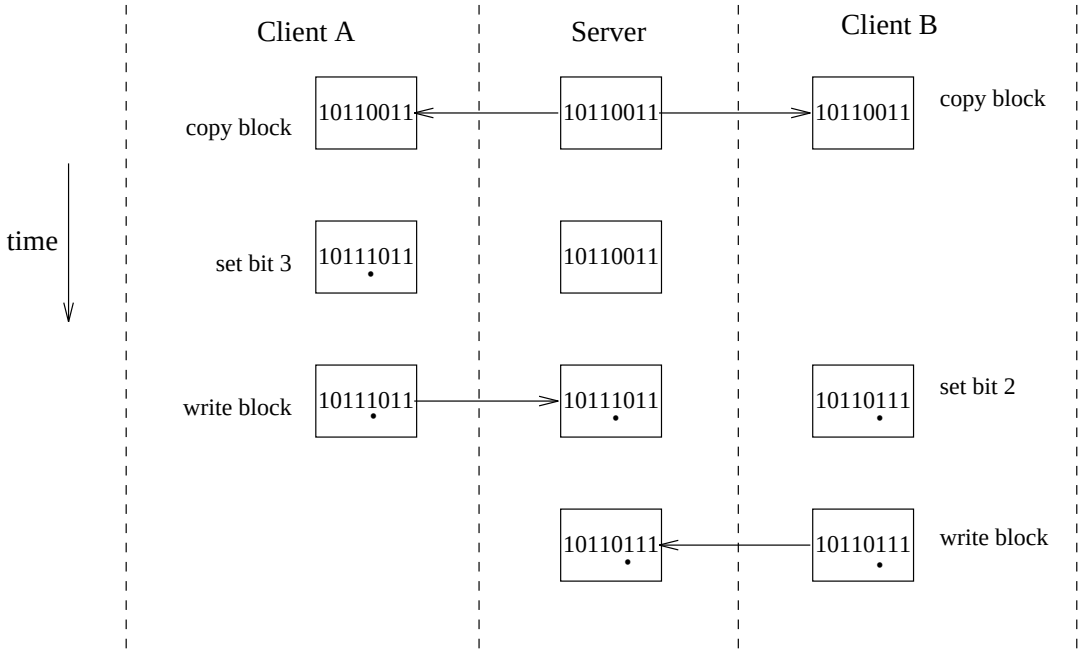


Figure 3.1: Lost Update Problem

between the copies arises. Bernstein and Goodman discuss consistency (in the context of distributed databases) in detail[25]. If no consistency mechanism is implemented, three consistency problems can occur:

- **Lost Update**: When two updates are executed concurrently, sometimes one of the updates is lost as shown in figure 3.1. An update is lost because both updates operate on a separate copy of the same initial data. The result of the transaction that completes first is overwritten by the result of the update that finishes last.

- **Inconsistent View**: When an update transaction and a summary transaction execute in parallel partial results from the update can cause an incorrect result for the summary transaction. Bernstein and Goodman give the example shown in figure 3.2. The diagram shows a transfer of money between two accounts that occurs concurrently with a balance enquiry for the pair of accounts.

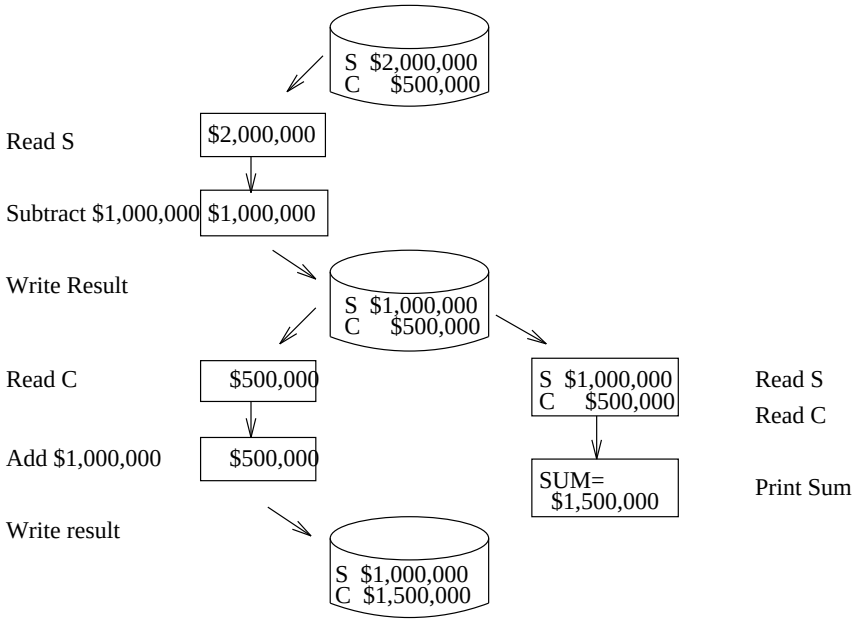


Figure 3.2: Inconsistent View
(adapted from [25])

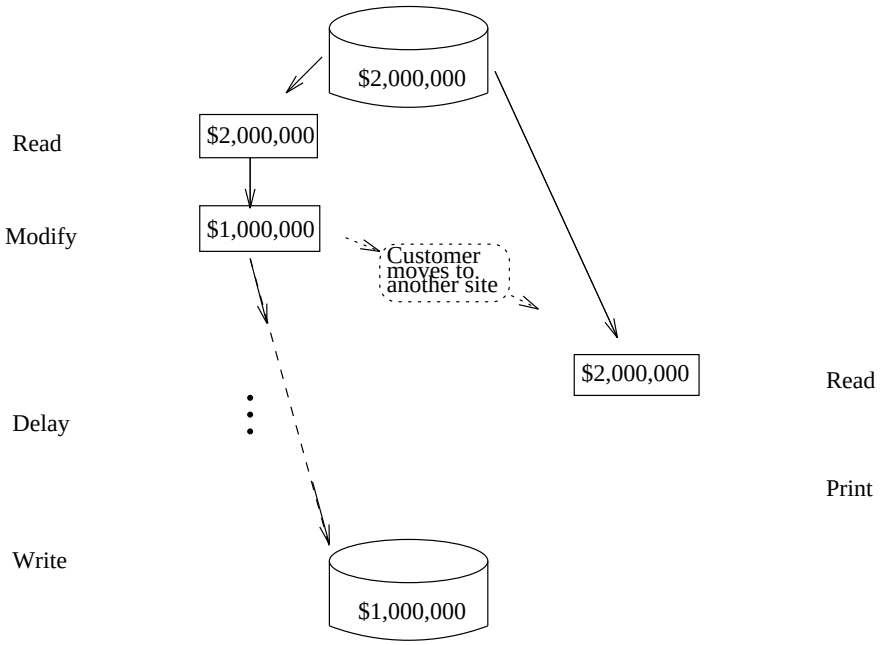


Figure 3.3: Obsolete View

- **Obsolete View**: Obsolete data can be returned if copies of data becomes out of date. This might happen momentarily during the update process or for a longer period if a failure occurs that blocks the update process. An example is shown in figure 3.3. The diagram shows a balance update which is concurrent with a balance enquiry because the update write back is delayed. A customer who is aware of the earlier transaction is delivered obsolete information. The obsolete view problem differs from the other two in that its definition is a matter of degree. If the delay in figure 3.3 is small, or if two users who were unaware of the order of their requests, the result could be acceptable. In most cases if the users of the system are unaware of the order of events then either result is acceptable. It is only when some other information flow defines the ordering that an error has occurred.

In discussions of distributed file system design these problems are often only discussed in the context of cache consistency. However they can occur whenever copies of data are kept. This includes when data is cached, when it is replicated, when copies of data are held by applications and when data is copied from memory into CPU registers.

3.1 Strict Coherence

Having transparency as a primary goal (see page 4) has led many distributed systems researchers to implement coherent systems. A coherent system is one where read operations always return the most recently written data. That is, obsolete data is never returned. Coherence is a natural artifact of centralised and simple remote access systems and is also known as one-copy semantics.

Coherence Algorithms

A number of cache coherence mechanisms have been designed. This section introduces the different approaches. In the next chapter examples of how coherence is managed in particular distributed file systems are provided, along with other details of those system.

The following approaches to maintaining coherence have been used:

- single cache
- central directory
- shared directory

- the snoopy cache
- locking
- token
- lease
- voting
- migration

Single cache

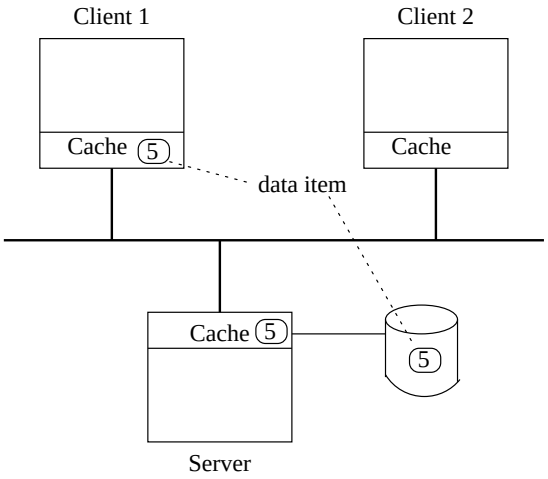
In a centralised system replication of data in a cache and the on the disk does not introduce the possibility of inconsistent views of the data because all requests pass through the cache and consequently have access to the most recent data. This approach can also be used in a distributed system if only the server maintains a cache. If all client read and write operations are passed to the server, coherence is a natural result.

There are costs associated with network transactions. The network introduces extra latency and can also act as a limit to scalability. These concerns were discussed under the heading 'Latency' in Chapter 1 on page 29. To address these concerns many distributed systems cache data at both the clients and the server. Client caching introduces the possibility for stale data to be maintained in one or more client caches after a write operation occurs (see figure 3.4).

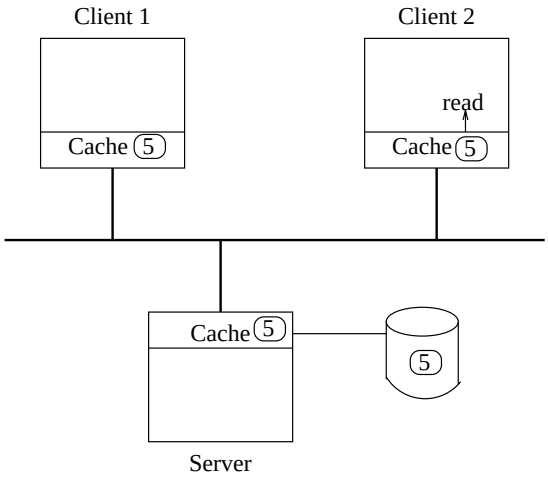
Central Directory

Tang proposed the first cache coherence mechanism (for tightly coupled multiprocessors) at the 1976 AFIPS National Conference[221]. Tang's solution (see figure 3.5) interposes a *'storage controller'* between the client caches and main memory. The storage controller maintains a directory of items in the caches. Cached data items are classified as either shared or private. Shared items are those that are not being written by any processor and may exist in more than one cache. Private items are those that are being written by a processor. They may only exist in a single cache.

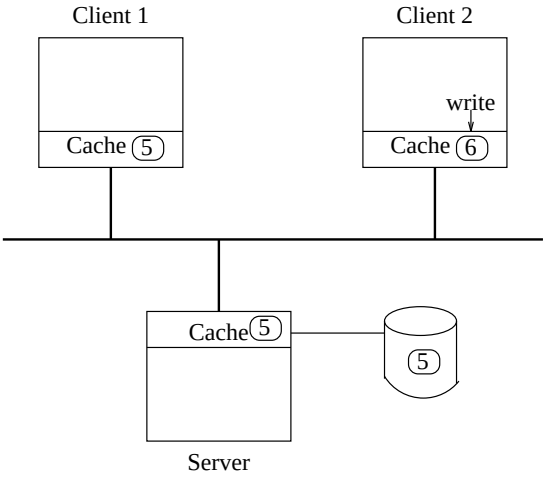
When a processor is about to perform a write operation the cache controller checks the ownership of the block and converts it to a private block if necessary. Conversion involves informing other systems which have the block cached that their copies are no longer valid. The process of a server contacting a client is the reverse of the normal operation of the client/server paradigm. Because of this change of role the mechanism is known as call-back.



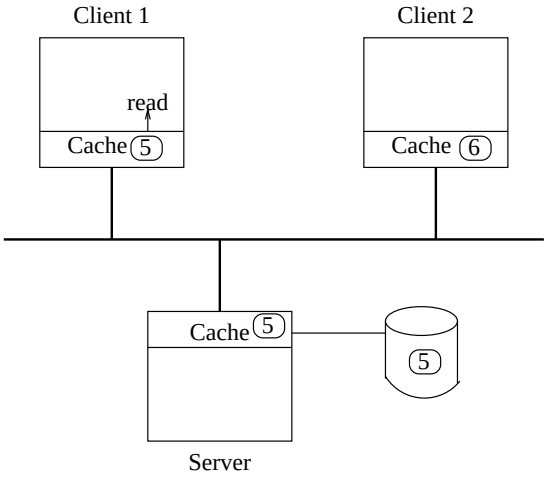
Client 1 has a copy cached



Client 2 reads data and caches it



Client 2 modifies the data



Client 1 reads stale data

Figure 3.4: Stale Cache

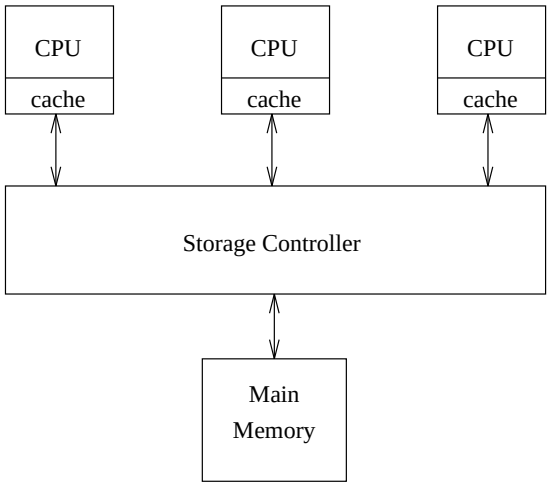


Figure 3.5: Tang's Solution

Although Tang proposed his algorithm for shared memory multiprocessors it is archetypical of central directory coherence mechanisms. Such mechanisms have been used in both tightly and loosely coupled distributed systems to address coherence in memory and disk caches. The Sprite distributed file system described in the next chapter is an example of a distributed file system using a central directory coherence mechanism.

Distributed Directory

Client/server architectures have a home site for data. This site is an appropriate location for a central directory. However in some forms of data replication all storage sites have equal status. In this case there is no obvious location for a central directory.

An alternative approach is to distribute the directory. Censier and Feautrier[45] proposed the first design of this type, known as the presence bit algorithm, by modifying Tang's algorithm. Solutions of this kind store the sites of replicas at each storage site. In the Censier and Feautrier algorithm this is implemented using a bitmap but other implementations are possible. The Ficus distributed file system[87] discussed in the next chapter uses both complete and partial replica lists to implement optimistic concurrency control.

Snoopy cache

If write operations can be monitored by all systems that hold a copy of a data item, as is the case when a central bus or broadcast network is used, then a snoopy cache can be implemented. A snoopy cache monitors all write operations and, when a remote write to data stored in the cache is detected, the cache is either updated or its copy discarded.

To function correctly a snoopy cache implementation must see write operations to data it caches. If data items in all caches are to be updated when a write occurs then all data must be written through the local cache so that the write operations will be seen by the other caches. If an invalidate policy is used then only the first write after replication needs to be written through. CHORUS[192] and Orca[17] are examples of snoopy cache coherent systems.

Locking

Some systems, particularly distributed database systems, use locks to limit the replication of data in a way that enforces coherence. Very simple systems allow only a single client replica of data. Any holder of a replica of the data can safely read or write the data. More sophisticated systems allow a number of levels of locking with mechanisms to change one type of lock to another. Most systems provide at least three types of locking.

1. **No locking**: Used when coherence is not required.
2. **Read-Read Lock**: Used when multiple sites access the data in a read-only way. Multiple replicas of the data are allowed.
3. **Read-Write Lock**: Used when a site will modify the data. Only a single replica is allowed.

Some systems provide more types. The VAX/VMS distributed lock manager[205], for example, provides six types of lock: null lock, concurrent read, concurrent write, protected read, protected write and exclusive.

Lock based coherence must be transaction oriented. That is, a system needs to bracket read and write operations with appropriate management (locking) operations. This requires some knowledge of the way that the data will be used before it is used. As a consequence locking can only be used in those systems where this knowledge is available. The Clouds[60] distributed system (discussed in the next chapter) uses lock based coherence.

Token

In environments where there is no home site for the data item it may be difficult to choose a site to manage locks. Tokens can provide the same functionality as locking but may be managed in either a centralised or a distributed way.

A token is an abstract attribute assigned to a replica storage site. A read token gives permission to hold an unmodified copy of the data while a write token gives permission to hold a modified version. Other tokens are possible if different degrees of control are required. The token management protocol ensures that tokens are generated, destroyed and passed from site to site in a way that ensures coherence.

Because of the distributed nature of token management a token recovery protocol may be needed to regenerate tokens lost because of site failures or protocol errors. Ar[136] and OSF's DFS[125] use tokens to implement coherence.

Migration

In some systems the home site of the data is moved in response to a request for update access. This is similar to a token mechanism but ownership of the data is the token. By contrast with most token systems the data is not returned to its previous location unless future accesses are made to it from that location. xFS[242] is an example of a system that does this.

Leases

Locks and tokens can become deadlocked. If the system that is currently holding the lock or token fails, the resource protected by the lock or token will never become available for use unless a recovery protocol is implemented. In a similar way the contract granted by a server when it issues a lock or a token could be broken if the server loses state information when it crashes and later recovers.

Gray and Cheriton[85] proposed a locking mechanism, which they called a lease, where locks are granted for a limited time. If the lease is not explicitly revoked within its lifetime, it is implicitly released when its lifetime expires. Leases make crash recovery simpler because neither the client or the server expect the lease to last indefinitely. When a client fails, the leases it holds are naturally released when they time out. When a server fails and then recovers again it can recover its lost state from lease extensions requested by clients providing that it does not immediately grant new leases.

The BSD 4.4 implementation of NFS[133] uses leases to maintain coherence, as does the file system discussed in Chapter 6.

Voting

If replication is used to improve the availability of the system it is desirable to be able to continue processing with an arbitrary subset of replicas. If coherence is also required a voting mechanism can be used.

Thomas proposed the use of a majority of sites for each operation[223]. Each time a write or a read operation is performed the data must be read from or written to, more than half of the replicas. The use of a majority ensures that any read operation must have at least one replica in common with the most recent write operation but does not require all or any particular copies to be available. Either time stamps[118] or version numbers[123] can be used to identify the latest version of the data.

Gifford[80] extended Thomas' algorithm to make it more flexible. In Gifford's algorithm some replicas can be given priority over others by giving replicas a different number of votes. This might be desirable because they are faster or more reliable. The algorithm can also be used to trade the cost of read operations against the cost of write operations by increasing the minimum number of votes required for one operation and reducing the number required for the other.

There are many other variants on voting including:

- **missing-partition dynamic voting** [101], which adjusts the vote assignments to accommodate network failures so that availability is maximized.

- **voting with witnesses** [164], which allows some replicas to be replaced with records of the current state of the file (called witnesses).
- **voting with bystanders** [165], which allows voting to be used with very small numbers of voters.
- **voting with ghosts** [231] which allows updates to continue even when failure of some storage sites means that a quorum can not be reached.
- **multi-dimensional voting** [50], which extends the concept of a vote to a vector and consequently increases flexibility of voting.

Deceit[203] and Echo[97] are examples of distributed file systems that use voting.

Limitations of Strict Coherence

The previous sections presented mechanisms for maintaining strict coherence. Although coherence does make a distributed system behave like a centralised one with respect to replicated data it does not totally solve the problem of consistency. The problems that remain, performance, availability and application level consistency, are discussed in the next sections.

Performance

Implementing a coherence mechanism requires a high level of communication and consequently increases the latency of cache operations. Mullender and Tanenbaum, in arguing for optimistic concurrency control, write:

> "[Caches are essential.] However, caches and concurrency control mechanisms are likely to become enemies: the administration of the caches in a rapidly changing environment can cause more inefficiency that not keeping caches at all."[146]

Reducing this impact is a major motivation for weakened coherence mechanisms.

Coherence and Availability

Coherence mechanisms require cooperation between a number of computer systems. Because of this, they work against fault tolerance, even though this is one of the goals of

replication. Consider a data item replicated on four different systems that uses majority consensus to guarantee coherence. If two of these systems are unreachable a consensus can not be gained and the data is unavailable, even though there are two copies of it that are accessible.

Application Level Consistency

The problem of consistency is wider than the distributed file system. Even if a coherent cache is provided, concurrency errors, like the lost update problem, can still occur because applications copy data from the cache into their working space to manipulate it. It is possible for more than one application to have a copy of the data at the same time. If consistency problems (such as lost update, inconsistent view and obsolete view which are discussed on page 37) are to be avoided, applications must coordinate their activities through a synchronisation mechanism, even if a coherent cache is provided.

The normal solution to this in databases is to use two phase locking (2PL)[25] and in operating systems and other concurrent programs it is to use a synchronisation mechanism like semaphores[68] or monitors[98].

3.2 User Coherence

Given that cache coherence does not ensure consistency at the application level a question that arises is "Does coherence provide any useful function?" Coherence does contribute to the consistency of some systems in the following ways:

- It provides a basis on which an application level synchronisation mechanism can be built. In particular it means that when an application that is coordinating its operations with other applications performs a write other applications are guaranteed to see the data written on their next attempt to read it.
- Even if applications, and the users of the applications, do not require strictly coherent operation they will expect read operations to return data previously written when they are aware of the ordering of events.

  As an example consider two users who are editing the same file in a Unix environment. If each user is unaware of the precise ordering of their operations and those of the other user non-coherent behaviour appears correct. If, for example, one of them saves the file a moment before the other reads it the lack of coordination between

the users makes non-coherent behaviour acceptable. Since the users themselves do not know the order of the save and read operations they can not be aware of the lack of coherence in the file system. If, however, the same two users perform the same operations in the same order but the time scale is such that they are aware of the order of the operations non-coherent behaviour will be seen as an error.

Strict coherence meets these needs but at a high cost. In this thesis the term *user coherence* is used to mean that, in so far as users are able to determine, a read request returns the most recently written data. In some cases user coherence can be implemented at a lower cost than strict coherence.

3.3 Weakened Coherence

Some systems do not provide strict coherence but some weaker, and less costly, form of coherence. Other systems avoid the need for coherence by defining different file system semantics. The following sections provide an introduction to the most common alternatives to strict coherence which are:

- cache validity check
- optimistic transactions
- immutable objects
- disconnected optimistic
- update on close
- application supported

Cache validity check

Some distributed file systems limit the window during which non-coherence can occur by checking the validity of the content of caches either regularly or before data is used. NFS[194] is an example of this kind of system.

A cache validity check implements user coherence provided that:

a) the check is done with a frequency that ensures users who communicate independently of the file system do so on a longer time scale that the validity check

b) if the operating system interface offers synchronisation mechanisms these mechanisms trigger a cache check

If either of these conditions does not hold it is possible for users to be aware of the presence of stale data in the cache.

NFS fails on the second count. Although NFS, like Unix, provides very limited support for synchronisation, file creation and deletion can be used to provide a primitive service of this kind. File creation and deletion do not trigger a cache validity check. In some environments, where there is rapid communication outside the file system, it may also fail on the first count.

Optimistic Transactions

Transaction-based systems allow operations to proceed on the assumption that they will be successful but, if a failure occurs, the transaction can be aborted with no effect. Most transaction systems use locking to avoid concurrent update of data items. Some, however, allow transactions to proceed without locking data but instead check when the results of the transaction are being written that no other transaction has altered the data. If the data has been modified the transaction is aborted and re-executed with fresh copies of the data.

Although this technique was originally applied to database systems it has also been used in distributed file systems. For an example see FUSS[146]. Optimistic transactions provide user coherence for update operations but not necessarily for read operations.

Session Semantics

Some systems, notably Andrew (described in the next chapter), accumulate changes to a file at the client until the file is closed. At the time the file is closed changes are written back to the server and become visible to other clients. In essence this approach is similar to a cache validity check but the potential window of non-coherence is longer and indefinite. Systems that provide session semantics do not offer user coherence.

Immutable objects

Some systems avoid the problem of coherence by making files unchangeable. Copies of immutable objects can be cached without concern for coherence because they will not be changed. Swallow[183] and the Cedar file system[81] are examples.

If a file system provides only immutable files many applications will need to discover the newest version of a file. It is common for an immutable-file-based distributed file system to also provide a version number mechanism, including a mechanism to locate the latest version of the file. In this case the need for coherence is moved to the version numbering system.

Disconnected Optimistic

The growth in portable computers has led to a new way of working for some users. These users sometimes connect their computers to a network (perhaps at their workplace) and sometimes operate disconnected from the network. To support this type of usage there is a need for file systems that can operate with periods of disconnection.

The Coda[112] and Ficus[94] file systems support disconnected operation through extensive caching and optimistic concurrency control. In this context an optimistic update policy means that updates to a single copy are allowed even if other copies are not available. When a client with modified files is reconnected to the network, changes made to files are propagated to other replicas. Disconnected operation makes the potential window of inconsistency the extent of the disconnection. There is a risk that two users update the same file during disconnected operation and consequently one user's updates can be

lost.

By itself disconnected optimistic access does not provide user coherence. However with the addition of a version vector mechanism (discussed in the next section) it may do.

Version Vectors

Some file systems support a no-lost-updates policy even when operating with some periods of disconnection. Version vectors, in the style of Locus[167] can provide a means of doing this.

A version vector is stored at each replica site and contains a count of the number of updates applied to each replica. Normally all counts will be the same unless an update is in progress. During a network partition or disconnected operation, some replicas may be unavailable in some partitions and the counts in the vector will differ. When the network is repaired the version vectors are compared. If every element of one vector is the same or greater than the equivalent element of another vector then no conflict has occurred (but some updates are still to be applied to the replica with the smaller version vector).

If an update conflict has occurred two resolutions are possible:

- If the semantics of the update are well known it may be possible for the system to automatically resolve the conflict. For example a lost update to a file system directory can be re-applied when the conflict is discovered.
- If the system does not have an automatic resolution mechanism human intervention is required to resolve the conflict.

In the case where automatic conflict resolution is possible, user coherence is achieved, provided that the unresolved conflicting data has not been made available to users.

Application supported

Some systems couple the mechanism for providing application level consistency and that for providing coherence. This allows coherence to be guaranteed when applications co-ordinate their activities but not at other times. For example, if locks are used, when an application locks a data item, coherence is provided on that item. Other items that are not locked are not maintained in a coherent way. This approach limits the cost of

coherence to those read and write operations that explicitly require it. Other operations do not pay the cost of maintaining coherence.

User coherence is provided if all communication between users occurs through the application synchronisation mechanisms or if it is coupled with another mechanism (validity check for example) that reduces the window of inconsistency so that it is sufficiently small that other forms of communication see what appears to be a coherent service.

Apollo domain[124] uses application-supported coherence, which integrates object locking and coherence.

3.4 Summary

Consistency is a widely studied problem in distributed systems. The presence of so many different implementations suggests that choosing a universal consistency mechanism is probably not possible. Instead a mechanism that meets the particular needs of the system under consideration should be chosen.

At the beginning of this chapter, it was noted that consistency is a key concern for the implementation of a block-based distributed file system. Strict consistency is required when file system meta-data is updated. Consistency may also be needed for the data stored in files but the type of consistency required will depend on the application that is using the files.

Consistency will be discussed again in Chapter 5 where a consistency mechanism is chosen for the block-based distributed file system architecture, in Chapter 6 where details of the implementation of the consistency mechanism are given and finally in Chapter 7 where the performance of the consistency mechanism is discussed.

Chapter 4

Distributed File System Survey

It is useful to consider the concepts discussed in the preceding chapters in the light of concrete examples. This chapter contains a review of some existing distributed file systems. The review focuses on the distribution interface and consistency mechanisms of the systems. This chapter is also intended to provide further context for the block-based distributed file system concept by considering other existing distributed file systems.

Because there are more than 80 distributed file systems reported in the literature this section only contains a review of some of them. The selection is intended to show a variety of approaches rather than being complete or representative. The annotated bibliography in appendix A covers many more distributed file systems but at less depth.

The distributed file systems covered here are: The Andrew File System, Bullet, Clouds, Ficus, Sprite, NFS and Zebra.

4.1 The Andrew File System

Design of the Andrew File System (AFS)[100][197][199] began in 1983 at Carnegie Mellon University. During its development AFS has had a number of name changes. Initially it was known as ITC then it then went through a number of generations identified as AFS-1 to AFS-4. AFS-4 was adopted by the Open Software Foundation (OSF) where it is known as Decorum[110] and Episode[52].

The Coda file system[112] is also based on AFS. Coda is designed to support file systems which have long periods of disconnected operation as might be expected where part of the file system resides on a portable computer.

Goals

The main goals of AFS's developers were:

- **Large Scale**: 5000 – 10000 workstations should be supported by AFS.
- **Location Independence**: Any user should be able to use any workstation. Consistent with this goal AFS supports a single shared name space.
- **Security**: AFS is used in the university environment to support student and research users and should provide adequate security for this sometimes hostile environment.
- **Performance**: AFS was designed to replace existing centralised systems. Performance was required to at least equal the performance of the previous systems. Further, users should not feel the need to place files at particular sites for performance reasons.
- **Integrity**: The probability of file loss should not exceed that of the previous centralised systems.
- **Heterogeneity**: It should be simple to add a new type of workstation.

AFS was not designed to support large databases and the original design is only suitable for files of a few megabytes or smaller.

Design

Figure 4.1 shows that AFS is split into a network of servers, called *Vice,* and the workstations that use the servers. Each client has an interface process called *Virtue*[11]. The workstations run a modified version of Unix with the file name space divided into two areas, a local name space and a shared name space. The local name space is used for temporary and system administration files including those files that give a workstation its identity. Most files, including the users personal files are stored in the shared name space implemented by Vice.

The shared name space provides location independence. Security is managed by making Vice physically secure and using encryption for all transfers between Vice and Virtue[198]. Integrity and reliability are provided by replicating files within Vice. Heterogeneity is

[11]In some versions of AFS Virtue is known as Venus

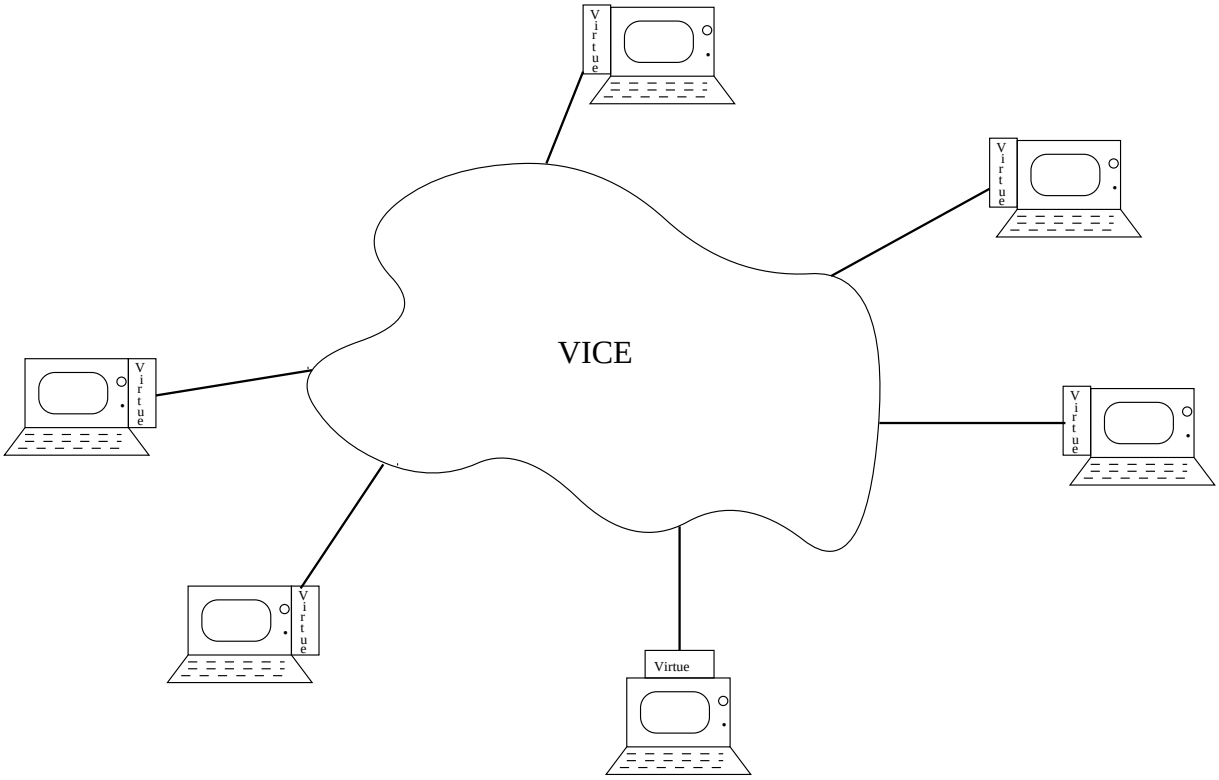


Figure 4.1: AFS

supported by placing symbolic links from the executable directories in the local name space to the appropriate executables in the shared name space. Scalability and performance are addressed through the simple design and the use of caching.

Caching and consistency

When an application program running on a users workstation opens a file in the shared name space Virtue locates and copies the file from a server to the workstations local disk. The file is then opened and processed in the same way as a local file. When a file that has been modified is closed it is written back to Vice. Notice that AFS transfers files as whole entries.[12]

The file is left on the local disk as long as space permits so that if it is required again it does not need to be fetched from Vice. Before Virtue reuses a file from the local disk cache it checks the validity of the file with Virtue. This mechanism only provides session semantics as described in Chapter 3 (page 50). Coherence and protection against the lost update and inconsistent view problems are not provided by the file system.

[12]Whole file transfer was seen as a cornerstone of the original AFS design[197]. Later AFS designs modified this approach to allow 64kb blocks (called pages in AFS literature) to be transfered[199].

Distribution Interface

In addition to file fetch and store operations the distribution interface also contains file creation, deletion and call back operations.

Experience

The AFS design is simple and has met many of the goals set by its designers. In some areas, however, experience has lead to changes in the design. The implementation of ITS showed that there was a significant cost associated with repeated cache validation when a file is opened many times by the same client. In later versions of AFS, Vice maintains state information and uses a call back to notify Virtue of changes to the files that it caches. The validity check mechanism is retained but used less frequently so that lost call backs do not cause out of date files to be retained indefinitely by clients. This change means that the window of inconsistency is increased beyond that of session semantics if there is a failure which defeats the call-back mechanism.

1	MakeDir	Constructs a target subtree that is identical in structure to the source subtree
2	Copy	Copies every file from the source subtree to the target
3	ScanDir	Recursively traverses the target subtree and examines the status of every file in it. It does not actually read the contents of any file.
4	ReadAll	Scans every byte of every file in the target subtree once.
5	Make	Compiles and links all the files in the target subtree.

Table 4.1: Phases of the Andrew Benchmark
(extracted from [100])

Whole file caching was not found to be wholly suitable and later versions of AFS modified this procedure by fetching only a 64Kb block of the file rather than the whole file. To be able to implement this, the AFS code was moved out of the Unix standard IO library into the Unix kernel.

The Andrew Benchmark

To test the performance of Andrew against other distributed file systems Howard *et al.* created a synthetic workload; the Andrew Benchmark. The benchmark consists of five phases which operate on a source tree of about 70 files. The phases are shown in table 4.1. Most of the work generated by this benchmark comes from whole file operations. Analysis of the results of Howard *et al.*[100] show that when the benchmarks were run on local file systems of three different Unix systems, with different amounts of memory, approximately 91% of the time consumed was by whole file operations (Copy, ReadAll and Make).

Although Howard *et al.* explicitly say that there is no demonstrated statistical similarity between the benchmark and the workload in any real system the Andrew Benchmark has been used in many file system performance studies (e.g. [208], [160], [152] and [100]). It is probable that whole file behaviour is a characteristic of academic, Unix systems but is not typical of many systems. As Howard notes, whole file operations are not satisfactory for file systems that support large databases. Some studies (e.g. Biswas[32][31]) indicate that non-sequential access is the predominant type of file system behaviour in computer systems supporting commercial activity. Whole file behaviour is discussed at more length in Chapter 2 beginning on page 27.

Summary

AFS is the largest scale and one of the best known distributed file systems. Strong support for scalability has lead to very weak file consistency which approximates session semantics when there are few network or system failures.

AFS was designed and tested under the assumption that whole file transfers provide the only significant workload. It does not address the needs of distributed databases or other large randomly accessed files. Data is always transferred from disk to disk. AFS will perform poorly if only small parts of large files are accessed by applications.

Dahlin *et al.*[58] point out that the client server nature of AFS limits its ultimate scalability because the server must manage all cache misses, call backs and directory modifications. The size of the server state and file location index may also limit scalability[242].

4.2 Bullet

The Bullet file server[233][220][235] is one of a number of distributed file systems built for the Amoeba distributed operating system[217][147] which was designed and implemented at Vrije Universiteit in Amsterdam, The Netherlands.

Goals

Bullet was designed and implemented to test the feasibility and performance of a distributed file system that does not fragment files into separately placed disk blocks. In Bullet files are stored on disk and used as whole entities, not as a collection of linked blocks. The Ousterhout *et al.* study[161] and the success of ITC[197] are quoted as motivating factors for using this approach.

Other design goals were:

- **Scalability**: Both geographic scalability, allowing Amoeba to operate in a WAN context, and scalability in terms of the number of workstations were desired.
- **Availability**: Support for replication was included.

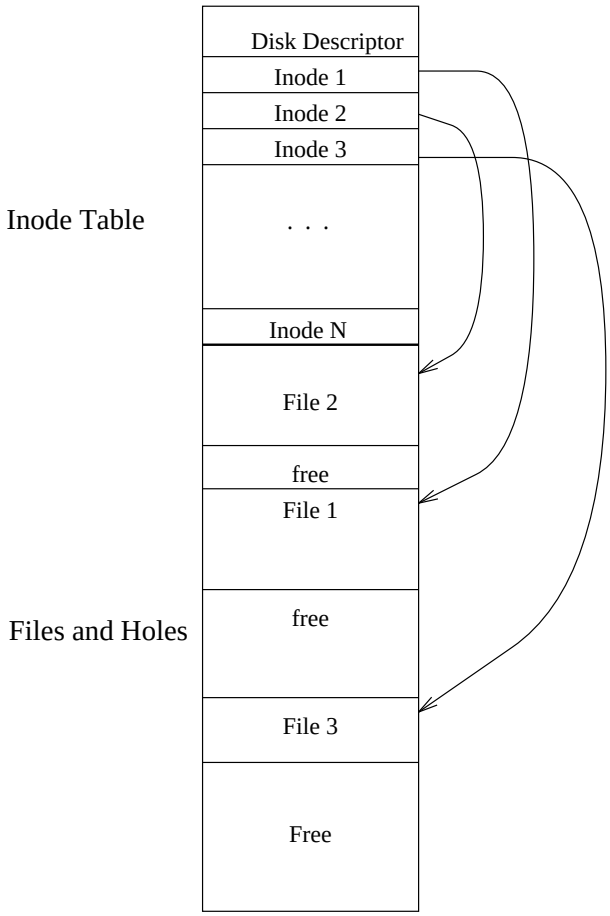


Figure 4.2: Bullet Disk Layout
(from [233])

1	create	Create a new file, returns a capability for the file
2	size	Returns the size of a file
3	read	Returns the contents of a file
4	delete	Deletes a file

Table 4.2: Bullet Distribution Interface

Design

Files are stored by the server in contiguous disk blocks as shown in figure 4.2. As in the rest of Amoeba files are identified by capabilities [218]. Capabilities, which are in essence large unique identifiers, combine location, naming and authentication into a single mechanism.

A parameter in the file creation operation specifies how many copies of the file should be stored. Each copy is stored on a different disk. If this parameter is set to 0 then the create operation returns as soon as the file has been saved in the server's RAM but possibly before it has been written to the server's disk.

Caching and consistency

Files stored on the server are immutable. A version number mechanism is provided to identify different versions of the same file. Because files are immutable they can be cached as needed without the possibility for different copies becoming inconsistent. A check must still be made to ensure that the file still exists and to check that the file is the most recent version, if that is required. These comparisons are made by comparing the capability of the cached file with the one held by the directory service.

Distribution Interface

Because Bullet only supports operations on whole files and the files can not change it has a very simple distribution interface. The Bullet distribution interface has just four operations as listed in table 4.2.

Experience

Tests indicate that Bullet outperforms NFS by approximately a factor of 3[233]. Most of this performance increase is probably from the higher disk bandwidth achieved by using contiguous disk blocks.

Summary

Although Bullet is faster than systems that store files on disk as block sized fragments it does not handle very large files. Nor does it support files which are frequently changed, like log files. Amoeba has a separate, block-based, server to meet these requirements. Because files are immutable the disk becomes fragmented with time. This reduces the amount of data that it can store and introduces the need for compaction from time to time.

4.3 Clouds

The Clouds distributed operating system[60][61][24] was designed and implemented at the Georgia Institute of Technology in Atlanta, USA. Clouds is an object oriented distributed system with support for persistent objects.

Goals

The main goals of Clouds' developers were to develop a system that provided:

- an integrated, location transparent, distributed system
- support for a variety of atomic transaction mechanisms and data consistency mechanisms
- fault tolerance
- portability
- extensibility
- an efficient implementation

Design

At the core of the Clouds design is a minimal kernel, *Ra*[24], that supports persistent objects stored in a global address space, as shown in figure 4.3. Lightweight processes (threads) execute in this address space.

Clouds does not have a file system in the normal sense. Instead objects can exist in a permanent, shared memory space. If necessary it is possible to model files using persistent

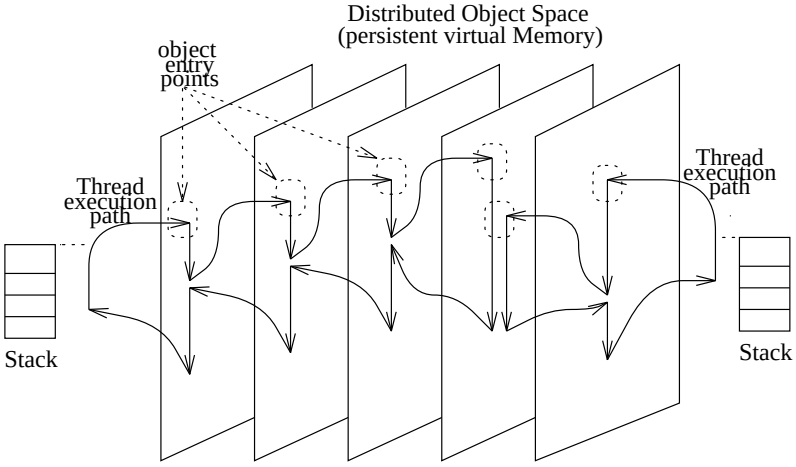


Figure 4.3: Clouds
(adapted from [60])

objects. File data and read, write and file management operations can be encapsulated within an object.

Caching and consistency

Because the memory space is shared, a coherence protocol is used to provide strict coherence. Each node in a Clouds distributed system has a distributed shared memory controller that owns the memory that is created by threads executing on that node. When a remote system requests a page, a locking mechanism is used to ensure that the cached copies of the page remain coherent where this is necessary. Weaker, call back based, consistency is also provided for those applications that do not need strict coherence.

There are four levels of locking:

- **Read/write**. This is an exclusive lock. While it is held no other systems have coherent access to the page.
- **Read only**. This is a shared lock. Other systems may have read only locks simultaneously.
- **Weak read**. This 'lock' can be concurrent with any other type. The process with a weak read lock is not guaranteed coherent but updates are provided via update messages.
- **None**. No protection or updates are provided.

Distribution Interface

The Clouds distribution interface is designed to support distributed shared memory. Consequently it contains operations to read and write pages and to associate pages with virtual memory. Before listing the operations supported by the distribution interface this section introduces the Clouds distributed shared memory system.

There are four abstractions in the Clouds distributed shared memory model. These are:

- **Segment**: A segment is a byte sequence that is stored in the system. Segments exist on a single computer in the system. Associated with segments are storage attributes including persistence or volatility, and coherence level (read-only, read-write etc., as shown above).

- **Page**: A page is a fixed length component of a segment.

- **Physical Frame**:

  On any particular machine the segments it manages must be mapped into the local memory management system for storage. A local storage component is known as a Physical Frame.

- **Virtual Space**: A virtual space is a (possibly distributed) virtual address space.

  There are three types of virtual space:

  - **O** The memory used to store an object.
  - **P** Memory used by a thread, including the stack and parameters.
  - **K** Kernel memory.

install	Creates a new virtual space of type O or P.
uninstall	Removes a virtual space
Create	Create a segment, giving it a name and associating it with a storage location.
Destroy	Remove a segment
Activate	Create the system data structures that are necessary for a segment to be mapped to a virtual space
Deactivate	Remove the system data structures that are necessary for a segment to be mapped to a virtual space
AddMapping	Introduces a mapping between a part of a virtual space and a segment.
DeleteMapping	Remove a mapping between a part of a virtual space and a segment.
WhoMaps	Gives information about the virtual space and segment associated with a virtual address.
Flush	Copies changes in the virtual space to the partitions(s) associated with it
Note: There are also virtual memory calls to manage the association between segments and physical frames. These are not include here because they relate only to virtual memory, not the persistent storage mechanism.	

Table 4.3: Clouds Distribution Interface
(extracted from [5])

Clouds has operations to map segments, or parts of segments, into virtual spaces. This makes an association between the virtual address space and segments stored on particular machines. The Clouds distribution interface contains the operations shown in table 4.3.

Experience

A performance study of Clouds shows that the two most significant sources of overhead are processing the coherence protocol and network latency[5]. Clouds has these limits in common with systems that directly implement distributed file systems but the use of a different abstraction (distributed virtual memory) means that more general solutions must be found to improve performance.

Summary

Clouds implements an object based distributed system. Persistent objects can be stored in a shared address space. A traditional file system is not provided (or needed) in the persistent object approach. While this is an advantage in some respects, the change in

paradigm limits Clouds immediate application base and requires applications to conform to the object oriented approach.

4.4 Ficus

The Ficus distributed file system[87][162][184] was developed at the University of California, Los Angeles in the early 1990s. Gerald Popek, one of the primary researchers in the Locus project[174], is also a contributor to the Ficus project and some of the techniques in Ficus had their origin in Locus.

Goals

The main goal of Ficus' developers is:

- **Very large scale**: Ficus should be able to scale to millions of sites.

Guy[87] discusses a number of problems that are likely to be met in any system of sufficiently large scale. These are:

- No single file can identify all the components of the system.
- Partial operation, in which some components of the system are unavailable, will be the normal mode of operation.
- Administration is made more difficult because of added complexities in resource management, protection and legal issues.
- It is difficult to avoid the need for users to interact with the system in a way that differs from interactions with smaller systems.

Other goals include simplicity, ease of use, file system boundary transparency, syntax transparency with respect to existing file system types, location transparency, name transparency, local autonomy for name management and compatibility with embedded names in existing software. Because of the need to support partial operation Ficus is also suited to disconnected operation[94].

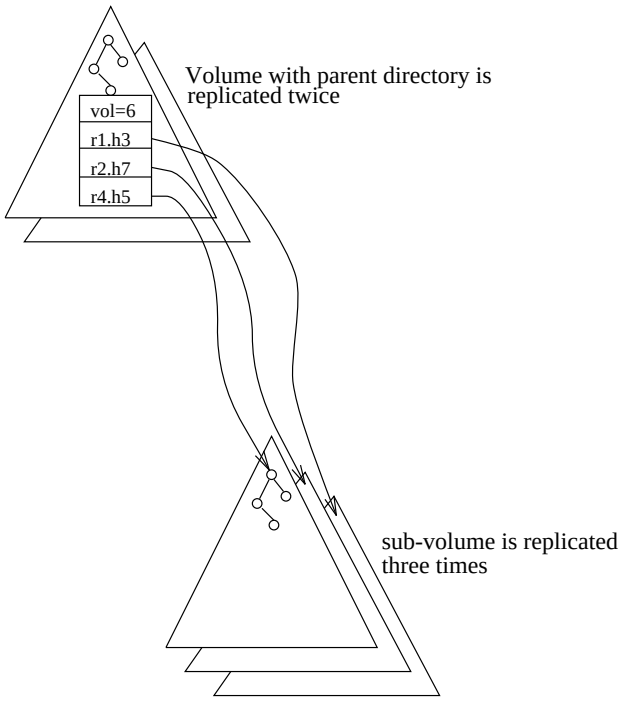


Figure 4.4: Ficus Replication
From [87]

Design

Replication and optimistic concurrency control are the two main components of the Ficus design. Ficus is built using stackable layers, that is layers that have the same interface but add additional facilities, such as replication[95].

Ficus replication is based on subtrees within the file system known as volumes. Any volume may be replicated. Each volume replica is a container which can contain replicas of the files within the volume. Volumes are implemented as a special kind of sub-directory. When a path name is being translated into a file identifier some components may be graft points, indicating that a different volume is integrated into the file system at this point.

As with most tree structured directory systems a pathname is processed by searching each directory for the next item in the path. Once found that directory is opened and searched for the next component. Because directories are just special purpose files they are replicated and processed in the same way as files. Each time a file (or directory) is opened Ficus searches the volume replicas for a copy of the file. If the volume has a replica on the computer that initiated the search this volume will be searched first, otherwise a volume is chosen at random. If no copy of the file can be found an error is returned. Figure 4.4 shows a graft point, where one volume is connected to another. In this example the parent volume is replicated twice and the child volume three times.

Caching and consistency

Ficus uses a One Copy Availability (OCA) consistency mechanism. OCA allows an update to be applied to any replica that is accessible. Propagation of the update is performed by a separate propagation service which operates when resources are available. OCA does not enforce coherence at the time of an update. Even if the most recently updated replica is available at the time a file is opened it might not be chosen because of Ficus' random replica selection policy. OCA guarantees that no updates will be lost by using version vectors (see Chapter 3, page 51).

When the propagation service discovers that updates have been applied to different replicas of a file, a reconciliation service is invoked to resolve the inconsistent updates. In some cases, such as directory update, the semantics of updates are sufficiently well known to allow automatic resolution of update conflicts[116]. Other conflicts must be referred to a human for resolution.

Distribution Interface

Ficus is layered on top of the SunOS version of Unix. The original implementation of Ficus used NFS to transfer files from machine to machine. This was not entirely successful because NFS contains internal optimisations that conflicted with the needs of Ficus. Newer versions of Ficus use a vnode distribution interface. A vnode is an abstraction of a file. It is implemented as an encapsulated object that supports a set of operations on the vnode. A vnode on one system may refer to one on another. To implement an operation the local vnode transmits operations on it to the remote vnode.

The Ficus distribution interface has 40 operations including operations to create files and replicas, open an available replica, manage directories and manage graft points. Full documentation of the Ficus distribution interface has not been published but the list of operations given in table 4.4 was extracted from source code supplied by one the authors of Ficus[93].

Experience

Optimistic operation allows good performance and availability in large systems. The use of version vectors and a reconciliation service ensures that updates are either processed or brought to the attention of a human user. Guy[87] suggests that manual reconciliation is only rarely required in a University environment.

addentry	clearupdater
cntl	commitshadow
control	debug
dropvolcache	ffshlook
ffshremove	getfattrs
getupdater	graft
graft	makeunnamedentry
newcreate	newmkdir
newpropdirentry	newremdirentry
newremove	newrename
openqueue	openshadow
openspecial	orphan
propdirentry	propfile
proprename	propsymlink
remdirentry	removeshadow
replica_lookup	replidalloc
selectvop_getfattr	setfattrs
subgetfattrs	switch_graft
ungraft	update_link
upddirentry	viewop_update_volumeinfo

Table 4.4: Ficus Distribution Interface

Summary

Although the basic service should scale well it is not clear that the reconciliation service will scale to very large distributed systems. Ficus does not meet the needs of applications that process large files with random access and frequent updates. It only allows replication at a coarse level.

4.5 NFS

NFS[141][194] is the best known and most widely used distributed file system. It is frequently used as the benchmark against which other systems are judged. Although NFS is often considered to be a distributed file system it is probably better thought of as a network file system (as its name suggests) because it provides only simple facilities for sharing. In particular it does not provide transparency to the system manager and does not provide migration or replication.

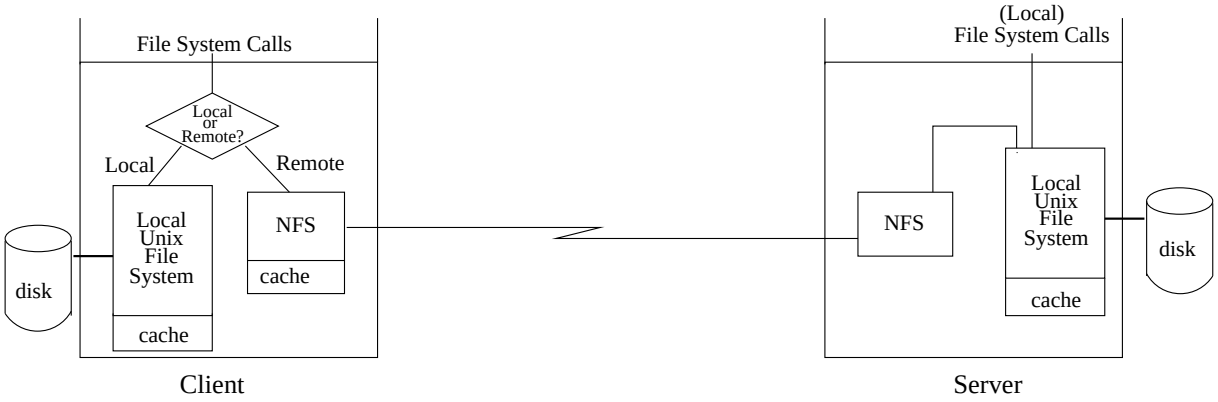


Figure 4.5: NFS

Goals

The main goals of NFS's developers were:

- **Machine and operating system independence**: Support for hardware and operating systems from different vendors.
- **Crash recovery**: Statelessness and a high degree of idempotence lead to a system that can recover from most faults without incorrect operation.
- **Transparent access**: Applications programs require no modifications to access remote files.
- **Reasonable Performance**: The extensive use of caching and allowing a degree of inconsistency leads to good performance.

NFS was not designed to support strict coherence, to scale well or to support automatic migration or replication.

Design

NFS provides a mechanism to allow file system-calls to be satisfied either locally or remotely. When a remote file is opened, the file name is passed to the remote system which returns a file handle for the file. This identifier is stored by the client and used in later operations concerning that file. When a read or write system-call is made on a file descriptor that is for a remote file, the call is translated to an NFS read or write call and passed to the remote system via the distribution interface. In the case of a read, the data is returned in the reply and then passed back to the user program that made the system-call. If the request exceeds 8kb it is divided into 8kb blocks for transfer.

Other operations such as deleting, renaming and creating links to files each have their own operations in the distribution interface.

A key feature of the NFS design is its avoidance of inter-system state information. The server does not hold any state information about the clients that are using it and the state information held by the client is limited to the file handle. The distribution interface is idempotent under most circumstances.

Caching and consistency

Both the client and the server maintain caches. The server cache has the same function as the standard Unix file system cache; that is to reduce the number of disk accesses. At the client a cache is maintained to reduce the amount of network traffic required and to reduce the workload on the server. Because several different clients can cache the same file a consistency mechanism is required to manage the client caches.

NFS does not provide strict coherence but it limits the time during which inconsistent data is visible. Write operations are written to the cache and also to the server as they occur. This means that any later read operation from the server will see the new data. However if more than one client holds a copy of a particular block a write by one of the clients will leave the other client cache(s) with stale data. The time for which files can remain inconsistent is limited using time stamps. Whenever data is fetched from a file the time the file was last modified is also fetched and saved by the client. The time is also fetched when a file is first opened and when a read operation is performed if it has not been fetched recently.[13] Whenever the time stamp changes all blocks cached for that file are discarded.

NFS caching allows NFS to perform almost as well as a local file system for read and write operations. However need for frequent cache validation messages causes high network loads and limits the scalability of the system. One study reports more than 50% of the NFS network load comes from consistency operations[145].

Distribution Interface

The distribution interface for NFS has the 18 operations listed in table 4.5. Supporting these operations is a remote procedure call mechanism (RPC[132]) and a data presentation standard (XDR[131]).

[13] Often a window of 3s is used for ordinary files and 30s for directories.

	Name	Description
1	NULL	Do nothing. Used to test server response and timing.
2	GETATTR	Get file attributes including last modified time (see discussion of consistency).
3	SETATTR	Set file attributes.
4	ROOT	Obsolete.
5	LOOKUP	Translate a name to a file handle.
6	READLINK	Get the pathname contained in a symbolic link Note: Because NFS parses pathnames on the client, the pathname in a symbolic link may mean something different (or be meaningless) on a different client or on the server if a different pathname syntax is used.
7	READ	Read data from a file.
8	WRITECACHE	Reserved.
9	WRITE	Write data to a file.
10	CREATE	Create a new file and return its file handle and attributes.
11	REMOVE	Remove an existing file. Note: This might not be idempotent.
12	RENAME	Change a file's name. Note: This might not be idempotent.
13	LINK	Create a (hard) link. Note: This might not be idempotent.
14	SYMLINK	Create a symbolic link.
15	MKDIR	Create a new directory Note: This might not be idempotent.
16	RMDIR	Delete an empty directory Note: This might not be idempotent.
17	READDIR	Read some or all the entries in a directory.
18	STATFS	Give information about a file system including the size, free space and block size.

Table 4.5: NFS Distribution Interface

Experience

Recent versions of NFS have not changed the core design but have improved security and performance and have addressed some management issues such as the automatic mounting of remote file systems when they are used.

The 4.4BSD implementation of NFS (called NQNFS, Not Quite NFS)[133] includes an alternative distribution interface that supports strict coherence. It is based on the concepts of Spritely-NFS[208] but uses leases[85] instead of locks so that crash recovery is made simpler.

Summary

NFS is a simple system that works well in many environments. It does not support migration or replication and, in most versions, provides only weak consistency and security mechanisms. Placement and management of files is done at a very course grained level. Despite the limitations, NFS has proved, through its wide and extensive use, that distributed file systems are a valuable resource in many organisations.

4.6 Sprite

The Sprite network file system[159][244][152] was developed at the University of California, Berkeley from the late 1980s until around 1992. A second project, Spritely NFS, applies the principles of Sprite to NFS. Ousterhout, the principle author of "A Trace-Driven Analysis of the UNIX 4.2 BSD File System"[161], was also a principle researcher in the Sprite project and a member of the Sprite LFS[190] (a log structured file system) and Zebra[92] (a striped file system described below) research teams.

Goals

The main goal of Sprite's developers was:

- **Transparency**: They wanted to provide services equivalent to those found in a centralised environment in a distributed system. In particular to provide true one copy semantics with good performance in a distributed file system.

Design

Sprite provides the Unix file system interface but is a re-implementation of it in a distributed environment. Unlike other distributed Unix file systems Sprite strictly implements the Unix semantics, not just an approximation to them. To improve performance, both the client and the server have caches. A delayed write policy is implemented so write operations are written into the client cache but are not immediately written to the server. Every 30s any dirty blocks that have not been modified in the last 30s are written back to the server.

To correctly implement the Unix file system interface requires servers to keep information about files that clients have open. This state information is needed to implement the Unix delete open file semantics (see page 11).

Caching and Consistency

Servers maintain a list of what files clients have open and whether the files are open for reading or reading and writing. No inconsistencies can occur unless more than one client has a file open and at least one client has the file open for writing.

When inconsistency is possible the server informs all clients that this is the case. The clients are required to bypass their cache and read and write data directly to the server. When this happens only the server cache is used and all read and write operations are applied to that cache. Consequently one copy semantics are provided.

Delayed write widens the window when a client failure will cause loss of data.

Distribution Interface

The distribution interface contains RPC calls to open, process and close files along with calls to maintain the file system.

Because of the nature of Sprite many of these calls parallel file system-calls. However there is not always a 1:1 relationship between calls to the file system and calls across the distribution interface. Two reasons for differences are the presence of the client cache and the need to maintain file system state between the client and the server.

	Name	Description
1	ECHO_1	Null operation used for performance and maintenance. ECHO_1 is implemented by the servers interrupt handler.
2	ECHO_2	Null operation. ECHO_2 is implemented by the servers RPC process.
3	TEST_SEND	Test data transfer to server.
4	TEST_RECEIVE	Test data transfer to client.
5	GETTIME	Find the current time, using a broadcast.
6	FS_PREFIX	Find the server that maps names for a part of the tree.
7	FS_OPEN	Open a file.
8	FS_READ	Read data from a file.
9	FS_WRITE	Write data to a file.
10	FS_CLOSE	Close a file.
11	UNLINK	Remove a name for a file and delete the file contents if there are no more names and no current users of the file.
12	RENAME	Change a file's name.
13	MKDIR	Create a directory.
14	RMDIR	Remove a directory.
15	MKDEV	Make a special file.
16	LINK	Make another name for an existing file.
17	SYM_LINK	Make a symbolic link to a file.

continued on next page

	Name	Description
18	GET_ATTR	Get attributes of an open file via its handle.
19	SET_ATTR	Set attributes of an open file via its handle.
20	GET_ATTR_PATH	Get attributes of a file via its pathname.
21	SET_ATTR_PATH	Set attributes of a file via its pathname.
22	GET_IO_ATTR	Get attributes as stored at IO server rather than the naming server.
23	SET_IO_ATTR	Set attributes stored at IO server.
24	DEV_OPEN	Open an IO device.
25	SELECT	Test an IO device for data.
26	IO_CONTROL	Configure an IO device.
27	CONSIST	Issued by a server to inform a client of the current consistency mechanism required on a file.
28	CONSIST_REPLY	Sent by the client when it has responded to a FS_CONSIST call.
29	COPY_BLOCK	Copy part of one file to another on the server without passing it through the client. This operation improves the performance of the fork() call.
30	MIGRATE	Inform the server that the user of an open file has moved from one computer to another.
31	RELEASE	Tell the original home of a migrated open file that the move is complete and it can release the file handle.
32	REOPEN	Send state information to the server about files the client has open. This is used during the recovery process after a server has crashed.
33	RECOVER	Indicates that the client has completed its REOPEN operations after a server crash.
34	DOMAIN_INFO	Return information about the file system (domain) including the amount of space available.

Table 4.6: Sprite Distribution Interface
(adapted from [244])

There are further RPC calls in the distribution interface for process management and migration and for processing signals.

Experience

Sprite successfully met the goal of providing strict Unix file system semantics including one-copy serialisation. Performance exceeds that of NFS when running the Andrew Benchmark (Sprite takes only about 70% of the time NFS takes[72]). Most of this performance improvement probably arises from Sprite's delayed write policy compared with the NFS write through policy.

Further developments by the Berkeley file system team have concentrated on improving the IO bandwidth between the file system and the disk, in particular using log structured and striped file systems.

Summary

Sprite is a step forward in the direction set by NFS. It maintains the style of NFS's file distribution interface with a loose mapping between file system-calls and the RPCs in the distribution interface. It addresses the semantic problems of NFS and improves its performance and supports migration of active processes.

4.7 Zebra

The Zebra file system[92][168][158] followed the Sprite log structured file system[190] and combines the concepts of a log structured file systems and RAID[168].

Goals

The main goals of Zebra's developers (Hartman and Ousterhout) are:

- **Bandwidth**: To improve the rate at which data can be written to disk. Beating the so called 'IO Bottle Neck'[158]
- **Availability**: To provide high availability.

Design

A Zebra file system is split into three types of component:

- clients
- block servers
- file managers

Zebra is log structured. Log structured file systems are based on the observation that the presence of large caches means that there are more physical disk write operations than read operations. Under these conditions it is more efficient to place blocks written to the the disk in time sequence that they are written rather than close to other blocks of the file to which they belong. Other log structured file systems include Sprite LFS[190], Minix LD[64], Highlight[113] and xFS[242].

Zebra adds striping to the log structure. The stream of blocks from a client is spread amongst a group of servers. Because each server only writes part of the workload, any write bandwidth that is desired can be implemented using Zebra, given sufficient resources.

Fault tolerance is provided in Zebra by grouping servers together and having one server maintain parity information for the data stored on the others. If any one server fails, the data it held can be deduced from the content of the other servers and the parity server.

Caching and consistency

Zebra is block-based in that individual blocks of the file are placed on different servers. However file meta-data is stored and maintained at a single site known as a file manager. Storing the file meta-data at a single site allows that site to enforce serialisation of updates to the meta-data.

Zebra uses the same file data coherence mechanisms as Sprite.

Details of the Zebra distribution interface have not been published.

Experience

Hartman and Ousterhout report that Zebra outperforms NFS and Sprite by $5 - 8$ times for large files[89]. Smaller files showed only a small increase (around $10 - 20\%$ over Sprite). The authors believe that this is primarily because Zebra does not cache directories.

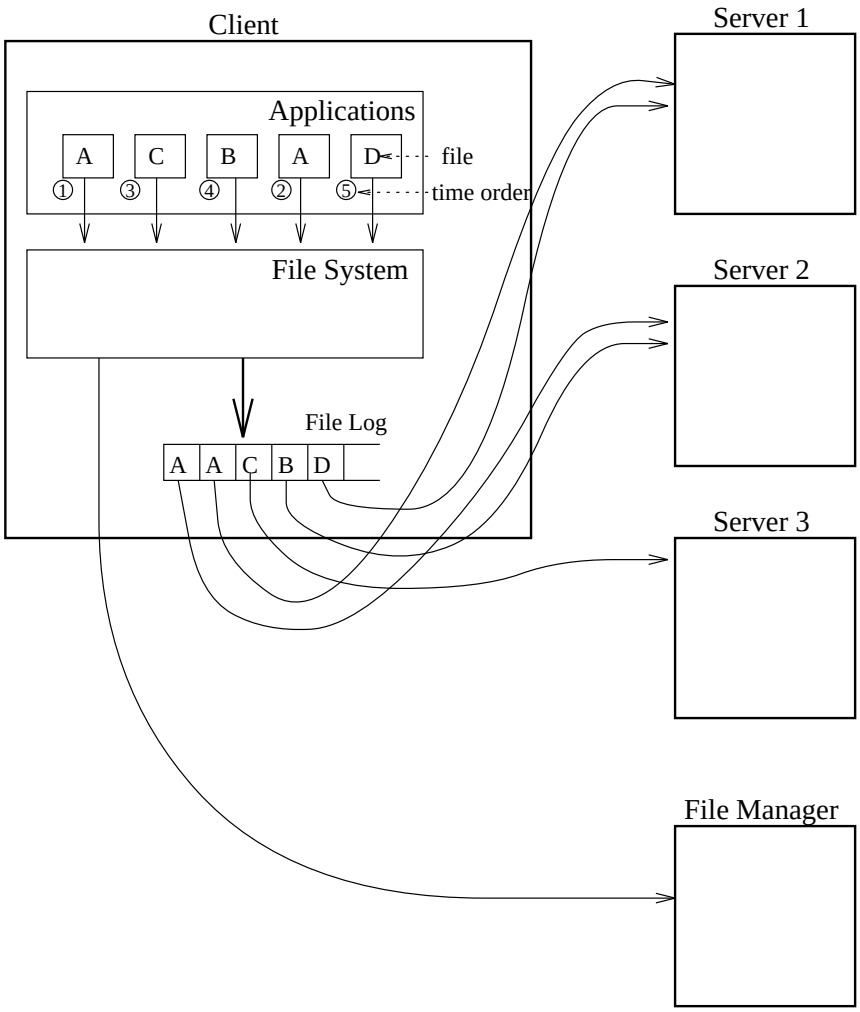


Figure 4.6: Zebra

Zebra Summary

When processing large files Zebra performs significantly better than its predecessor, Sprite, without sacrificing strict coherence. Although data is separated into individual blocks for writing to servers Zebra is still primarily file based, in that file meta-data is maintained at a single site.

4.8 Summary

The distributed file system area has a long history and there are many distributed file system designs. The distributed file systems in this chapter illustrate the major types of system. Early systems focused on achieving remote access to data, the interface presented to users, replication and performance. In the mid to late 1980s there was less work addressing traditional file system interfaces but many object-based systems were designed.

Two main themes are apparent in recent work. The first is a return to addressing performance issues. Log-based storage and striping are two techniques that are being used to improve file system performance. Zebra typifies such systems, being both log based and striped. Striping is performed because the server is often the component that limits the performance of a distributed file system.

The second theme is support for systems that are not permanently connected to the network. Coda and Ficus are examples of systems that support disconnected operation. Typically support is provided for disconnected operation through long-term optimistic operation.

The block-based approach can contribute positively to performance by making the server simpler and consequently able to support a higher workload. As discussed in the conclusion to this thesis (Chapter 8 page 130) it may be possible to implement the server in hardware. The block-based approach also reduces the complexity of the communication protocols (improving their performance) and allows the advantages of striping and log-based operation to be applied to file meta-data as well as file data.

An outstanding issue in current distributed file systems is support for fine grained operation. Chapter 2 contains a detailed discussion of the fine grained nature of the block-based approach (see page 19) and also of other advantages of the approach, including openness and integration with other system components, flexibility and the server and protocol issues mentioned above.

Chapter 5

Architecture

The previous chapters have discussed the context and motivation for this investigation into block-based distributed file systems. This chapter presents an architecture for a block-based distributed file system. The architecture provides a framework which addresses the distributed file system issues presented in Chapter 1.

In Chapter 1 distributed file system issues are presented from a system perspective. In this chapter the same issues are treated from the perspective of the architecture. Some components of the architecture address more than one of the issues discussed in Chapter 1 while others only provide part of the solution to an issue. This chapter ends with a summary of the issues presented in Chapter 1 and the way they are addressed by the architecture present here.

The architecture is a broad overview of a block-based distributed file system that addresses all the major issues of a distributed file system and shows the relationships between them. It is not constrained by what has been implemented to date. The next chapter, Chapter 6, describes the implementation of a file system (BB-NFS) that demonstrates that the key concepts of the architecture can be implemented in a practical system. BB-NFS implements the distribution interface, file system-call interface, caching and cache consistency but does not demonstrate the security, replication, migration or naming parts of this architecture.

Component	Block level	File Level
distribution interface	√	
file system-call interface		√
caching	√	
cache consistency	√	√
security		√
naming	√	
replication	√	
migration	√	

Table 5.1: Components of the Architecture

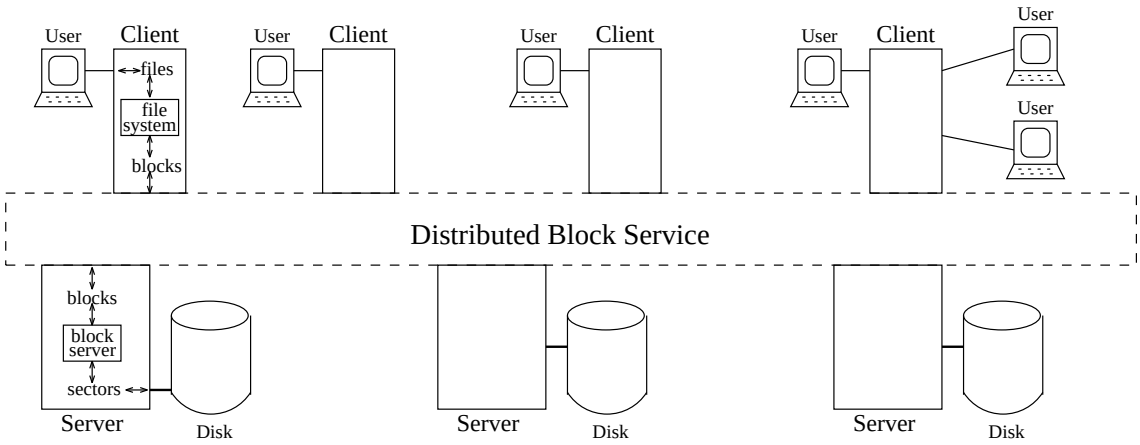


Figure 5.1: Block-Based Distribution Interface

The architecture can be considered in two parts (see figure 5.1):

- the architecture of the distributed block service
- the architecture of a file system that uses the distributed block service

Table 5.1 shows the main components of the architecture and which part, or parts, they belong to. The sections that follow discuss these components.

5.1 Distribution Interface

The concept of a block level distributed file system was introduced in Chapter 1 under the heading 'Level of Distribution' (page 17).

The file system is built on a distributed block service. This service offers its users access to fixed sized blocks of data. Each block has a unique identifier. Clients make requests at the distribution interface to read and write blocks and also to lock and unlock blocks.

Operations

There are six types of operation that can be performed at the distribution interface. They are:

- `read` read a block
- `write` write a block
- `locked read` read and lock a block
- `unlocking write` write and unlock a block
- `revoke` revoke a lease

`Read` and `write` perform unlocked remote block operations. The provision of remote read and write operations is sufficient to implement a file system that does not support sharing. `Locked read`, `unlocking write` and `revoke` implement user coherence which is discussed later in this chapter.

Communication

A major distinction between a block interface and other interfaces is avoidance of any notion of a file, or other high level structure, in the interface. The servers in the distributed system offer the distributed block service. Clients make use of the block service to provide a file system service to their users. The division of functions into client and server functions is a logical one. It does not imply that a single machine cannot perform both roles.

5.2 File System Interface

The servers enhance the block interface to provide a file abstraction to their users. Many file abstractions are possible. It was noted in Chapter 1 that the use of an existing interface allows a wide range of existing applications to be supported (see 'Portability' on page 10). It also ensures that no essential aspects of a file system interface are omitted. The file system provides the Unix file system interface[187].

5.3 Caching

Caching is necessary to provide satisfactory performance (see page 11 of Chapter 1) and scalability (page 7). The architecture includes both client and server caching. In Chapter 2 it was noted that most file systems are unable to cache file system meta-data such as directories and file tables (see page 21). Because all data access, including directory and other file system meta-data, is block-based a block cache will improve the performance of all block accesses not just those for application data blocks.

Cache management, including pre-fetch and block eviction policies, are the same as those for other file systems.

5.4 Cache Consistency

Cache consistency is treated in two parts, file data consistency and file system meta-data consistency.

A wide range of consistency needs and policies for file data were discussed in Chapter 2. The particular consistency required is application specific. The architecture leaves the choice of file data consistency open, allowing the implementor of a particular file system to choose an appropriate consistency mechanism or perhaps to provide a selection of them.

File system meta-data, however, requires protection against concurrent update. Application supported user coherence is provided for this need. As discussed in Chapter 3 (page 51) application support allows the coherence mechanism to operate on only the data which needs to be kept coherent.

In Chapter 3 it was also noted that application supported coherence requires all communication between users to use the consistency mechanism. In this case the file systems of the client machines are the 'users' of the coherence service. The file system implementor can ensure that all communication is through the coherence mechanism.

User coherence is provided through a coherent block locking mechanism. Using the `locked read` and `unlocking write` operations individual blocks may be locked and, when a lock is granted, the client cache is guaranteed to be coherent. Locking is a simple consistency mechanism that is sufficient for the needs of the file system. Coherence is ensured when a lock is granted by returning the data of the locked block with the reply that grants the lock.

Four concerns must be addressed by the implementation. These are:

1. **Identification of coherency needs**: Application supported coherency relies on the application requesting coherence when it is needed. To implement this correctly requires a careful analysis of the file system to identify those parts of the code where concurrent update of a file system data structure can cause inconsistencies in the file system meta-data.

2. **Correct failure operation**: Locks must not be held indefinitely in the event of a client failure and locks must not be granted in an inconsistent manner in the case of a server failure. These requirements are met by issuing locks with a limited lifetime. This type of lock, which is known as a lease, is described in Chapter 3 (page 44).

3. **Partial results hidden**: The file system must not make partial results visible until the file system meta-data is in a consistent state. This is achieved using a two phase locking approach[25]. In the first phase, locks are acquired but no locks are freed. In the second phase, locks are freed but no new locks are acquired.

4. **Deadlock recovery**: The order in which locks are requested is determined in part by the ordering of application requests to the file system. This order is outside the control of the file system. Because locks are not taken in a known order it is possible that two or more clients might enter deadlock. (See 'Deadlock' on page 6 of Chapter 1).

   Deadlock is detected by the block server when a lock request is generated that causes the system to enter deadlock. The request that caused the deadlock is refused and all locks held by that client are required to be returned, so that other clients can proceed. To support this, all operations performed by the client that require coherent data must be able to be aborted and restarted.

5.5 Security

Because the block servers may reside at arbitrary physical locations and are vulnerable to attack directly through the hardware, it is not possible to protect data stored on them from unauthorised disclosure or modification.

Security is provided by the file system using encryption and digests. Users authenticate themselves with a password, which is used to decrypt the content of an access directory. The access directory contains the decryption keys for files that the user has permission

to access. It may also contain the keys to other access directories that contain the keys for files that belong to groups that the user is a member of.

Encryption based security is discussed on page 10 of Chapter 1. A limitation mentioned there is that any user who is able to decrypt the file can both read and write the file. The block-based distributed file system architecture addresses this by associating a message digest function with each file. The digest function takes the content of the file and produces an output which is difficult to reproduce without knowing the digest being used. The output is stored with the file. However, the digest function itself is only stored in the access directories of users who have write access to the file. The combination of encryption and digest allows separate read and write access to be granted to users or groups.

Although the architecture provides the Unix file system interface, there are two aspects of Unix security that cannot be implemented if the hardware of the servers and clients are accessible to an attacker. The first is the provision of execute (but not read) access. Execute only access cannot be provided because once the program is loaded into the memory of a client workstation an attacker can read the content of the memory. This weakness also exists in many centralised systems that implement Unix's execute protection mechanism.

The second security risk that cannot be addressed in this environment is unauthorised destruction of file data. An attacker who has access to the server hardware can write a program that accesses the disk directly or may even remove the disk drive from the server. Deliberate removal of data is similar in many respects to accidental loss of data through failure of a disk drive. Most sites have integrity mechanisms to protect data from loss in the event of hardware failure. These mechanisms also provide a way to recover from deliberate attack and may be all that is required, particularly if the environment is predominantly cooperative.

5.6 Replication

Replication was introduced in Chapter 1 under the heading 'Availability' (on page 4). Replication can be included in a distributed file system for three reasons:

- To improve the performance of read operations at the expense of write operations.
- To improve the performance of write operations at the expense of read operations.
- To improve the availability of data at the expense of performance.

While replication may be important to meet the needs of a particular application, the cost of replication will be undesirable to others. Further the appropriateness of a replication mechanism depends on the need it is intended to address. It is not possible to implement a single replication mechanism to meet the needs of all applications. Because it is not possible to pick an appropriate replication mechanism without knowledge of the applications concerned, the block-based distributed file system architecture leaves replication as an open issue. Replication can be implemented as required to meet the needs of a particular system.

5.7 Migration and Naming

If a block is mostly used at one location it should, where possible, be stored at that location. The architecture supports migration by using location-independent block names and by allowing servers to exchange blocks amongst themselves.

This area of the architecture is not fully developed because more work, based on measurements taken from the test implementation, is required to make decisions on the most appropriate techniques. The discussion included here focuses on general principles and gives some suggestions for further study.

Block Names

To allow migration between servers, block names are independent of the location of the block. They do, however, contain a hint for use in locating the block.

When a block is required the name translation mechanism maps a block name into a block address which contains two parts, a server identifier and a block number at that server. Name translation occurs, in the first instance, at the client but may also involve one or more servers.

The client begins the name translation procedure by making the most accurate translation possible, based on the block name, the hint, past translations and any other information available to it. It then sends the name and the address to the server specified by the address. If the translation is correct the server acknowledges the correct address, otherwise it replies with a new block address identifying a server that might store the block. In the latter case the client retries the translation based on this new address. When a server replies that a correct translation has been made, the client uses the block address to

complete the operation (e.g. read or write) on the block.

Although the block operation follows name translation, the architecture does not imply that separate messages must be used for the two operations. It is permissible to send both the attempted name translation and the data needed to perform the block operation to the server in the same message. If the name translation is successful the operation will also be performed by the server.

It is the goal of the implementation of the naming mechanism to minimize the number of incorrect name translations, and thereby, minimize the overhead of the name translation mechanism. A name cache is used in achieving this goal.

Migration

The discussion in Chapter 1 noted that manual load balancing is difficult to do correctly (see 'scalability' on page 7). The block-based distributed file system architecture places the responsibility for making the decision to move a block from one server to another with the distributed block service, not with a system manager. This allows automatic load balancing in the file system.

Further information needs to be gathered before a particular block migration policy is chosen. However, there is some evidence, from distributed virtual memory research, to suggest that simple migration schemes perform almost as well as more complex ones. A possible migration scheme, based on the work of Marchetti *et al.*[137], might be to move the block to the site where it is first referenced and not to move it again until it becomes idle.

5.8 Summary

The architecture described here provides a framework for the development of a block-based distributed file system. The main features of the architecture are:

- Block level distribution interface.
- Unix file system interface to applications.
- Client and server caching. Block locking with coherence for updating file system meta-data.
- No security at the block level; encryption and message digests used by the file system layer.

Need	Meet by
transparency (§1.1)	Low level communication used to offer traditional file service (§5.2).
availability (§1.2)	Replication if chosen (§5.6).
integrity (§1.3)	Application supported user coherence, leases, two phase update (§5.4) plus careful implementation, e.g. Unstable Atomic Update (§6.9).
scalability (§1.4)	Block nature (§5.1) allows many servers (§2.2). Low overheads at server (§2.2).
openness (§1.5)	Different file systems can share data through the block service. Other openness can be added at the file system level if desired. (§2.2)
heterogeneity (§1.6)	No explicit support for data format conversion. Simple block service (§5.1) allows different architectures to access the data.
security (§1.7)	Encryption and digests (§5.5).
portability (§1.8)	Unix file system interface (§5.2).
performance (§1.9)	Fine-grained load spreading (§2.2). Low CPU overhead at server (§2.2). Light weight network protocol (§5.1).

Table 5.2: Distributed File System Requirements met by the Architecture

- File data coherence and replication policies left open to allow the architecture to be tailored to meet a variety of needs.
- Location independent block names.
- Naming and migration require further investigation.

Chapter 1 contains a discussion of the needs of a distributed file system. Table 5.2 summarises how those needs are met by this architecture.

Chapter 6

Implementation

The previous chapter presented an architecture for a block-based distributed file system. A number of issues are left unresolved in the architecture. The most important of these are:

1. Can a block-based distributed file system be implemented with a reasonable amount of effort? This question arises because concurrent programming is difficult, especially within the kernel. Some distributed file system implementors have deliberately avoided concurrency in their designs.

2. How much extra work will be caused by the need to lock and unlock blocks?

3. Does a block-based distributed file system exhibited behaviour that the design can take advantage of to improve performance? As noted in Chapter 5 this includes the migration policy (see 'Migration' on page 86).

4. Can a naming mechanism be implemented that has satisfactory performance?

This chapter describes an implementation of a file system built to answer these questions and to provide the basis for further development of the block-based distributed file system concept. The file system described here is not a complete implementation of the architecture of Chapter 5. It is intended to test the key concepts.

The following chapter, Chapter 7, discusses measurements taken from the file system. The results of this study are discussed in the final chapter, Chapter 8.

6.1 Service

Although the file system is not a complete implementation of the architecture of Chapter 5 it is a compete network file system. The file system provides a networked Unix file system much like Sun's NFS, consequently it has been named BB-NFS.

Using BB-NFS, file systems that are physically located on remote machines can be mounted on the local machine. Once mounted file system-calls can be carried out in the same way as with a local file system. No modification or recompilation of application programs is required.

BB-NFS includes the following features of the architecture:

- block-based distribution interface
- light weight communications protocol
- Unix file system-call interface
- caching
- lease based locking
- deadlock detection
- application supported user coherence

It also includes a number of implementation features that are not part of the architecture, or are a specific implementation of part of it. These include:

- Transaction based file system implementation, allowing a file system-call to be aborted and executed as often as needed. This is useful in deadlock recovery.
- Unstable Atomic Update, which ensures that updates to file data have all-or-nothing semantics, even in the presence of network or client failures.
- A lease cache to improve the performance of repeated lease requests on the same lease.
- Piggy-back lease returns for better performance when leases for unmodified blocks are returned to the server.
- Divergent partitions for file-systems[14] that only vary slightly from client to client.
- Instrumentation and logging.

[14]The term file-system is used in its Unix context here, that is a part of the tree structured directory system that resides on a single disk partition.

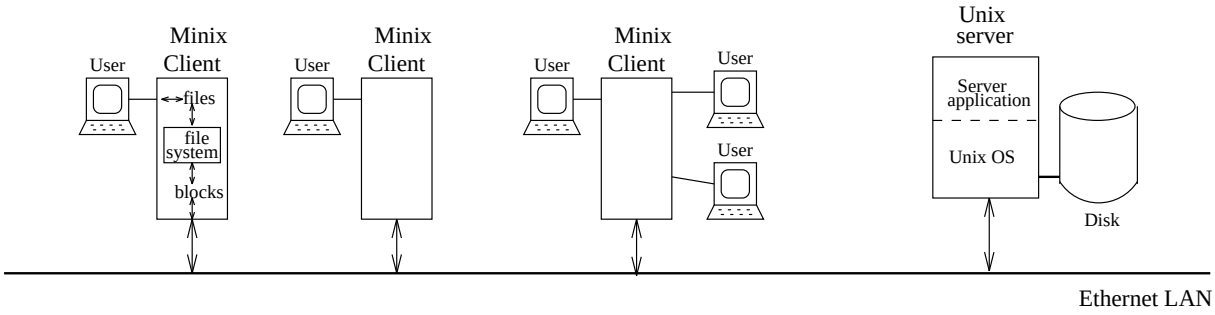


Figure 6.1: Implementation Schematic

Like Minix, BB-NFS supports all the Unix version 7 file system-calls (see appendix B for a list of the file system-calls). It also has two new calls, `instr()`, which causes the operating system to display kernel statistics and `unlkall()` which causes the return of all leases in the lease cache. `Unlkall()` is included to facilitate taking performance measurements.

As explained in Chapter 5 (page 80) the implementation does not include the following aspects of the architecture:

- location independent block names
- replication
- migration
- encryption and digest based security[15]

6.2 Implementation Base

Figure 6.1 gives is a typical example of how BB-NFS is implemented.

The file system is implemented as modifications to the Minix[214] version of Unix. Minix was written by Andrew Tanenbaum as to tool for operating systems teaching and research. It is available free to universities and comes with complete source code for the operating system and all utilities except the C compiler. The kernel (including the the file system and memory manager) of the version of Minix used for this work (version 1.5.10) contains approximately 30,000 lines of code. More information about Minix is contained in appendix B and Tanenbaum's Minix book[137].

Minix runs on the IBM PC architecture. In this study a range of PCs have been used from 1MB '286 machines through to 16Mb Pentium 100's. The clients and server communicate

[15]The standard Unix security mechanisms that assume the hardware and systems software network, client and server are all trusted *are* provided.

via an Ethernet LAN.

Minix was chosen as the implementation base primarily because of the availability of source code. There are now other versions of Unix with readily available source code (including Linux and NET BSD). These did not exist when the project began.

The block server operates as a user program on any Unix system that supports BSD socket operations[55]. The remote file system may be a complete disk partition or a file within a file system mounted on the server. The former will give better performance but the latter is often more convenient. Linux, Ultrix and SunOS have been used to host the server.

The remainder of this chapter discusses the implementation and the way it meets the goals of the Architecture. Further details of the changes made to Minix, at a detailed implementation level, are given in appendix D.

6.3 Distribution Interface

The block-based distribution interface has six operations[16] supported by the distribution interface. They are:

- `read`
- `write`
- `locked read`
- `unlocking write`
- `lease revoke`
- `log`

Read and Write perform unlocked remote block operations. Remote read and write operations can support a network file system that does not allow sharing.

User coherence is achieved by combining locking and coherence, as discussed in the architecture (see page 83). `Locked read` and `unlocking write` implement user coherence. When a lock is granted, coherence is ensured by returning the data of the locked block with the reply that grants the lock.

[16]These 'operations' are transactions at the distribution interface. The term 'transaction' can also be applied to an operation within the file system. Its use in this chapter is limited to that context with the term 'operation' used in the context of the distribution interface. The term 'transaction' might also be applied to the system-call made by an application program when requesting the file system to perform some task. The term 'file system-call' or just 'system-call' is used in this context.

If the data required was not cached at the client, coherence is achieved at very little extra cost because:

- Both the lock granting reply and the block data fit in a single LAN frame. The cost of sending extra data in a frame is small, compared with the cost of sending a request and receiving a reply.
- The file system never locks a block that it does not need. If the data is not already cached at the client it will be fetched from the server. If this was not done in the lock granting message, a subsequent message would be required.

The log operation allows a client to generate log entries that are recorded at the server. This is useful where the relative timing of events that occur on different machines is of interest. For example, if a concurrency error occurs in a particular file system table, a trace of changes made by clients is much more helpful if the interleaving of the client operations can be determined. This is difficult if the trace is recorded separately at each client.

The parameters passed by these operations are shown in table 6.1. The protocol messages and procedures used to implement these operations are explained later in this chapter (starting on page 106).

6.4 Caching

Caching occurs at both the client and the server. At the client the original Minix cache forms the basis for the client cache. As mentioned above the server runs as a user process on a Unix system. The standard Unix cache provides caching to all applications including the server.

The original Minix cache replacement algorithm and most of the management techniques have been retained. The cache has been modified to support the consistency operations required by other parts of the file system. When a request is made for a block an extra parameter indicates whether the block should be locked. If the cache does not hold the block, or it is held but is not locked, the block will be requested from the server via the `locked read` operation. When a block is no longer needed it will be released by the file system code. If a locked block is released the cache code ensures that the lease is returned to the server (the discussion on lease caching on page 97 explains when the lease will be returned).

Operation	Originator	Parameters	(in reply)
Read	Client	Block Number	
		Device Number	
		Piggy Back Lease Returns	
(reply)	server		Result code[1]
			[Block Data][2]
Write	Client	Block Number	
		Device Number	
		Piggy Back Lease Returns	
		Block Data	
(reply)	server		Result code[1]
Locked Read	Client	Block Number	
		Device Number	
		Piggy Back Lease Returns	
(reply)[3,4]	server		Result code[1]
			Short Term Flag[5]
			Deadlock abort Flag[6]
			[Block Data][2]
Unlocking Write	Client	Block Number	
		Device Number	
		Piggy Back Lease Returns	
		Block Data	
(reply)	server		Result code[1]
Lease Revoke	Server	Block Number	
		Device Number	
(no reply)			
Log	Client	Length of message	
		Piggy Back Lease Returns	
		Message	
(reply)	server		(no parameters)

[1] Result is successful or an error code, typically indicating a disk read error

[2] For successful result only

[3] Reply is delayed until lock can be granted

[4] A Lease Revoke to another client may also be generated

[5] See section 6.5 on page 98

[6] See section 6.5 on page 99

Table 6.1: Distribution Interface Operations

The cache has also been changed so that it provides a location for the copy blocks required for the transaction mechanism (discussed later in this chapter, beginning on page 100). The cache and its associated routines are the most heavily modified part of the Minix system. Other changes to the cache, discussed later in this chapter, include eviction of stale blocks, piggy-back and short term leases, support for Unstable Atomic Update and file system measurement.

6.5 Leases

`Locked read` and `unlocking write` are used by the file system to maintain the integrity of its tables and indexes. Keeping locking information means that the client and the server maintain state information about each other. State information can make recovery after the crash of a client or server difficult (see 'Integrity' on page 6). In Chapter 3 (page 44) Gray's leases were discussed as a locking mechanism that made recovery simple.

Leases are locks that are granted for a limited time. The form of leasing used in this implementation is described in the following sections.

Lease Request

When a lock is granted to a client it is given for a maximum time T. Under normal operation the client must return the lock within time T.

The limited-term nature of leases makes them less desirable for a general-purpose application-level locking facility. Applications vary greatly in how long they hold locks for. Because the duration for which an application needs a lock is unknown, selecting an appropriate lease lifetime is difficult. This means that in many cases leases will have to be renewed and this causes extra network traffic.

These problems do not occur in the file system. Because the characteristics of the file system are known, an appropriate lease duration can be chosen. The operations that the file system performs on file meta-data are short. Leases are able to protect against concurrent update of this data without needing to be renewed in normal operation.

When a client requests a lease from the server (for a block that is not currently on lease to another client) the block is returned and a timer set to expire after time T. This is shown in figure 6.2. When the client returns the lease the timer is canceled. If the lease is not returned within time T an error has occurred. The lease is discarded and the error reported to the system manager. This should not happen in normal operation, but can

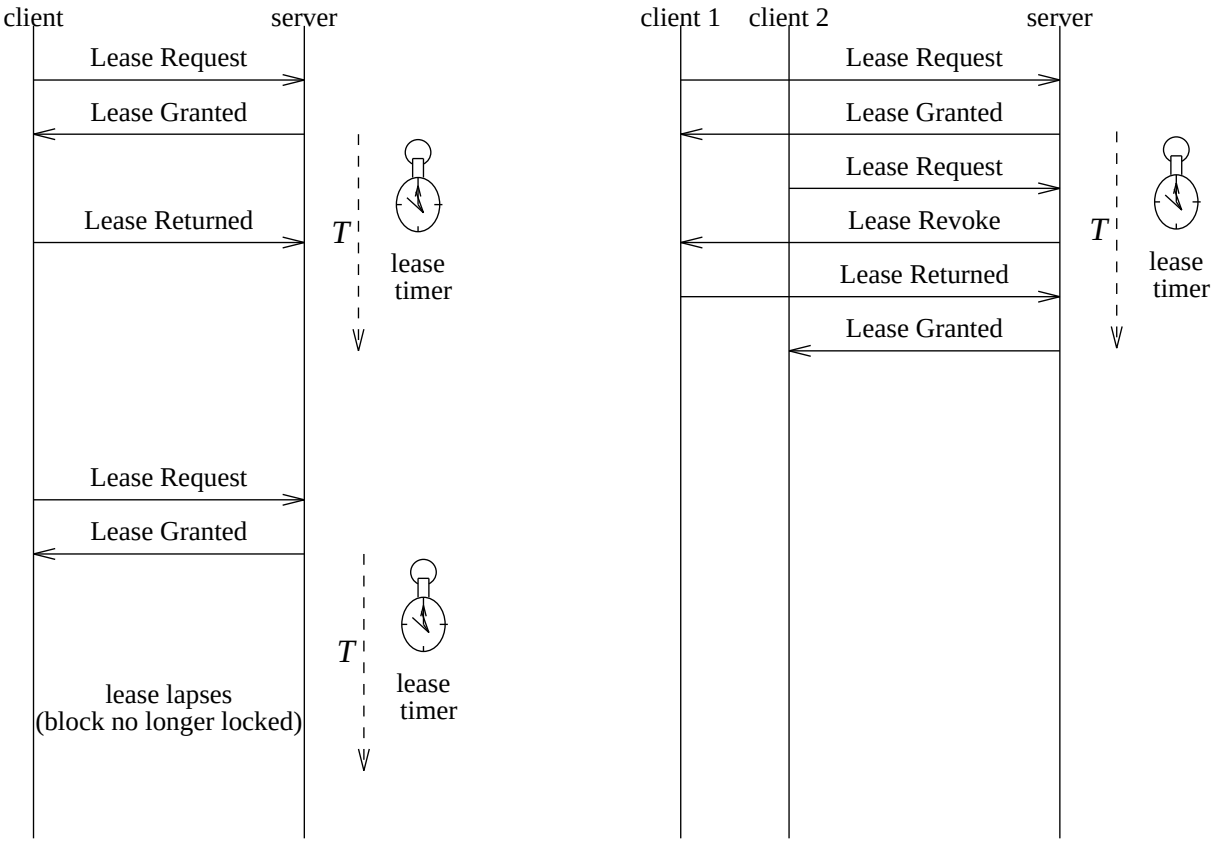


Figure 6.2: Block Leasing

be caused by the failure of the client that held the lease or the network.

Lease Cache

When the file system completes an update of its meta-data, it may return the lease associated with that data immediately. This will allow any later lease requests to be serviced.

Early studies of the behaviour of the file system showed that there were many repeated lease requests and returns for the same lease over short time frames. This behaviour, which is discussed in detail in Chapter 7, suggests that a cache of recently used leases will improve the performance of the file system.

To implement lease caching, the file system normally holds the lease beyond the minimum time required for correct concurrency control in the expectation that it is likely to be needed again. The duration for which the lease is held is limited by the maximum lease lifetime which is discussed under the heading 'recovery' on page 109. Chapter 7 contains a more detailed discussion of the effect lease caching has on file system performance.

Lease Collision

If a lease is requested for a block that is already on lease, no reply is given to the lease request until the lease is returned by the current holder, or it expires. If the lease has some time to run, the performance of the client waiting for the lease will be reduced. Two techniques are used to reduce the delay. The first is lease eviction which is discussed here. The second is short term leases which is discussed in the next section.

When a lease that is in use is requested by another client, the client that holds the lease will be sent a lease eviction. A lease eviction is an advisory message that informs the client that another user is waiting on the lease. If the client is not currently using the lease (i.e. it is a lease that has been cached in the hope that it will be needed again) the client will return the lease immediately. If the block is in use it will be marked so that when it is no longer needed the lease will be returned.[17]

[17] If Unstable Atomic Update (see page 111) is in use, other leases needed to end the current transaction are also returned.

Short Term Leases

Some blocks in a file system are heavily used. In a Unix file system, frequently used blocks include the blocks that store the inode and zone maps and the directories. If leases for heavily used blocks were cached when two or more systems were active they would almost always need to be revoked (i.e. an eviction issued). As a consequence, the latency for these crucial blocks would be increased. The revoke message would also cause an increase in network traffic. To reduce the latency for these blocks the server will issue a short term lease. This instructs the client to return the lease as soon as possible.

The server must decide which blocks should only be granted short term leases. Rather than define a fixed set of blocks on which short term leases apply, the server builds this list from the history of the usage of the block. When a server revokes a lease it marks the block as a short term lease block. The short term lease mark has a limited lifetime (like a lease). If the block is used by more than one client within the lifetime of the short term lease mark, the life of the short term lease is extended. If the block is only accessed by a single client in the short term lease lifetime, the short term lease mark is removed and the block reverts to a normal, long term, lease.

Measurements of a pair of clients running the Andrew benchmark (i.e. the $n = 2$ case shown in figure 7.3 on page 122) shows that about 13% more lease revoke operations would be required if short term leases were not used. Of the lease revoke operations that do occur when short term leases are provided 99% are for leases that are already marked as short term. These are blocks which are currently being used by the client that hold them but will be returned as soon as possible without the revoke mechanism. Under normal conditions the revoke operation is not required for this case. However it is retained to reduce the time required to recover from a lost lease return message.

Piggy-back Lease Returns

Sometimes a lease is obtained for a block but the content of the block does not change while the lease is held. This occurs when the file system needs a coherent copy of the block but does not modify it. For example when data is to be read from a small file, the inode, which contains the block map for small files, is locked. This is done so that the file system can be sure which blocks belong to the file. When the operation is complete and the lease comes to be returned, the inode will not have changed.

To return the lease, the server only needs to be informed that this lease is no longer needed. Instead of generating a special message for this return the file system holds the

lease a little longer and, when a message is next sent to the server the lease return is carried in a field of the message reserved for this purpose. Up to 16 lease returns can be 'piggy-backed' on a single message.

Measurements of a single client running the Andrew benchmark (the benchmark is described in Chapter 4, on page 57) show that about 47% of leases are returned using the piggy-back mechanism.

Deadlock

As discussed in Chapter 5 (see 'Deadlock Recovery' on page 84) it is possible for the file system to become deadlocked. Deadlock is detected by the server and recovery occurs by the server issuing an abort command to one of the clients in the deadlock. That client must return all its leases.

Deadlocks are detected by looking for a loop in the 'wait-for graph' as shown in figure 1.2 (page 7).

Deadlock detection makes use of records kept by the server about the state of leases and clients that are waiting for a lease. Each time the server grants a lease, a record is made of the client to which the lease has been granted. Each time a lease is requested that can not be granted, because the lease is held by another client, a record is kept indicating that the client is waiting for that lease.

Each time a lease can not be granted, the server checks these records to discover which client currently holds the lease. If that client is itself waiting for a lease, the check is repeated for the new client. This process continues until a client that is not waiting, or a loop, is found. If a loop is found a deadlock has been detected and the reply to the lease request indicates that the client should abort the current attempt to perform the system-call and return all locks.

By requiring the client that creates the last link in the deadlock to return all its leases one, or more, of the other clients that participated in the deadlock can make progress towards the end of their system-call. This means that some progress towards a solution will happen and indefinitely repeating deadlocks (known as dynamic deadlock or 'live-lock') will not occur.

The frequency of deadlocks in the system varies greatly depending on the amount of concurrency that is occurring and the way data structures happen to have been stored in blocks. Experience with BB-NFS has included sessions with no deadlocks through to

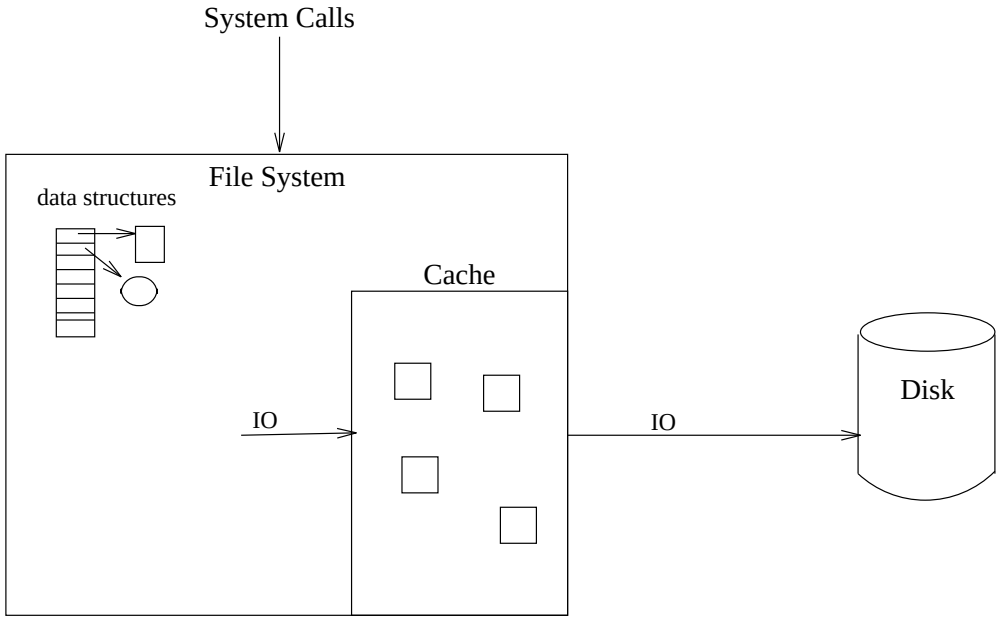


Figure 6.3: File System Outline

session with deadlocks being detected and resolved a few times a minute.

File System Transactions

Figure 6.3 illustrates the operation of the original Minix file system. The file system receives requests to perform some system-call on behalf of an application process. It uses its internal data structures, including the cache, and the disk sub-system, carry out that operation.

As explained in the previous section, deadlock recovery means that sometimes a client receives an abort message in response to a lease request. Receiving an abort means that the client can not complete the system-call it is performing. It must return all its leases, undo the effects of the partly completed system-call and then re-attempt the call.

Figure 6.4 shows how BB-NFS supports this. The Minix file system has been modified so that it is transaction oriented. A system-call being processed by the file system can be aborted at any time before it is complete. An aborted system-call will have no effect on the state of the file system, including the client's internal data structures and data stored on disk at the server. This allows an aborted system-call to be restarted as many times as is needed without corrupting the file system.

When the file system begins to process a system-call it calls an internal routine that begins a new transaction. When the system-call is complete an end transaction call is made. If

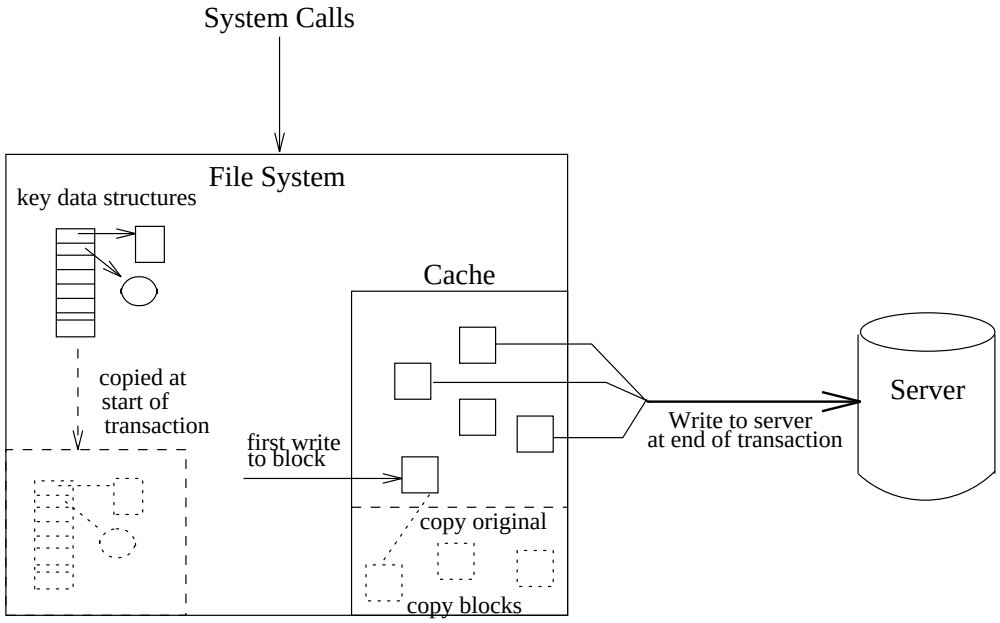


Figure 6.4: Transaction Based File System

at some point during the processing of the system-call an abort message is received the abort routine is called.

The main function of the begin transaction routine is to copy long term data from the system data structures that can be changed by the system-call. If the transaction is aborted these data structures can be restored to their original state from the data copied. The data copied includes:

- Information associated with active inodes, including how many processes are using the inode and whether the cached copy differers from the copy on disk.
- The current file pointers.

- The file system part of the process control block, which contains information about the currently active process including its working directory, pointers to the files it has open and the current state of the process, e.g. active or suspended.
- The message which the application process sent to the file system to request the system-call. This contains the parameters to the system-call.
- The status of the blocks in the cache. This indicates the number of processes that are currently using each block and weather a block is a clean copy or has been modified.

When a transaction includes a write operation the content of a block in the cache is modified. These modified blocks are held in the cache until the end of the transaction so that the changes are not visible to other clients if the transaction is aborted.

Writing to the cache means that the content of the cache will be changed by a partly completed transaction. If a transaction abort occurs these blocks will need to be returned to their original values to ensure that the partially completed transaction has no effect. Also some of these blocks may have been locked by previous transactions and the lock cached. When the abort occurs these locks must be returned along with the data for the block as it was before the aborted transaction began.

The first write operation to a particular block during a transaction causes the cache to make a copy of the original block. In the event of an abort the original copy can be used to restore the block to the state it was in before the transaction began. At the end of a normal transaction block copies are removed from the cache. The transaction mechanism is implemented at the client. No support from the server beyond that already described is required to implement client transactions.

The size of all the data items mentioned is bounded by the nature of the file system. In both the original Minix file system and BB-NFS (and most operating systems) the active inodes, file pointers and cache blocks are all stored in fixed sized tables. This limits the amount of information that must be stored at the start of the transaction.

The number of blocks modified during a transaction is limited because each transactions contains a single file system-call and, with the exception of the write call, only a few blocks can be modified at a time. BB-NFS limits the size of the buffer passed in the write call to that which can fit in the cache. In practice this did not cause any program to fail. A production implementation of a block-based distributed file system could avoid this limitation by breaking the write operation into smaller transactions.

There are some performance implications of making the file system transaction based.

The primary measure of the distributed file system performance is the number of messages exchanged with the server for a given workload. This will not change as a consequence of making the client transaction based. The messages will be bunched together but this is largely already the case in Unix systems. There will, however, be an increase in the number of CPU cycles required to run the file system because of the extra copying at the start of a transaction. Given that clients using a distributed file system normally have spare CPU cycles (see the discussion on page 21 of chapter 2) this is not expected to have a major performance impact. As already mentioned no changes are required at the server so no extra server CPU cycles are required.

6.6 Data blocks

Although the architecture does not constrain the consistency mechanism for data blocks, an actual implementation must support at least one type of consistency for normal data blocks. Like NFS, the file system uses a time based cache validity mechanism (see 'Cache Validity Check' on page 48 of Chapter 3 for more on the cache validity check algorithm).

This approach was adopted because of its simplicity and proven usefulness in a wide range of situations. A cache validity check is not, however, strong enough to meet the needs of all applications. The block-based approach does not prohibit the use of other consistency methods if appropriate. Chapter 3 contains a discussion of these issues.

6.7 Consistency

The preceding sections explain the block level locking facilities used by the file system. In the next sections the way these facilities are used to protect the file system meta-data is explained.

The file system contains the following structures[18] which must be protected against concurrent update:

- inodes
- directories
- bit maps

[18] Appendix B contains an introduction to the structure of the Minix file system.

Inodes

Inodes contain all the information *about* a file except for its name(s). This information includes the block map of the file, the number of names (links) the file has, the size of the file and the ownership and access control information for the file. In Block-Based-NFS inodes are 32 bytes long. 32 inodes are stored in a disk block.[19]

This information must be protected against concurrent update. For example, if there are two concurrent writers to a file, on different clients, the system must avoid the situation where each client caches a copy of the inode, adds a block to the block map and then writes the modified inode back to the server. This would cause the loss of one write to the file and cause the loss of a block in the file system. See figure 3.1 on page 37 and the surrounding text for further discussion of the lost update problem.

Because the BB-NFS distribution interface only supports whole block locking it is not possible to lock a single inode. Instead, the whole block of inodes must be locked. Care must be taken to avoid deadlock if two inodes in the same block need to be locked.

To implement inode locking, each client maintains a list of inodes that it has locked and the blocks that they belong too. If an inode needs to be locked and another inode in the same block is already locked, a lease will already have been granted for that block and so no new lease request is generated. When all inodes in a block are unlocked, the lease is returned.

The possibility of locking inodes in a block which is already leased does not mean that the lease might need to be maintained indefinitely. At the completion of a file system-call, all leases will be available for return. Multiple concurrent inode locks only occur during the execution of a single file system-call.

Soft Inode Updates

Some updates to inodes are not critical. If these 'soft' updates are overwritten by another update that occurs at about the same time there are no undesirable consequences. In particular this is the case for updates of the times kept in the inode. If a later update of the inode overwrites an updated time, it will also include more recent times and the loss of the earlier update is not an error.[20]

[19] Appendix B has a detailed description of an inode.

[20] In common with other distributed Unix file systems, poorly synchronised clocks could cause the times recoded in an inode to appear to go back in time when an update occurs. The more recent time will still be recoded even if it indicates a time that is earlier than the previously recorded time.

Some inodes have many soft updates. For example, the last accessed time of the root (/) directory in a Unix file system changes each time a file name is processed. It is important to avoid locking the inode every time a soft update occurs. When a soft-update occurs the file system records the need to update soft information each time it changes, but only performs the update when the file system finishes using the inode.

Directories

Unix directories contain entries with two components, a name and the inode number for the named item. Directories are implemented as special purpose files and, as such, have an inode.

When a directory is to be modified, i.e. when a file is added to, or removed from the directory, the directory must be protected against concurrent update. This is done by locking the inode for the directory.

Bit Maps

There are two bitmaps in the file system:

- **The zone bitmap** which records which data zones[21] are in use.
- **The inode bitmap** which records which inodes are in use.

These bitmaps are used heavily when the file system is being modified. To improve performance, the Minix file system keeps these blocks in memory and copies them to disk when they are modified. This approach can not be used unmodified in a distributed system because the memory copies will become inconsistent when they change.

BB-NFS maintains an unlocked copy of these blocks in memory. When a free item is required, the unlocked bitmap is searched for a free entry. Once a potentially free zone or inode has been found, coherence is established by obtaining a lease for the bitmap block that contains the bit for that item. If the bit is still found to be free once the lease is granted, the bit is set and the lease is freed.

Returning an inode or a zone causes a lease to be requested for the bitmap block containing the bit for that item. The bit is then reset and the lease freed.

[21] A zone is a collection of blocks allocated to a file as a single unit.

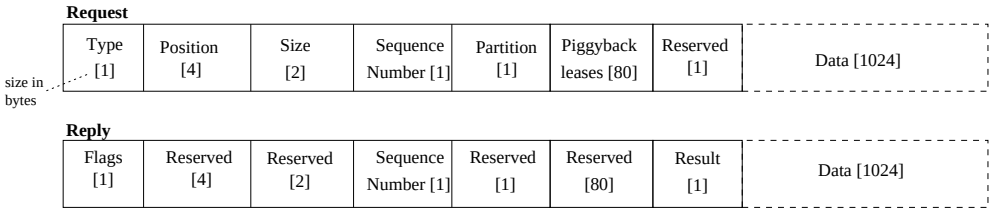


Figure 6.5: Message Formats

6.8 Communication

In Chapter 2 it was noted that the block-based approach can be supported with a very simple protocol (see page 23). In addition to making the implementation easier this provides two advantages: lower latency and efficient internetworking.

The file system uses a special purpose, light weight, protocol that is encapsulated within the UDP and IP protocols from the TCP/IP protocol suite. The use of these standardised protocols allows interworking through the world-wide Internet.

Protocol

The messages used to support remote access in BB-NFS are shown in figure 6.5. The fields in these messages have the following meanings:

Partition

The disk partition on the server that the read or write operation acts on.

Position

Which block in the partition is being read or written.

Sequence Number

Like most distributed systems BB-NFS requires *at-most-once* semantics[30] from its protocol. This is achieved by including a sequence number in the message. The protocol then ensures *exactly-once* operation by retransmitting operations for which no reply is received.

Type

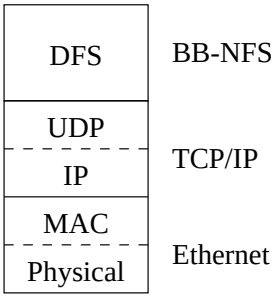


Figure 6.6: DFS on UDP/IP

This field has six main values, reflecting the six operation types. It also carries any flags that are part of the request; for example the end of update flag that indicates that writes that are part of an Unstable Atomic Update should now be committed. See the discussion of Unstable Atomic Update on page 111 of this chapter.

Result

This field indicates that the operation has been successfully completed or it contains an error code that indicates why the operation could not be completed.

Flags

This field indicates any special operations that are part of the reply, such as a short term lease.

Reserved

This space is reserved to allow for future expansion of the protocol. It is also used to make fields with similar functions occupy the same positions in the request and reply messages.

Data

A disk block. This field is present in write requests and read replies.

Implementation of the Protocol

The block-based distribution interface protocol is implemented on top of the Internet UDP[175] and IP[177] protocols. Because of the design of IP and UDP and the limited set of features required by the distribution interface protocol, it is possible to implement the two extra layers of protocol in a very efficient way.

Conceptually the protocol stack contains four independent layers: the LAN, IP, UDP and the block-based distribution interface protocol layers. The distribution interface PDU is

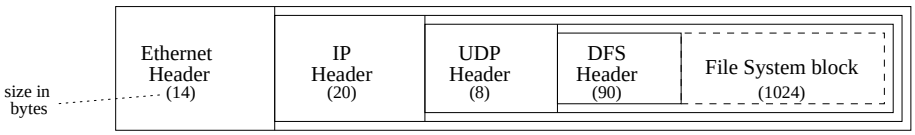


Figure 6.7: DFS within UDP/IP

carried within a UDP datagram, which is carried within an IP datagram, and this, in its turn, is carried by an Ethernet frame (see figure 6.7). The PDUs of all the protocols in this stack are structured as a header followed by the user data, which is the PDU from the layer above. This results in the series of headers shown in figure 6.7 followed by the BB-NFS data block.

By selecting UDP as the transport layer protocol and by making an appropriate choice of protocol options, the headers of the UDP, IP and Ethernet layers are constant in BB-NFS. The details of these choices can be found in Appendix C ('Protocol Headers') on page 164. By choosing a set of protocols and options that give constant headers, a very efficient implementation of the protocol stack is possible.

When a particular remote system is selected for communication, the headers can be assembled and stored in memory. When transmission of a PDU is required, it is stored in memory immediately after the headers. The start of the Ethernet header and the total length of the Ethernet frame, including the IP, UDP and Block BB-NFS components is then passed to the Ethernet interface card. The card generates the frame check sequence and appends it to the end of the Ethernet frame and transmits the Ethernet frame, including the Ethernet, IP and UDP headers and the block-based distribution interface PDU. The presence of the IP and UDP layers of the protocol stack, adds no additional per-frame processing beyond movement of the headers to the Ethernet card. This is very small compared with an independent layer implementation.

In essence this is a customised use of the TCP/IP protocols. While this approach could not be used to provide a general TCP/IP service it provides all the support required to carry the BB-NFS protocol. Interworking was tested extensively in the LAN environment and was also tested internationally by operating a server in New Zealand and a client in Australia. Although geographically close, at the time this test was undertaken, New Zealand and Australia were only connected through the United States.

An experiment to measure the cost of encapsulation was undertaken. A BB-NFS message, carrying a dummy data block, was sent from a client to a server which then generated a reply and returned it to the client. The process was repeated 10,000 times and the total time taken measured. The time taken to copy the dummy data block to the Ethernet card

was not included in this experiment because it is highly hardware and implementation dependent (a Pentium DMA based copy to a 16 bit Ethernet card might be orders of magnitude faster than a 386 CPU based copy to an 8 bit card). The experiment was repeated using an IP/UDP encapsulated protocol (over Ethernet) and a raw Ethernet service. The measurements showed that the encapsulated protocol was around 2%[22] slower than the raw service.

An implementation of this kind would not be possible with a more complex distribution interface. If, for example, data items larger than an Ethernet frame needed to be sent, some fragmentation and reassembly would be required and this would involve modifications to the headers. This in turn would require recalculation of the header error checksum at the IP layer for each IP datagram.

Other Protocol Requirements

In addition to IP and UDP the implementation of BB-NFS requires the ICMP[176] ARP[173] and Bootp[57] protocols from the Internet suite.

Although conceptually a separate protocol, ICMP (the Internet Control Message Protocol) is a required part of an implementation of the IP protocol. It supports low level management of the IP protocol, including reporting errors and testing connectivity.

The ARP (Address Resolution Protocol) is used to discover the correct Ethernet address to use for a particular IP address. The BOOTP (Bootstrap Protocol) is used to allow a client to discover its IP address at boot time.

6.9 Recovery

A practical requirement of a distributed file system is that it must able to recover from failure of one or more of its components with reasonable ease. As explained in Chapter 1 systems vary in the degree of fault tolerance they support (see "Availability" on page 4).

BB-NFS is not intended to continue its service when a component fails and is not immune to all Byzantine faults. It does, however, offers simple recovery after the most common types of failure; when the client, server or network unexpectedly stops operating.

[22]Even 2% is larger than was expected. The size of the increase is probably due to the current implementation not using DMA to copy the headers to the Ethernet card.

Locks

Because leases are used, recovery of lock state after crashes is simple. When a client crashes, no lease return will occur for leases held by that client. Other clients waiting for these leases will experience a delay while the leases timeout. Once the leases expire other clients' operations will continue.

If the server crashes, clients will be unable to retrieve new blocks or to return leases that they hold, until the server recovers. When the server does recover it is important to ensure that new leases are not granted for blocks already locked. To ensure this, the recovery time for the server must exceed the lease lifetime.

File system-calls normally complete within a few tenths of a second. This is the longest time that a lease is needed. However the effect of the lease cache extends the time that a lease will be held by the client. The measurements in the next chapter indicate that a lease cache that holds leases for about 10s is appropriate. In normal operation, a lease will not be held for longer than just over 10s. A wide margin of error has been introduced by setting the lease lifetime to 40s. Even an immediate reboot of the server takes much longer that 40s, so no direct action is needed to ensure that existing leases are not reissued after a server crash.

Network failures can be considered in three classes: transmission errors which affect a small number of messages, disconnections which last for more than a few minutes and transient failure where a few seconds or minutes of messages are lost.

Transmission errors are handled by the client retransmitting its request. If the server's reply is lost in transmission the server will see two consecutive message with the same message sequence number. The server takes no action on the second message but re-transmits the reply. If a disconnection occurs both the client and server will undertake recovery as if the other had failed. In the case of a transient failure one of the two actions above will occur depending on the length of failure.

A failure might occur when some blocks that changed during a transaction have been sent to the server, but before they all have. This might leave the file system in an inconsistent state. This problem is addressed by Unstable Atomic Update.

Unstable Atomic Update

As explained above, a fault might leave the file system in an inconsistent state. Failures in centralised file systems can cause similar problems. The `fsck` command was developed to allow centralised Unix file systems to recover from this type of error. `Fsck` checks the file system data structures for consistency and if errors are found allows them to be corrected in a way that normally preserves the content of the file system. `Fsck` can also be used in the distributed environment. However the existence of client and network failures are additional causes of file system inconsistencies which mean that failure will be more common in some distributed environments. An additional complication in the distributed environment is that it is necessary to shut down the server while recovery is happening and this will frustrate the users of clients that did not fail.

BB-NFS supports a mechanism that ensures the integrity of the file system is maintained, even when a client or the network fails. A modification of Lampson's atomic update scheme[120], which has been given the name "Unstable Atomic Update," is implemented to support this.

The algorithm works by having the server delay all write operations until it has all the data to change the file system to a new consistent state. Instead of writing data to disk immediately it is placed in a *quarantine*. When the transaction is complete the server writes all the modified blocks to the disk (and releases any leases associated with the blocks).[23]

If the network fails, or the client crashes, part way through a transaction no end of transaction message will be received. The leases on all the modified blocks will timeout and the quarantined data is discarded.

Unstable Atomic Update is weaker than Lampson's Atomic Update because it does not protect against server crashes. While the server accumulates write operations it builds an 'intentions' list in the memory of the server. If the server fails, this list is lost and the transaction fails. A crash while this list is being built leaves the file system as it was before the transaction. If the server crashes part way through writing the blocks from the intentions list, the transaction is only partly executed and the file system might be left in an inconsistent state. It is this violation of the atomic nature of a transaction that gives *Unstable* Atomic Update its name.

[23]This mechanism, which operates in the server is distinct from the transaction mechanism in the client (discussed on page 100). The quarantine in the server ensures that no updates are made to disk until all updates arrive at the server. The client transaction mechanism allows the client to repeat file system operations if necessary. Lease caching means that the two mechanisms can not easily be combined. Unstable Atomic Update operates on a longer time scale than the client transaction mechanism.

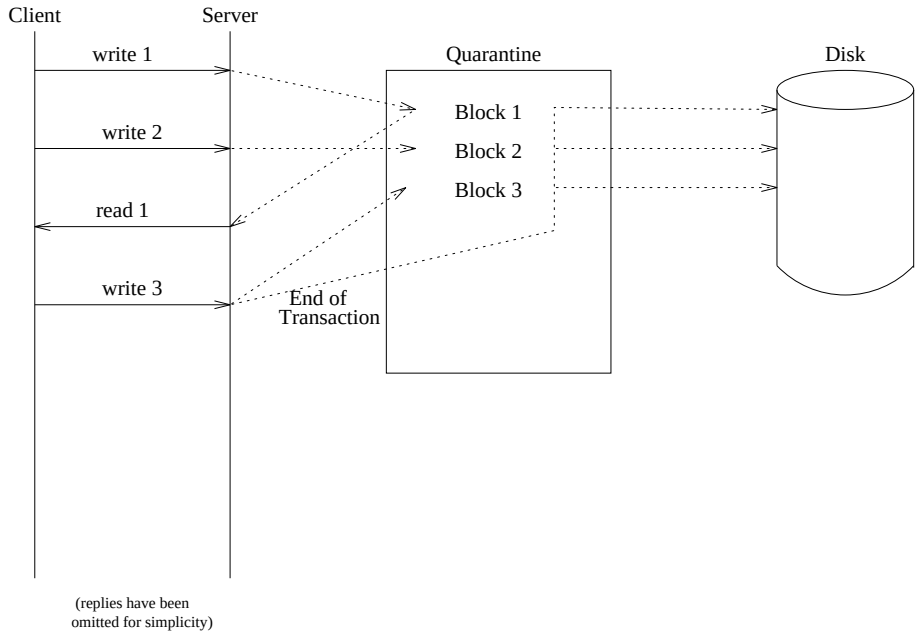


Figure 6.8: Unstable Atomic Update

It would be possible to protect the file system against server failure by using stable storage[120] for the intentions list. However, stable storage requires that at least three extra write operations occur for each write operation placed in the intentions list and is based on assumptions that are difficult to meet without writing the intentions list to two different disk drives.

(Full) atomic update is not provided by the BB-NFS because the cost does not justify the advantages gained. Unstable Atomic Update means that only server crashes have the potential to corrupt the file system; network failures and client crashes leave the file system in a consistent state.

There is a performance penalty associated with Unstable Atomic Update. When a lease is returned, the block associated with it is not available to other clients until the end of the atomic transaction. This causes an increase in latency for any client waiting for the lease. For this reason the use of Unstable Atomic Update is optional and can be enabled or disabled depending on whether optimum performance or file system stability is preferred.

6.10 Miscellaneous

This section describes some modifications to the Minix file system that were made as part of the project but are not central to the operation of a block-based network file system.

/tmp and /etc Partitions

Some Unix commands do not operate correctly in a distributed environment [174][220][100]. There are two common causes of problems:

- Temporary files that include the process identifier.
- Concurrent access to administrative information.

Many Unix programs create temporary files in the directories `/tmp` and `/usr/tmp`. In an attempt to ensure that these files have unique names, some programs include the process identifier (PID) of the process creating the file, as part of the filename. Because the operating system ensures that PIDs are unique, this scheme works well if the `/tmp` and `/usr/tmp` directories are only accessible from a single machine. If these directories are available to more than one machine the algorithm may fail, because PIDs are not unique over multiple machines.

The second problem concerns administrative files. There are a number of files that record information about the operation of the system or that are specific to a particular client. These include the table of currently mounted file systems and the accounting information. Unix was designed on the assumption that there is only one client, so it assumes that a single file for these purposes is satisfactory. If there are multiple clients this is not the case.

In other network file systems, such as Sun's NFS, these two problems are normally solved by allocating a separate disk partition to each machine for each of the temporary and administrative directories. This procedure can also be used with BB-NFS. The provision of suitable temporary space sometimes presents problems for system administrators. Enough space needs to be allocated for the maximum needs of each of these directories for each of the clients. Allocation of separate administrative directories is also wasteful because much of the administrative information is the same on all clients.

BB-NFS provides "Divergent Lazy Partitions" as an alternative solution to these problems. A number of other systems use lazy copy[24] including Tenex[35] which was the first system to do so. Tenex used lazy copy in its paging system to allow large programs to be shared without a complete copy of each needing to be kept in memory. Sprite[151] also uses lazy copy at the file level. BB-NFS's divergent lazy partition mechanism applies the lazy copy technique to file system partitions. It allows the server to offer a unique copy of a partition to each client that accesses it remotely. Changes made by one client are not

[24] Lazy copy is also known as copy-on-write.

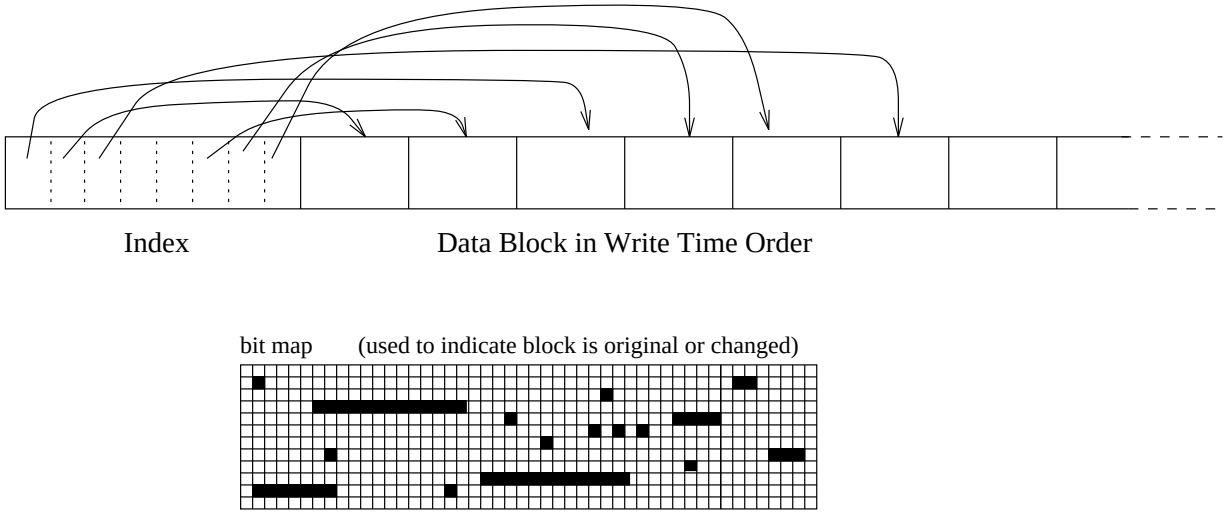


Figure 6.9: Lazy Divergent Partitions

visible to others. Updates cause the views of the partitions to diverge, each client seeing only its own modifications.

The server offers this facility by opening two partitions on its local disk. The first contains a read only copy of the unmodified file system. The second is used as a log-structured[158] store for modified blocks. When a client performs a write operation, the modified block is written to the modification partition and an index to it is made in the data structure shown in figure 6.9.

When read requests are received, the modification bitmap is checked to see if an original or modified block should be used.

It is possible that some files might be changed and then brought back into alignment with the original file. The lazy divergent partition mechanism does not consider the content of blocks, just that they have been written so it will continue to treat the data as a divergent. A utility program checks the contents of the copied blocks against the original partition and removes identical blocks from the copy log. It also removes blocks from the log that belong to files that have been deleted by the client that owns the log.

Instrumentation

To better understand the behaviour of the system and to check for the occurrence of unusual situations, a kernel instrumentation package has been developed. The package contains two components, an event counting facility and a message reporting mechanism.

Counters within the kernel record the occurrence of a number of interesting events. A new

system-call has been added to return the current value of these counters, and the status of the cache buffers. The events include counts of read and write operations, the number of cache hit and misses and the number of lock operations performed. See table D.1 (in appendix D) for a complete list of the counters.

Logging

At times it is useful to generate a message from the kernel to note the occurrence of some unexpected, but non-fatal event. It is particularly useful to be able to do this during kernel development. Minix contains a facility to display kernel messages on the console but these messages are easily lost, particularly if a large number of them are generated. Also screen messages do not provide any information about the sequence of events that occur on different machines.

A remote message facility has been added to Minix that allows a client to generate a message that is sent to the server and recoded there. Two forms of logging are supported. Kernel messages, in addition to being displayed on the console, are sent via the log operation supported by the distribution interface (see page 92).

Log operations are easy for the kernel to generate but are not available outside the kernel. For application processes that wish to log information at the server a special device has been installed. If data is written to the special file[25] 3/20, which is normally installed as `/dev/log` the data written will be logged at the server in the same way as a log operation generated within the kernel.

6.11 Complexity

Table 6.2 shows the number of lines of code added or changed to implement BB-NFS.

As expected the most difficult part of the implementation was correctly identifying critical code and implementing the synchronisation mechanisms in the kernel. Much effort was required to located and rectify concurrency errors. Often the only symptom was eventual corruption of the file system. The next most time consuming aspect of the implementation was building the low level network drivers, which are specific to the particular Ethernet cards in use.

[25] 'Special file is a Unix term referring to a file that interfaces to an IO device.

Function	Lines of code
Server	1242
Cache	835
Communication –	
Startup	903
Main code	248
Ethernet device driver	1437
BOOTP	135
ARP	103
ICMP	84
Locking	852
Divergent lazy directories	111
Event Counters	117
Other	118
Total lines changed or added	6185

Table 6.2: Lines of Code

6.12 Testing

Although it is not intended that BB-NFS be used for purposes other than the testing of the principles of the block-based approach and for collecting information about it, extensive testing was undertaken. Because Minix was developed to be experimented with, it is supplied with a set of rigorous test programs.

These test programs were run on a single client/server system and then concurrently on three systems. When run concurrently some tests interfere with one another and consequently they report some errors. The file system is left in a consistent state at the end of these test which is the result of concern. It would be possible to modify the testing programs to work in different directories and not interfere with other concurrent tests but this would reduce the interaction between them and might not uncover file system concurrency bugs that would be found when the tests interact.

Like most operating systems BB-NFS does have some outstanding bugs but these appear infrequently. Substantial use, including periods of heavy continuous use for a day or more, and amounting in total to many days of constant activity, was made of the system while generating the measurements reported in the next chapter.

6.13 Summary

The beginning of this chapter listed the most important reasons for implementing BB-NFS. These were: to test how implementable the block-based approach is; to allow the amount of locking traffic to be measured; to look for patterns of behaviour that can be optimised and to explore issues associated with block naming.

That BB-NFS was successfully implemented within the limits of the current project is a practical demonstration that a system of this type can be built and that the effort involved is reasonable.

The implementation has allowed the behaviour of a block-based distributed file system to be studied. The next chapter presents measurements of the cost of consistency. It also contains a discussion of the lease cache which is an optimisation included after observing locality in access traces taken from the implementation. Other optimisations may be possible.

Block naming has not yet been studied in detail, however the presence of BB-NFS is a major asset for an investigation into appropriate naming mechanisms in the future.

Chapter 7

The Cost of Consistency

For a distributed file system to be useful it must both operate correctly and have acceptable performance. There are two main reasons why a distributed file system will perform differently to a local file system. The first is the cost introduced by the need to use a network to access data. The second is the need to perform consistency checking. There are other issues that affect the performance of the system but their effect is small compared with these two issues. For example in BB-NFS the CPU time used at the client to make copies of blocks in the cache will reduce performance a little. Assuming the timings given by Dahlin *et al.* (see 'Latency' on page 29 of Chapter 2) and a 100MIPS CPU around 3,000,000 instructions can be executed in the time needed to send two Ethernet messages and perform one disk access.

It is harder to predict the impact of block level consistency on the performance of BB-NFS. The bulk of this chapter presents the results of an investigation into the performance effects of BB-NFS's block level consistency mechanism. Chapter 6 contains a description of the way BB-NFS uses leases to enforce consistency (see the discussion beginning on page 95).

There are three questions addressed by this chapter:

1. Is a lease cache required?
2. If a lease cache is required how long should leases be held?
3. How many extra transactions are required to support block consistency?

```
r0188 r0189 r0191 r0192 r0193 r0194 r0201 r0203 r0207 r0210
r0226 r0248 r0249 r0250 r0251 r0252 r0253 r0254 r0255 r0256
r0257 r0131 r0005 r0149 r0006 r0196 r0231 r0232 r0264 r0265
r0266 r0267 r0268 r0269 r0270 r0680 r0018 r0017 r5134 r5194
r5185 r5186 r5187 r5188 r5189 R0018 r5136 r5183 W0018 R0018
R0003 W0003 W0018 R0018 W0018 R0018 W0018 R0018 R0003 W0003
W0018 R0018 W0018 R0018 W0018 R0018 R0003 W0003 W0018 R0018
W0018 r5190 r5191 r5192 r5195 r5196 r5199 r5200 r5201 r5202
r5203 r5204 R0018 W0018 R0018 R0003 W0003 W0018 R0018 W0018
```

key: r = read, R = locked read
w = write, W = unlocking write

Figure 7.1: Transaction Trace Without a Lease Cache

7.1 The Importance of the Lease Cache

Early in the development of BB-NFS it was noted that some sequences of locked read followed by unlocking write are repeated frequently. Figure 7.1 shows a trace of the operations performed by an implementation of BB-NFS without lease caching. Notice the repeated reading and writing of block 18 and block 3.

This behaviour arises because the implementation attempts to hold leases for the minimum time required for correct consistency even though activity in the file system is often clustered. The lease cache is intended to reduce the effect of this behaviour by allowing leases to be held beyond the minimum time they are required.

Figure 7.2 shows the effect of the lease cache on the number of read or write transactions required to execute the Andrew benchmark (see 'The Andrew Benchmark' on page 57 in Chapter 3). Even holding leases for one second produces a large drop (74%) in the number of transactions required to execute the benchmark.

With a 10s lease cache there is a drop of 76% and only 5% of the locking transactions that occur without a cache remain.[26]

7.2 Lease Holding Time

Given that a lease cache is effective, it is necessary to choose the length of time a lease should be held for. There are two opposing, time related, factors which have an impact

[26]There is a reduction from 94563 transactions to 22429 transactions. Both these figures include the 18608 transactions required to execute the benchmark without locking.

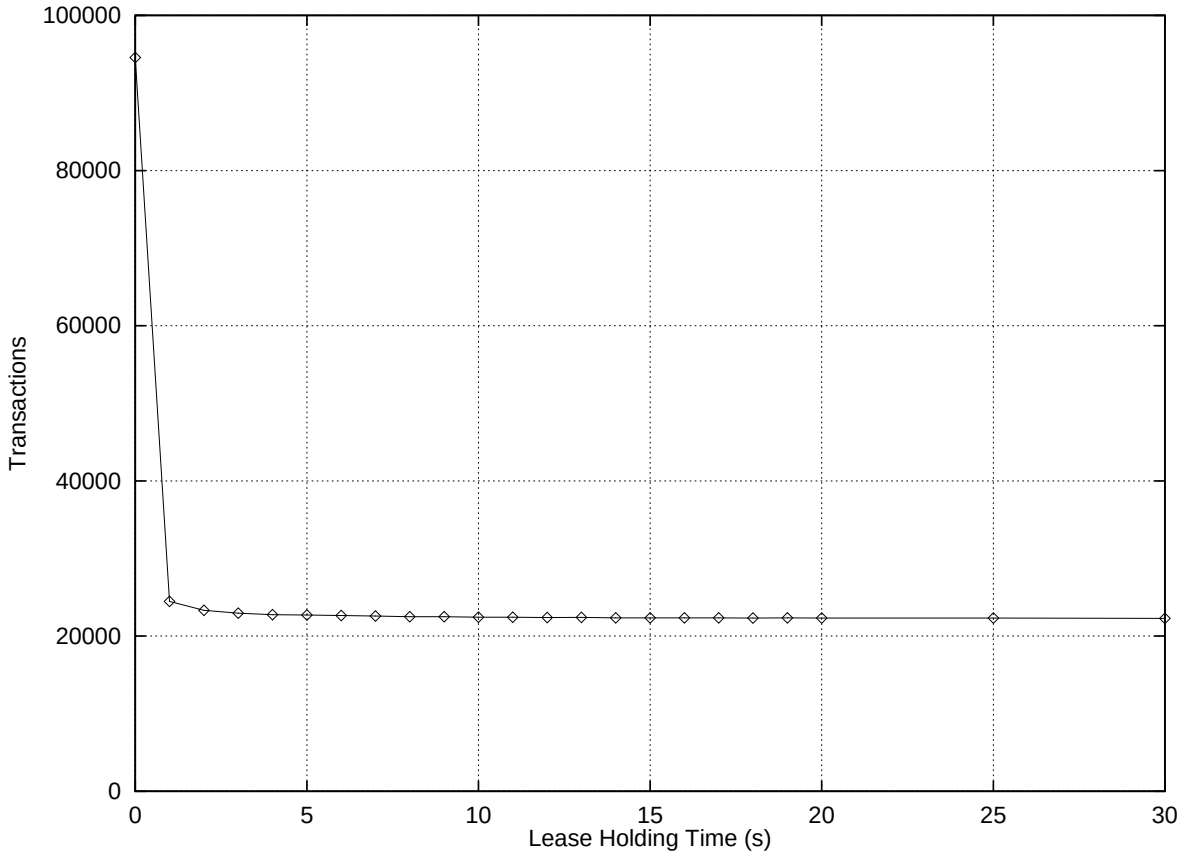


Figure 7.2: The effect of a lease cache on the number of transactions required to execute the Andrew benchmark, on a single client.

on the effectiveness of the lease cache.

The first is the probability of the block being reused. The longer the lease is held the greater the chance that the block will be reused in that time. If it is reused, a pair of distribution interface operations (a locked read and an unlocking write), or four network messages, are saved. The opposing factor arises because the longer the lease is held the greater the chance that the block will be needed by another client. In this case the lease will need to be revoked requiring a revoke message to be sent to the client that holds the lease.

A third factor limits the maximum lifetime of a lease. Minix (like most Unix implementations) restricts the time a modified block can remain in the cache to at most 30s. This is done to reduce the loss of data when a crash occurs. When BB-NFS writes a locked block to the server it also returns the lease, consequently leases are never held for more than 30s.

In addition to time-related effects, the number of active clients, might be expected to affect the behaviour of the block cache. When there are more active clients there is a greater chance that a lease will be revoked. Figure 7.3 shows the number of transactions required to execute the Andrew benchmark as the length of the lease and number of active clients is varied.

This data was collected by running the benchmark simultaneously on the indicated number of clients connected by an otherwise idle network. The clients were Pentium 133Mhz PCs and the server ran on a 486 DX4-100 PC. Only 5 clients were available for measurements.

Figure 7.3: The effect of varying the time for which leases are held for on the number of transactions required to execute the Andrew Benchmark.

No. Clients	Extra Txns	%
1	3835	21%
2	7427	40%
3	10867	58%
5	13563	73%

Table 7.1: Extra Transactions Required to Maintain Consistency

Each point represents the mean of a number of executions of the benchmark. Seven samples were taken for each point, with the exception of the measurements on a single client (n=1) where only two measurements were taken because these did not vary significantly. [27] Each sample took between 5 and 20 minutes to collect, depending on the number of clients. The range of values, from the maximum to the minimum sample value, is shown by the range bars in figure 7.4. The range bars for the n=3 curve are offset by a small amount to make them easier to distinguish from the others on the plot.

The plots indicate that the precise length of time a lease is held in the cache is not critical but should be greater than a few seconds. A lease cache holding time of 10s has been used in BB-NFS.

7.3 The Cost of Block Consistency

The previous section shows that a lease cache that maintains leases for 10s reduces the number of transactions required to run the Andrew Benchmark. These measurements do not give an indication of the overall cost of the consistency mechanism. To establish this the number of disk IO operations required to execute the benchmark locally was measured. All operating system parameters, including the block size, size of the cache and file system structures were held constant. Only the networked nature (including the consistency mechanism) was changed. This measurement provides a base line against which the networked implementation can be compared.

For the Andrew benchmark the local file system performed 18608 IO operations. Table 7.1 shows the number of extra transactions required to maintain consistency (with a 10s lease cache). The measurements shown in table 7.1 approximate a worst case situation because they were taken when all clients were executing the same code at the same time.[28]

These measurements show that the overhead required to maintain consistency increases more slowly as the number of clients increases (for small numbers of clients). This is

[27] All variations were less than 0.05%. See figure 7.4.

[28] The clients were started at the same time by a synchronisation program.

Figure 7.4: The Effect of varying the Time for which leases are held for on the number of transactions required to execute the Andrew Benchmark, including range bars.

a surprising result. As the number of clients increase there is more opportunity for interaction per client because each new client can interact with all the others. The most probable reason for the observed behaviour is that as the execution proceeds small variations cause the benchmark being executed on each client to becomes less synchronised with the others. The larger the number of clients the more variations that occur. As the clients fall out of step they interact less.

It would be helpful to compare these measurements with other distributed file systems. Unfortunately comparable figures are not widely available.

Srinivasan and Mogul[208] measured NFS while running the Andrew Benchmark on a single client. In their study, `getattr` transactions, which are primarily used to maintain cache consistency, added about another 48% to the read traffic (993 `getattr` transactions and 1961 `read` transactions. Moore, McGregor and Breen studied NFS traffic in a university computing department. They found `getattr` transactions out-weighed cache read operations by nearly 5:1 under this type of workload[145][144].

Although indicative of the high cost of maintaining consistency these results are not directly compatible to the measurements present here because many of the operating system parameters (including the cache sizes used) were different. In the case of the Moore *et al.* the workload was also different. However it is clear that the block-based approach can be implemented without an unreasonably high amount of locking traffic. In the case of BB-NFS the lock cache is instrumental in achieving a low locking overhead.

Chapter 8

Conclusions and Further Research

Most distributed file systems include communication between machines at the file level. In contrast the block-based approach, proposed in this thesis, implements communication at the block level. Moving the distribution interface to the block level simplifies the interface and brings a number of other advantages, the main ones being:

- finer grain
- added flexibility
- another level of openness
- natural idempotence
- simpler server code
- improved scalability of the server
- scope for use of a light weight protocol
- efficient internetworking
- more general caching and migration
- scope for more extensive striping

Some of these issues are important for the development of future distributed systems. In particular the fine-grained nature of the block-based approach may be a key step in distributed system development.

The need for fine-grained systems is highlighted by Coulouris *et al.* who write:

> "There are some features not found in current file services that will be important for the development of distributed applications in the future:
>
> *Support for fine-grained distribution of data:* As the sophistication of distributed applications grows, the sharing of small data items will become necessary. This is a reflection of the need to locate individual objects near the processes that are using them and to cache them individually in those locations. The file abstraction, which was developed as a model for permanent storage in centralised systems, doesn't address this need well. ..." [56]

This thesis proposes an architecture for a block-based distributed file system. The architecture is described in Chapter 5. Its main features are:

- block level distribution interface
- Unix file system interface to applications
- client and server caching
- block locking with coherence for updating file system meta-data
- no security at the block level, encryption and message digests used by the file system layer
- file data coherence and replication policies left open to allow the architecture to be tailored to meet a variety of needs
- location independent block names

This thesis also describes an implementation of a block-based distributed file system. BB-NFS was implemented to determine the practicality of the architecture and to allow measurement and optimisations to be made. BB-NFS is the first distributed file system that conforms to the architecture. It demonstrates that the block-based approach is both implementable and performs well. It also allowed the behaviour of a block-based system to be studied and the design to be improved based on observed patterns of behaviour.

8.1 Further Work

There are some areas where it would be useful to further develop and study the BB-NFS prototype. The following sections discuss further measurements and development of BB-NFS.

Measurement

Further measurement is important because it will allow accurate decisions about the kinds of distributed system that would benefit from the block based approach.

- An elapsed-time based comparison of BB-NFS with other similar file systems, particularly the industry standard NFS. This would further help to establish the relationship between BB-NFS and these other systems and might highlight areas for further development.

  To be able to do this BB-NFS would need to be implemented on similar hardware and configured with similar operating system parameters to the distributed file system it is to be compared with.

- As discussed in Chapters 2 and 4 the Andrew Benchmark, although widely used, does not test the file system under a non-sequential workload. As Biswas[32][31] found random access behaviour probably exceeds sequential behaviour in commercial environments (see 'Other Environments' on page 32 of Chapter 2). An advantage of the fine grained nature of the block-based approach is that it is expected to perform better under non-sequential workloads than systems designed only with whole-file behaviour in mind.

  Construction of a benchmark that more accurately models random access behaviour and the study of BB-NFS and other distributed file system using the benchmark would be valuable.

- It is not clear that the trend for there to be less locking traffic per client as the number of clients increases (shown in table 7.1 on page 123) will continue for much larger numbers of clients. It would be valuable to investigate the cost of consistency with large numbers of active clients.

- The behaviour of the consistency mechanism is central to the performance of a block-based distributed file system. Careful study of the mechanism may reveal further optimisations. One possibility worthy of consideration is the use of optimistic concurrency control (see 'Optimistic Transactions' on page 49 of Chapter 3). The transaction nature of the current implementation of BB-NFS would make the move to optimistic transactions easier. It might also be possible to identify particular data structures that are used in a highly concurrent manner and modify the file system to reduce this usage. The bitmap blocks are an area where further performance gains might be made.

Development of BB-NFS

BB-NFS is a prototype; it is not intended as a production system. There are many improvements that could be made that would make BB-NFS more useful if it was supporting real users. This section does not address those issues but proposes extra functions that would be useful in a prototype form.

- A major step in further development of BB-NFS would be the development of a block naming strategy as outlined in the architecture (see page 86).

  A study of the way blocks are used could provide information for the implementation of a location independent block naming mechanism. In particular data for a simulation study of the performance of an implementation of the naming architecture would be useful.

- Block level migration has the potential to automatically spread the server workload between the available servers and to minimize the need to communicate with remote systems by storing data at the site where it is used most often. It would be helpful to use the prototype to collect data that could be used to study of different approaches to block level migration.

- The development of security mechanisms in the block-based environment. In particular meeting the need for flexible security without relying on the security of the hardware base is an interesting area for further study.

8.2 Future Research

The main role of BB-NFS is to enable further research into the block-based approach. The first step to this enabling was the demonstration that the block-based approach is practical. Having successfully demonstrated the reasonableness of the approach many avenues for further research become possible. Two areas which may be particularly fruitful are described below.

Serverless Block-Based Distributed File System

The separation of the resources of a distributed system into servers and clients is not always appropriate. One reason for this is the low cost of small and medium sized disks compared with the cost for very large disks. It is much cheaper to put a disk on every

machine in a network than it is to put the same amount of disk space onto a single server. Another reason is the imbalance between the CPU utilisation of the typical slave and client.

The block-based approach can address these issues by allowing many machines to cooperate to offer the block service. Potentially every machine in the distributed system could be both a client and a server.

This 'serverless', block-based approach allows the workload of serving blocks to be shared between many machines. If combined with an appropriate migration scheme it has the potential to reduce the communication needs and improve the performance of a distributed file system by locating blocks at that machines where they are most frequently used.

Hardware File Server

Because the block-based distribution interface is simple it may be possible to implement a server in hardware. This would reduce the latency of transactions, particularly those that can be serviced from the servers cache. It would also allow the server to support a much higher workload, probably higher than any other distributed file system server.

Deadlock detection, cache management and disk scheduling are the main challenges in a hardware implementation. Some of these functions do not need to be performed in the same time-frame as the distribution interface operations and could be provided by a support CPU.

Providing high performance servers is an important distributed file system design goal. One reason for this is the rapid growth of the Internet which means that an information provider can make information available which very large numbers of users wish to access concurrently. Current, software based, file servers at popular sites are unable to service the demand for files and some providers operate many high performance servers to meet the workload. In addition to the capital cost operating multiple parallel servers has a high maintenance cost.

A hardware file server has the potential to service a much higher workload.

8.3 Conclusion

The block-based approach to distributed file systems offers many advantages including some that have been identified as fundamental issues for the future development of distributed systems. This thesis has explored the nature of block-based distributed file systems. The block-based approach has been shown to be feasible through the development of a suitable architecture and the testing of key aspects of the architecture in a working prototype. Having demonstrated the feasibility of the approach and developed an implementation base many opportunities for further research in the block-based distributed file systems area are now possible.

Appendix A

Annotated Bibliography

This appendix contains an annotated bibliography of distributed file systems that were discovered during the course of the project. It also contains a small number of file systems that are not distributed but have been included because they are frequently referenced in distributed file system work or because it is expected that they will be of interest to distributed file systems researchers. These systems are noted with the symbol $^{\bar{d}fs}$ in their section heading.

Some systems which are distributed and provide persistent data storage and management but do not provide the traditional file interface are also included. These are not explicitly noted but are clear from the description.

Where a reference is included in parenthesis the reference includes information about the file system being described but is not primarily about that system.

A.1 Acorn

The Acorn file system is a simple client server distributed file system. It does not provide synchronisation services to its users. Operations at the distribution interface are idempotent and stateless and include the notion of files and directories. Read and write operations are byte aligned. Hints are used for location of objects in the server cache. There are no client caches. [65]

A.2 Alpine

Alpine is a log structured file system. It provides both a database and a file service to its users. Atomic transactions and two phase locking are also provided. Capabilities are used for protection management. The server keeps state information for open files which allows files to be locked when they are opened. To improve performance extra blocks are preallocated to files as they are extended. Alpine includes a deadlock detection mechanism and is built using light weight processes. [38] ([146])

A.3 Amoeba file service (FUSS)

The Amoeba distributed system supports four different distributed file systems. Very brief descriptions of a non-concurrent and Unix-like file system are give in [146]. No references are included. More detailed descriptions of The Free University File System (FUSS) and Bullet are provided. See the entries for FUSS and Bullet for more details.

Amoeba references: [217][147][220][218][233][236][235][232][234][230][14][19] [70][146] ([148][108][12][13][219][24][192][46][227])

A.4 Andrew (AFS)

Andrew is one of the best known distributed file systems. The main goal for Andrew was large scale but see [58] for some limitations. Whole file caching is employed although large files can be cached in 64kb chunks. Andrew provides the Unix file system interface. Only session semantics are provided. See also Coda and Decorum. [100][197][198] ([152][46][58][242][126][87][199])

An important part of the Andrew work is the Andrew benchmark which is often used for file system comparisons. [100][152][208]

Further details of Andrew and the Andrew benchmark are given in Chapter 4, beginning on page 53.

A.5 Apollo domain

Domain is a one-level storage, object oriented distributed system. Indirect objects are used to get the latest version of an object. It is built on proprietary hardware. It provides

time stamps which allow users to implement concurrency control. Hints are used for location. Domain is the origin of the concept of distributed virtual memory. [123][124] ([152][111][26][143][227])

A.6 Athena

Athena is a workstation server architecture, distributed system. It supports a single name space and is implemented on top of Unix. NFS, RVD and AFS are used to access remote files. Athena separates most services into separate entities. These include the file, name, authentication, mail and printer services. Hesiod provides the name service and Kerberos (which is now widely used outside the Athena project) provides the authentication service. Athena draws heavily on Locus. [46]

A.7 Bostryche

Bostryche is a replicated distributed file system. It supports locking and provides strict consistency. Voting is used to ensure consistency during a network failure. Where appropriate, transaction reordering occurs to improve availability during a network partition. [196]

A.8 Bullet

Bullet is a file system for the Amoeba distributed system (see the entry for Amoeba). The aim of Bullet is fast file access. To achieve this, Bullet stores files contiguously and uses whole file caching. To allow this, files are immutable. Version numbers are supported. Replication is supported and security is provided using capabilities. [233] ([220][235][192])

Further details are given in Chapter 4, beginning on page 58.

A.9 Cambridge DS

The Cambridge distributed system supports a distributed file system that provides a traditional file service to its users. Atomic updates are also supported. Protection management is provided using capabilities. The distribution interface is file based. Directory functions

are integrated with file operations and this allows automatic garbage collection. To support a user environment the single user operating system TRIPOS is run on a CPU from the processor pool for each logged in user. [69][78][186] ([106][142][123][219][146][219])

A.10 Carnegie-Mellon central file system (CFS)

CFS provided the network accessible storage facility for local file systems in the Carnegie-Mellon Computer Science Department in the early 1980s. CFS was intended to provide a 'back-end' file service to various local file systems. It provides file-server, name server and authentication services. It provides transactions that allow a group of read or write operations to be indivisible and aborted if desired. Transactions are limited to a single file and a user may only have one transaction in progress at a time. Only one user may be performing a transaction on a file at a time. Access Control Lists are provided for access management but it assumes that the network is secure. It provides both standard (one copy) and stable (two copy) storage. Files can also be classed as mutable or immutable and version numbers are supported. Semaphores are provided. The design of CFS was influenced by Multics and Hydra. It predates ITC and Andrew at Carnegie-Mellon.[3]([197][100])

A.11 Casper

Casper is a distributed persistent storage system. Its mail goal is to support concurrent access to the persistent store from client workstations. The Casper file system is intended to be highly network transparent including transparency in the performance and management area. It supports heterogeneous systems and automatic file placement and migration. Casper decouples the name space from the file system. This allows it to be replicated and managed independently of the rest of the file system. Casper appears to be a derivative of ROE. [237]

A.12 Cedar (CFS)

Cedar provides a distributed file system based on immutable files and version numbers. Whole files are transferred. Clients cache files on their local disks. File creation is atomic. [81][37][200] ([152][38][201][111][97])

A.13 Charlotte

A simple file level, remote access Unix-like file system with some simplifications and optimisations to enhance distributed working. File location is based on hashing the path name of the directory containing the file. This requires protection on a file by file basis and requires files to be deleted from both the directory and the server. The architecture of the distributed file system is seen as a weakness in [36]. [74][10]

A.14 Choices

The Choices system is based on persistent storage and distributed virtual memory. Objects are the basis of consistency, which uses a write-invalidate policy. The file system is memory mapped. Physical memory is represented as a file, the pages of which are mapped onto virtual memory regions. [43][42][134]([143][227])

A.15 Chorus

Chorus is a modular distributed system which includes support for real time applications and also includes a Unix emulation package. One module provides a remote file system. Internally it contains data structures for supporting the file system and a cache of recently used data. File operations that originate in an application, have contextual information added to them and are sent, as a message, to the file module (which may be on a remote machine). The file system supports lazy copy. Fault tolerance can be provided through multicast. Coherent shared virtual memory is also provided and services can by migrated by moving the ports at which the service is accessed.[2][86][51][9][192] ([43][227]).

A.16 Clouds

Clouds is an object oriented distributed system. It provides a single level, persistent, distributed, virtual address space. Memory can be shared between objects. When memory is shared, the paging system will fetch pages from the remote system as needed. When more than one object is using the same memory segment the kernel ensures that only one object has a copy of any particular page at a time. The kernel is structured as a light weight message passing kernel. [5][24][60][61]

Further details are given in Chapter 4, beginning on page 61.

A.17 Cocanet Unix

Cocanet Unix is an extension to Unix that supports use of remote files. The Unix naming system is extended by adding the names of remote machines as top level directories. I.E. `/ing70/A` is the name of the file `/A` on machine `ing70` as seen from some other machine in the network. The changes are implemented within the Unix kernel and as application programs but no changes are required to applications or system programs that use remote files. When a remote file is opened a server process is created on the remote system. This server then opens the file and returns the result to the original machine which, if the open was successful, allocates a file descriptor and marks the descriptor as referring to a remote file. Read and write operations on the descriptor are passed to the remote server process which performs the operation and returns the result to the original machine which then passes it back to the application. [191]

A.18 Coda

The Coda file system is designed for file systems that include computers that are frequently detached from the network for long periods of time. A laptop computer that is taken home in the evening or away on a trip is an example. This is called disconnected operation. The Unix file system interface is provided. Whole file caching is used. A long period, very optimistic, concurrency policy allows files in the cache to be read and modified even when the computer is not connected to the network. When a conflict is discovered (when a computer rejoins the network) conflict resolution is undertaken. This may be automatic (in the case of a directory) or manual in the case of a user file where no semantics are known. Authentication is token based and end-to-end encryption is used for security. Coda is a derivative of Andrew. [112] ([242][94][199]).

A.19 Concurrent file system (CFS) From Intel

CFS allows more than one processor to be accessing a file (that is stored on multiple IO devices) in parallel. It provides the Unix file system calls. Caching is performed at the IO nodes. Caching at the compute nodes is limited to the duration of an IO operation. Files can be 'declustered', that is spread across more than one volume. One IO node is

responsible for directory operations and all directory requests and updates are processed by it. [75] ([227])

A.20 Cosmos

Cosmos was intended to be a fifth generation operating system employing a knowledge base to "provide a more intelligent programming environment." The knowledge base supports typing, version control and configuration management. The kernel is object oriented and files are objects. All objects are immutable. Communication is based on message passing. [154][155][34]

A.21 Dacnos

DACNOS is a distributed operating system designed to support heterogeneous systems. It is hosted by the local operating systems of the machines that comprise the system. It uses a custom protocol (NT) where possible, but also supports OSI and TCP/IP.

The file system (RFA) gives a homogeneous global view of the local file systems but requires all modifications through RFA (not the local file system) if consistency it to be maintained. Hierarchal naming is used. RFA is implemented as a modification to the local file system allowing it to intercept file system calls. When it appears that "it is appropriate" data is transfered using a bulk transfer protocol. [79]

A.22 Dash

Dash is a distributed system architecture, designed for a time when CPUs are very cheap (10-100 per workstation); communication is high speed (0.01 - 1 Gbps), low latency (30-50ms world wide) and ubiquitous. Vertical integration and avoiding pre-existing systems are key goals. Fast, reliable IPC is at the core of the architecture. Very few details of the file system are included in published reports but it appears to be based on file servers which extend the global name space. Tokens are used to improve the performance of naming. Replication of whole servers is permitted. It is unclear whether the file system was implemented. [6][7]([46])

A.23 DCE/LFS

DCE/LFS is the file system component of the Open Software Foundations Distributed Computing Environment. It is also known as Episode and is described under that heading.

A.24 Decorum

Decorum is the name given to the Open Software Foundation's (OSF) Distributed Computing Environments (DCE) File System. It is a derivative of Andrew. Decorum contains three components, the file server (also known as the protocol exporter), the Episode (or DCE/LFS) physical file server and the cache manager. Single copy semantics is provided using tokens. Although it is intended to be supported by Episode other file systems can also be used. Deadlock is avoided through partial ordering of locks. The system is POSIX compliant. See also the description of Episode. [110][52][125]

A.25 Dunix

Dunix is a location transparent implementation of Unix. All system calls and names are location transparent. The file system name space is like the traditional Unix name space but with the addition of a 'super-root' directory (...). Path names come in three variants, working directory relative, root relative and super rout relative, the latter two starting with / and .../ respectively. Dunix deliberately excludes heterogeneity. The key concept in Dunix is system-wide UIDs. UIDs contain the identifier of the computer that provides low level services for that device. File system operations are handled like other system calls. That is when a system call is made and the UID is translated to the computer providing low level services for the file and the operation forwarded there. The result is returned to the high level code which then replies to the caller. Dunix appears to be a derivative of MOS. [129]

A.26 Eden

Eden is an object based system with object based consistency. Long term storage is provided through object check pointing. Security is managed using capabilities. Both migration and replication are supported. Like many object based systems Eden is reported

to perform poorly. Emerald appears to be a derivative of Eden. [4]([219])

A.27 EFS

EFS (Extended File System) is an extension of the Unix file system. It provides transparent access to files on a remote computer. It is implemented within the kernel. The client and server are connected by any Ethernet. File operations are sent to the remote machine for execution. Remote trees can be mounted on the local machine with an extended version of the mount command (rmount). EFS uses a remote form of Unix inodes (rinode). The rinode contains the address of the remote machine and a memory address of an 'in-core' inode on that remote machine. [53]

A.28 Emerald

Emerald is an object oriented distributed system. The main focus of Emerald is support for object migration which is triggered by the programmer. Emerald supports small objects and hence fine-grained migration but the same object model is also used for large objects. Emerald appears to be a derivative of Eden. Emerald is both a distributed system and a language. Emerald has a more sophisticated compiler than Eden, which is able to detect whether an object is local or remote and generate the appropriate code for each case. [127][33]

A.29 Episode

Episode is the physical storage file system for the Open Software Foundation's (OSF) Distributed Computing Environments (DCE). It features log structured crash recovery, copy-on-write, Kerberos security and token based consistency. Files are generalised as containers which are used to store file data as well as bitmaps and the meta-data log. [52][110]([125])

A.30 Felix

The Felix file server provides a simple flat file service that is network accessible. It is designed to support both user files and virtual memory. It provides atomic update over

one or more files and appears to use lazy copy (see [76] page 39). It maintains version numbers for files. UIDs are used as capabilities for files. Felix provides low level services to Helix[77]. [76]([77]).

A.31 Ficus

Ficus is a large scale distributed file system. The main goal for Ficus is to be able to scale to millions of sites. Ficus draws on some of the techniques of Locus. Other goals include simplicity, ease of use, file system boundary transparency, syntax transparency with respect to existing file system types, location transparency, name transparency, local autonomy for name management and compatibility with embedded names in existing software. Because of the need to support partial operation Ficus is also suited to disconnected operation; it supports large scale replication and optimistic concurrency control. It uses a one copy availability (OCA) consistency mechanism. Ficus does not meet the needs of applications that process large files with random access and frequent updates. It only allows replication at a coarse level. [87][162][184][94][95]([185])

Further details are given in Chapter 4, beginning on page 65.

A.32 Frolic

Frolic is an attempt to meet the needs of large-scale non-academic distributed systems, where sharing of unstable files is common, as found in [82]. It is based on the notion of clusters of workstations with widespread automatic replication of files. Replicas are created when a file stored in another cluster is accessed. Only one cluster has write access to a file at a time. Frolic only offers session semantics for consistency. Replica update is achieved by invalidating all copies except the updated one and one other. The other copy is sent updates to the file. [163][195]

A.33 FTAM

File Transfer, Access and Management (FTAM) is the file system component of the International Standards Organisations (ISO) standards for Open System Interconnection (OSI). It is oriented around a virtual file store which is a generalised abstraction of real file systems. Computers which provide an FTAM service must map their local file ser-

vice to the virtual file store. Protocols for the transfer of whole files, parts of files and information about files are provided. These build on the framework provided by the rest of the application layer (i.e. the Common Application Service Elements –CASE) and lower layers of OSI. Layer 6, the presentation layer, provides support for heterogeneity. [103][63][67]

A.34 FUSS

FUSS is one of the file systems supported by Amoeba. FUSS uses both optimistic concurrency control for small updates and locking for large ones. Files are immutable and have version numbers. This allows simple client caching. IO is done a block at a time. Atomic update is provided and lazy copy gives good performance on copy if few changes occur. The Unix file interface is provided. [220][146]

A.35 Gaffes

Gaffes is designed to be a global information sharing system serving more than a million workstations. GAFFES supports replication of files but because of its scale it allows users to select the number of replicas at the time a file is created and the degree of consistency when a file is opened. Files may contain references to other files which, when read, return the data of the referenced file. Version numbering is supported. Security is based on the use of public key and encryption hardware. Caches are maintained in a consistent state by callbacks from a replica server. Replicas are maintained in a consistent state using a two phase update (update then commit). Files may have triggers associated with them which cause an application to run on some event. For example a compressed file may have a trigger so that it is decompressed when a read operation occurs. [84]

A.36 Gemini

Gemini is a replicated file system test-bed built using Unix systems. The primary goal of replication is to provide fault-tolerance. Gemini was used to compare voting with witnesses, dynamic voting and a form of semi-synchronous control. Unix programs, when re-linked with the Gemini library will run unaltered. An additional file system-call, `gcreate(path, mode)` creates a new replica. Unix pathnames are extended to support

remote access to files. The system is divided into servers, which trust one another and coordinate file access and update, and clients which access a single local server. [40] ([87])

A.37 Grapevine

Grapevine is a special purpose distributed system that provides message delivery, resource location, authentication and access control. Naming information is replicated at every name server but messages are only stored at a single site (although a users messages may be distributed over several sites). Hints are used in the naming process. The security and integrity of the grapevine servers is assumed but the failure or insecurity of any user client will only affect the users of that client. About 1500 workstations were involved in Grapevine. Cedar makes use of Grapevine's naming scheme. File storage is provided by Xerox's IFS (Interim File System) file servers. [29]([46][126])

A.38 Guardian

The Guardian operating system runs on Tandem NonStop computers. Its main feature is a very high degree of fault tolerance. Guardian has, at its center, a fault tolerant message passing system. Communication between processes is by way of messages. Processes operate with at least one duplicate in a master slave arrangement. When a process (including a file system process) receives a message, it checkpoints it to its slave. File system processes maintain mirrors of the file system on different disks. At a physical level communication is by way of separate IO channels, each with its own redundant power supply. At least two processors are included in any unit. [23]

A.39 HARKYS

HARKYS gives transparent access to remote files in a Unix environment. Complete remote file systems can be mounted on the local machine. Use of remote devices through their Unix special files is supported. HARKYS is implemented as a replacement for some of the standard Unix libraries. This requires a recompilation of existing software if it is to work with HARKYS. The design is built around the RPC paradigm and uses TCP for communication. Remote requests are intercepted by HARKYS and forwarded to the appropriate machine. [20]

A.40 Harp

Harp is a Unix file system that is designed for high availability and reliability. Files are replicated and continue to be available even in the presence of some server, media failures or network partitioning. Harp uses a primary copy replication technique but achieves good performance by using write behind. Servers are equipped with small UPSs to ensure that data is written to disk from the write-behind log even when a power failure occurs. Harp is built to support the vnode (VFS) file system interface. Operations are atomic. Harp is intended to provide the low level support for NFS or Andrew. [128]

A.41 HCS

HCS is a software project supporting Heterogeneous Computer Systems. Goals include, easy integration of new system types and not masking the unique features of new systems. The key components of HCS are a naming service and RPC which operate across the disparate systems. Two types of file service are provided, a remotely accessible centralised file service and remote access to existing file systems. The latter is called the HCS File System (HFS). Both are non-transparent. To allow support for format conversion, structured information is transfered (not a byte stream). [157][172]([46])

A.42 Helix

Helix is an object oriented distributed file system. It is limited to a single LAN. The basic unit of the 'file' system is an object. Objects are collected into volumes which include a special directory object. Read and write operations are provided to all objects, but other operations (e.g. record oriented IO) may also be provided. Operations are atomic transaction based. The file system is implemented in a hierarchical manner. Object and directory management are done by the server storing the object. Capabilities are used for access control. [77]

A.43 HFS

HFS is one of the file systems supported under HCS (described above).

A.44 HighLight$^{\overline{d}fs}$

HighLight is a log structured file system (see LFS) structured to support a storage hierarchy including a large disk 'farm' and tertiary storage. The total storage capacity is expected to be in the order of 10 terabytes. A log based technique is used because archival environments are predominantly write based. Migration between secondary and tertiary storage is automatic. [113]([242][178])

A.45 Hydra$^{\overline{d}fs}$

Hydra is an early (1974) object oriented operating system kernel designed for a multiprocessor. Files are represented by objects and the file type may be inherited by other objects to be customised. [246]

A.46 IBM Zurich Research Labs File System

The IBM Research Labs in Zurich developed a distributed file system based on TRIPOS. It uses token ring protocols and IBM PCs hardware. A 7 layer communications protocol modeled around IBM's SNA is used. It supports whole file operations only. Atomic update is provided through a rename operation implemented in a single disk write operation. Careful ordering of file system operations means that hanging blocks should not occur but the disk map is rebuilt at boot time so lost block problems are resolved at that time. [106]

A.47 Ibis

Ibis is an extension to Unix, implemented at the user-level. It is the successor of Stork. Files are named globally by adding the machine name to the local file name. The naming system locates both the primary copy (used for writes) and a local copy if there is one. Ibis migrates the primary copy to the machine requesting an update (demand replication) with invalidation of replicas on update. Replicas are not guaranteed to be consistent with the primary copy at all times. When a file is migrated all directories that mention the file must be changed. [224]([126])

A.48 IFS (Xerox Interim File Server)

IFS is the file server used by Cedar. It transports whole files to and from workstations and does not support random access. Filenames include version numbers. Directory updates are atomic. Alpine was intended to replace it partly because IFS can not support databases. IFS was limited by the need to run it on the Alto workstation, which only provided a 16 bit address space and has no hardware support for virtual memory. ([81][38][200])

A.49 ITC

ITC is one of the early file systems used in the Andrew project. See the description of Andrew for further details.

A.50 Jade

Jade provides a uniform access method to files stored on heterogeneous machines on a network. Jade is implemented on top of local file systems which do not need to be modified. It is scalable in four dimensions: size, area, heterogeneity and autonomy. Files are cached as whole units and sequential write consistency is provided but not concurrent write consistency. [180]

A.51 Juniper (Xerox DFS) (xDFS)

Juniper (also called xDFS and Distributed File System) is an early distributed file system developed at Xerox. It runs as an application process and provides atomic file operations to its clients even when multiple servers are involved. Deadlock is resolved by breaking locks. It uses an approximation of optimistic concurrency control to avoid dynamic deadlock. In [38] Juniper is criticised for being slow and unreliable. These factors were a motivation for building Alpine. [209][38][142])

A.52 LFS (Log Structures File System)

LFS is a pseudonym for The Sprite Log-Structured File System. See Sprite LFS for more details.

A.53 Locus

Locus is one of the best known early distributed file systems. It is a distributed Unix system offering network transparency, automatic replication, global naming and continued operation during network partitions. Context sensitive files allow the appropriate executable file to be used when different architectures are included in the same system. A central site (for a particular group of files) is responsible for ensuring file synchronisation. This site (the CSS) is consulted when a file is opened and only allows the open if no conflict occurs. Replicas are updated when the file is closed. The existence of the CSS might limit scalability. In any case Locus is limited to 32 hosts. [174][239][240]([46][123][126][87])

A.54 LucasFS

LucusFS is a memory-mapped distributed file system which is implemented on the Lucas distributed system. Lucas provides a single level, 64-bit address space. Coherence is provided by a server (the cache server) which operates as a user level process on each system and communicates with other servers. Lucas is built using the Mach micro kernel. LucasFS allows users to customise, through a protocol description language, the coherence protocols making them more suitable for the particular application they support. The Lucas system is named after Edouard Lucas, the French mathematician who invented the "Towers of Hanoi" puzzle. [229]

A.55 MOS

MOS is a network transparent distributed operating system that supports dynamic process migration. It provides the Unix (version 7) system calls. Path names come in three variants, working directory relative, root relative and super root relative, the latter two starting with / and .../ respectively. Each tree resides in a single machine. The major internal mechanism is an (RPC like) extended procedure call mechanism. Dunix appears

to be a derivative of MOS. [22]

A.56 Mungi

Mungi is a single address-space distributed operating system based on 64-bit addresses. Addresses are globally unique and may refer to either persistent or transient objects. It is aimed at medium scale systems, i.e. those up to a few hundred machines. The address space stores objects, but the system manages them in pages which are the unit of migration and replication. Objects may be created, destroyed or truncated but not expanded. Protection is based on capabilities. [96]

A.57 ND (network disk from Sun)

ND is a simple remote access protocol for disk partitions. It is block based without locking and is not suitable for situations where the disk is shared. ND provided early support for diskless Sun workstations, allowing them to access root and paging partitions on a server.

A.58 Netware

The Netware operating system provides server support for a variety of operating systems including MS-DOS, OS/2 and Unix. To achieve this Netware supports multiple protocols including NFS and proprietary Novell protocols. In general these protocols allow disk sharing by forwarding operations on remote files to the server. Netware is non-preemptive and so meta-data updates are naturally serialised. [135]

A.59 Newcastle connection

The Newcastle Connection (also called Unix United) is a union of Unix systems. The main device of The Newcastle Connection is a super root (`/../`) which allow the standard Unix file system naming to be extended to include multiple computers. The Newcastle Connection is implemented by an additional layer of software between the Unix kernel and applications programs. This provides RPC facilities and diverts system calls to the appropriate host if they are not local. [39]([126])

A.60 Nexus

Nexus provides an object oriented programming environment modeled on the processor pool architecture. It is implemented as an extension to Unix on Sun workstations. Operations on objects are atomic and objects are made persistent by using a virtual disk object. [44]

A.61 NFS

NFS is the best known and most widely used distributed file system. It provides remote access but does not provide migration, replication or strict coherence. NFS provides a mechanism to allow file system-calls to be satisfied either locally or remotely. A key aspect of the NFS design is its use of a stateless protocol. When a read or write system-call is made on a file descriptor that is for a remote file, the call is translated to an NFS read or write call and passed to the remote system via the distribution interface. In the case of a read the data is returned in the reply and then passed back to the user program that made the system-call. If the request exceeds 8kb it is divided into 8kb blocks for transfer. Both the client and the server maintain caches. [141][133][194]([107][81][152][208][111][46][203][58][126][207][87][199])

Further details are given in Chapter 4, beginning on page 68.

A.62 Plan-9

Plan 9 is a distributed system intended as a production system for software development and general computing using heterogeneous computers. It does not provide an existing operating system interface but does draw heavily on Unix. Some ideas from Plan 9 are included in most modern Unix systems. It is designed around a hierarchal network architecture with central servers being connected by a high speed links (in the order of 10Mbps but not always by LANs –some point to point links are used). User workstations are connected by lower speed links (down to 9600bps). The file system is the key component of the Plan 9 architecture; all resources look like files. Communication is file (not block) oriented. Plan 9 provides a per-process name space which allows customisation. For example the set of command binaries available can be selected by mounting the appropriate binaries binaries (e.g. `/mips/bin`) on the standard location for directories (`/bin`). Several directories may be mounted on the same location in a union fashion. File servers provide

byte level IO operations based on file names and locations within the file. Files are stored on a WORM with a disk and memory cache. Copy-on-write is used to support backups of the file system content at a particular instant. [171][179][170]

A.63 PULSE

PULSE provides a single Unix-like file system to microcomputers linked by a LAN. PULSE is designed for homogeneous machines each of which runs the same software and can operate as a stand-alone system. PULSE supports replication of files using a primary copy strategy. Broadcast is used to locate a file replica when needed. Communication is stateless and allows IO operations to be repeated if they appear to have failed. Although the system draws many concepts from Unix it is not compatible with an existing operating system interface. [225]

A.64 QuickSilver

QuickSilver is a distributed operating system that is intended for large geographically dispersed systems. Theses systems experience high latency. QuickSilver provides both byte stream and record structured data storage objects. Each object has a value (content) plus a series of attributes, each of which have a name, type and value. QuickSilver uses lazy-open, i.e. work needed to open the connection is not done when the object is opened, but when it is actually used. Where possible work is moved from the client to the server to reduce network traffic, e.g. a copy operation can be sent from the client to the server and the server will perform it on the clients behalf. The name server is used to send required attributes of the server to a potential client. A Unix standard IO (stdio) package has been implemented for QuickSilver. [222]

A.65 RFA

RFA is the Dacnos file system. See the entry for Dacnos for more information.

A.66 RFS

Remote File Sharing (RFS) is a distributed file system for System V Unix. RFS is implemented as an additional module in the Unix kernel. Remote access to all types of files, including Unix special files, is supported. System calls that act on remote files are sent to the appropriate remote machine. Communications is based around message passing. AT&T 'streams' plus an appropriate network layer are used to transport the messages to a remote machine. Remote file systems can be mounted on the local system with a remote mount command. It uses state information at the server, to allow the server to invalidate cached copies on a write, thereby maintaining cache coherence. Invalidation is done at the file level. Cache write-through is used to inform the server of all file updates. [15][188]([208][152]).

A.67 RNFS

RNFS is an interface to Sun's NFS that supports replication. The design is intended to be applicable to network file systems other than NFS. It is built using the ISIS toolkit. It's primary goal is to be tolerant of fail-stop faults. Client requests are managed by an agent which chooses an appropriate replica to service the request. A single agent would be a critical resource for the clients that use it so agents are is replicated. Replication consistency is managed on a read-one/write-all basis. There is a single write-token for each file, avoiding write-write conflicts. Requests for the token are broadcast to ensure the current token holder hears the request. [139]([27])

A.68 Roe

Roe provides replication and automatic migration of files in a heterogeneous environment. The goals of Roe are: high availability, strict one-copy consistency, network transparency and dynamic reconfigurability. Files are migrated to balance the storage allocation across the system. Consistency is maintained using weighted voting. Locking during an update is based on whole files. Directories have a callback mechanism to avoid long term locking. [71]([213])

A.69 ROSCOE

ROSCOE is an operating system for networked microprocessors that is intended to give a high degree of transparency. Communications is based on 'links' which are a combination of a communications channel and a capability. Files are read and written a block at a time from a remote machine via a link. The distribution interface and includes open and close operations. State information is kept by the client and the server about the status of the other. When a file is opened a link is returned. Read and write messages may be sent to the link, which returns or accepts the associated data. [206]

A.70 RVD

Remote Virtual Disk (RVD) service is used as part of Athena. RVD allows a remote disk to be accessed a block at a time. No concurrency control is provided so the file system must be read only or only used by a single client. Because of its simplicity RVD performs well. Treese[226] notes that a 1MIPS CPU (a Vax 11/750) can support 75 clients with acceptable performance. [226][46]

A.71 S-Unix and F-Unix

S-Unix and F-Unix are the client and server (respectively) of a distributed Unix system. The two components operate in separate hardware and are connected using a virtual circuit based data switch. F-Unix file servers are mounted on the S-Unix file space. Access to remote files is by sending file operation messages to the F-Unix server. Local references to inodes are kept for open files. There is no local caching of remote files. [130]

A.72 Saguaro

Saguaro provides a replication facility for Berkeley Unix. It is based on a concept called meta-files. A meta-file is one that contains a list of symbolic links to replicas of a file. When a meta-file is opened, one of the replicas that is available is located and a reference to it returned. Sets of replicated files are known as reproduction sets. Consistency between the replicas that make up a reproduction set is maintained by the system, but only on a best-effort basis. Session semantics (update on close) are used. Saguaro makes use of the

Ibis transparency mechanism.

A.73 Sprite

The Sprite operating system includes a distributed file system called the Sprite Distributed File System. Sprite offers the same view of the file system to all workstations. This includes Unix like special files which can be accessed across the network. Sprite sends IO operations (like read, write, open, close etc.) to the server for execution. This ensures serialisation and meta-data consistency. Files are read block-by-block as requested by the application. Large caches can be deployed at both the client and the server. A write behind policy is used (the application continues once the block is in cache) allowing good performance at the risk of data loss in a crash. (This behaviour is similar to most centralised Unix systems.) Strict file consistency is a major contribution of Sprite. Sprite is a stateful system; the server uses information about which files are open to detect situations where two clients have the same file open for write. When this occurs any blocks for that file are flushed from the client caches and no further client caching occurs. [152][151][244][159]([70])

Further details are given in Chapter 4, beginning on page 72.

A.74 Sprite LFS

Sprite LFS is a log-structured file system for the Sprite Distributed Operating System. Log structured file systems write data to a log which occupies consecutive disk locations. This makes write operations much quicker and improves the performance of write dominated disk IO. The presence of large caches has been predicted to make IO mostly write dominated. Measurements indicate that LFS improves Sprite's performance by an order of magnitude. [189][190]([158][113])

A.75 Spritely NFS

As an experiment in file system consistency, the Sprite research group implemented sprite style consistency in an NFS system. Like Sprite this gives Spritely NFS strict coherence. In low sharing situations, it also improves the performance of NFS because less consistency traffic is required. Under heavy sharing Spritely NFS performs less well than NFS because

of the lack of local caching for shared files. [208][207]

A.76 Stork

Stork is a distributed file system intended to meet the needs of both local and wide area file systems. A single transparent file system is provided to all users. Files migrate to the machine on which they are most commonly used. To gain access to a file it must first be seized. This causes the whole file to be migrated to the machine making the request. Replication is only supported for recovery files. [166]([126]).

A.77 Swallow

Swallow is an object oriented distributed system. It is similar to WFS and Juniper. Encryption is used for access control. File servers are not responsible for protection. Swallow also supports stable storage and atomic transactions using Lampson methodology[120]. Atomic actions are built using a version history for each object. Swallow uses the client/server model, but servers are known as repositories while clients are called brokers. The protocols employed are an implementation of those suggested by Reed[182]. Swallow is probably the first use of immutable files. [183]([146][81])

A.78 Swift

Swift uses RAID[168] style striping in a distributed file system to achieve high performance. It is designed to use a high speed interconnection medium, but the prototype operated on an Ethernet. The prototype provides a Unix-like file system. Consistency is based on a call-back mechanism. File operations (including open, close, read and write) are sent to the storage site (called a storage agent) for execution. The storage agent maintains replicated meta-data for the file. Authentication and access control are based on encryption. [41]

A.79 TRIPOS

TRIPOS is the operating system run by the Cambridge Distributed System. TRIPOS is a single user operating system. It is run on a CPU from the processor pool for each user.

See also the entry for the Cambridge Distributed System. [186][106]([219])

A.80 Truffles

Truffles is an extension to Ficus that improves its security and management characteristics. The Privacy Enhanced Mail (PEM)[21] standard is used to support security. See the entry for Ficus for more details on the basic sharing mechanisms used in Ficus and Truffles. [185]

A.81 Unix United

Unix United is another name for The Newcastle Connection. See The Newcastle Connection for more information.

A.82 V

V is a distributed file system that provides a Unix-like file system. Files are mapped into virtual memory address space. Internally, V is a message passing light weight kernel. Virtual address spaces are files. Physical memory acts as a cache of pages for the file system. I.E. access to memory, is semantically the same as accessing the file the memory is bound to. Consistency is managed with an ownership protocol and a lock manager at the server. Internally, I/O uses the UIO[47] interface, which is a stateful interface. This allows it to support pipes, windows and network connections as well as files. Naming is managed by servers, e.g. each file server has a directory system. This makes lost file recovery easy. Multicast, with an application based client name cache, is used for name resolution. The entries are kept consistent with a problem specific consistency mechanism. [48][49]([219])

A.83 Vax cluster

The VAX cluster includes a distributed file system that provides DEC's `$QIO` VMS file system interface. The system uses the client/server architecture. Files are identified by a 48 bit file UID, which includes a 16 bit sequence number. Files are accessed on a block

by block basis, with consistency provided by a separate locking system, the VAX/VMS Distributed Lock Manager[205]. Locks are applied to whole files, volumes or devices. There is a single lock on the block availability table. When a file is created or extended the XQP process on that system must obtain that lock. Whole files are locked before any operation that affects the state of the file is performed. Deadlock is prevented by always taking locks in order. Computers are connected by the CI (computer interconnect) which is a proprietary 70Mbps LAN. Ethernet can also be used at a performance penalty. Shadow disks are supported for replication. [114][153][83][205][115][138]

A.84 The Version 8 Network File System

The Version 8 Network File System is a Unix (version 8) network file system built at AT&T Bell Labs. A modified mount command is used to mount remote file systems on the local machine. The Version 8 Network File System is implemented as a modification to the Unix kernel and includes the concept of a remote inode. When an operation is performed on a remote inode an RPC is made to the appropriate remote system. [243]

A.85 WFS

WFS is a shared distributed file system. Files are stored on file servers and accessed a page at a time. A lock may be applied to the file to guarantee consistency. Meta-data is maintained at the server. WFS uses structured internal names and a hash table to map internal names to disk file addresses. WFS is typical of simple shared distributed file systems. [212]([169][123])

A.86 Xerox Distributed File System (XDFS)

XDFS is an pseudonym for Juniper. See the entry for Juniper for details.

A.87 xFS

xFS is intended as a file system that can support a large tertiary memory system and a wide area network. It is based on the concept of arranging hosts in a hierarchical structure, based on geographic proximity. This allows good scalability in a LAN environment

while supporting the typical locality patterns of file access. An invalidation based, write back, cache consistency mechanism is used where clients can maintain ownership of a file indefinitely. A write-on-demand scheme is used where clients maintain modified copies of data until it is requested by the server. Storage is log structured[189]. Automatic migration of blocks between different levels of a storage hierarchy is supported. The use of checksums to prevent modification of data stored on remote machines is proposed. [242]

A.88 Zebra

Zebra uses a combination of a log structured file system (LFS) and a striped file system. Writes to the file log are striped across servers. Stripes are immutable. It is designed to provide high performance and high availability, with good scalability. Meta-data is maintained by a file manager. All name space changes (create and delete operations) as well as open and close events, are sent to the file manager, which is a critical resource. [92][91][90][89]

Further details are given in Chapter 4, beginning on page 76.

Appendix B

Minix Information

B.1 Introduction

In the introduction to "OPERATING SYSTEMS: Design and Implementation" (which is informally called "The Minix Book") Andrew Tanenbaum writes:

> "Most books on operating systems are strong on theory and weak on practice. ... To correct this imbalance I have written an new operating system from scratch. MINIX has the same system calls as Version 7 UNIX (except for the omission of a small number of unimportant ones)." [215]

Minix runs on IBM PC compatible computers and also on some 68000 based machines. It is supplied with source code for the whole system with the exception of the C compiler. Minix is available free of charge to Universities and is also sold bundled with the Minix book and separately by Prentice-Hall.

The operating system kernel[29] of version 1.5.10 (the version used as the base of the practical work in this thesis) is about 24000 lines of C code and about 3000 lines of assembly code. The file system, which was the most heavily modified part of Minix in this project, contains about 7500 lines of C code.

Internally Minix is structured as 3 main components:

- the kernel which provides process management and low level IO (device drivers)

[29]The term kernel is used here to refer to the parts of the operating system that implements the system-calls the operating system provides. That is the process management, IO, file system and memory management code. It does not include utility programs or user applications. Although this is a widely accepted definition it is not how it is used in Minix which uses a micro-kernel approach.

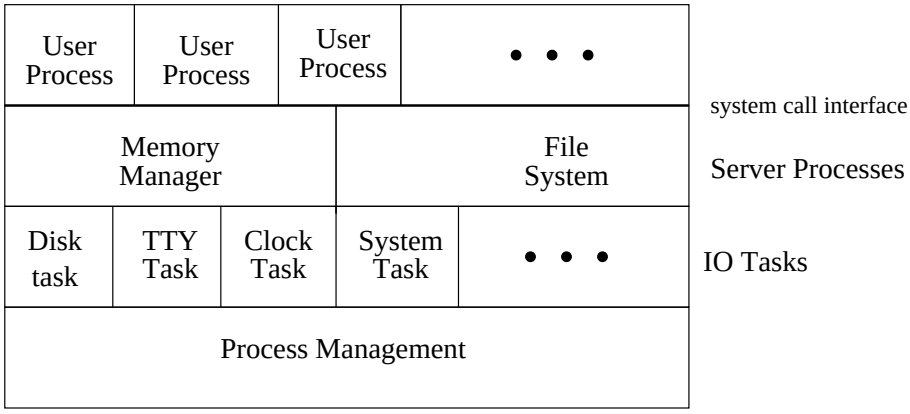


Figure B.1: The structure of Minix
(adapted from Fig 2-26 in [215])

- the memory manager, and
- The file system

B.2 File System Calls

As mentioned above Minix implements the Unix version 7 system-call interface. BB-NFS maintains this interface. The file system calls are listed below:

exit Terminate the current process.
fork Create a new process.
read Read data from a file.
write Write data to a file.
open Open a file.
close Close a file.
creat Create a new, empty file.
link Create a new name for an existing file.
unlink Remove a name of a file. If the file has no more names remove the file.
chdir Change the working directory.
time Get the time of day.
mknod Create a 'special file'.
chmod Change the file's access permissions.
chown Change the owner of the file.
stat Get file information, including owner and size.
lseek Move to a new position in a file.

Figure B.2: Minix Disk Layout
(adapted from Fig 5-30 in [215])

mount Make a disk partition part of the file system.
umount Remove a disk partition from the file system.
setuid Set the current user identifier.
fstat Get file information for an open file.
access Determine if a file is accessible.
sync Send modified blocks in cache to disk.
rename Change the name of a file.
mkdir Create a new directory.
rmdir Remove an empty directory.
dup Duplicate the descriptor for an open file.
pipe Create an interprocess pipe.
setgid Set the group identifier.
ioctl Set the flags of a device (e.g. speed of a communications port).
fcntl Set the modes of an open file.
umask Set the default permissions.
chroot Change the root of the file system as seen by this process.

B.3 Disk Layout

A Minix disk partition is divided into six parts. These are shown in figure B.2. The boot block is always the first block on the disk and is loaded and executed by the PC hardware when the system is started if this partition is selected at the boot partition. The boot partition is followed by the super block which contains information about the rest of the layout of the disk. In particular it stored the number of inodes and the number of data zones (blocks).

The Super block is followed by the two bit maps which indicate which inodes and zones are in use. The bitmaps may be more than a single block each depending on the number of entries they contain.

The bit maps are followed by the inode blocks. Each inode block stores 32 inodes. The rest of the partition is filled with data block.

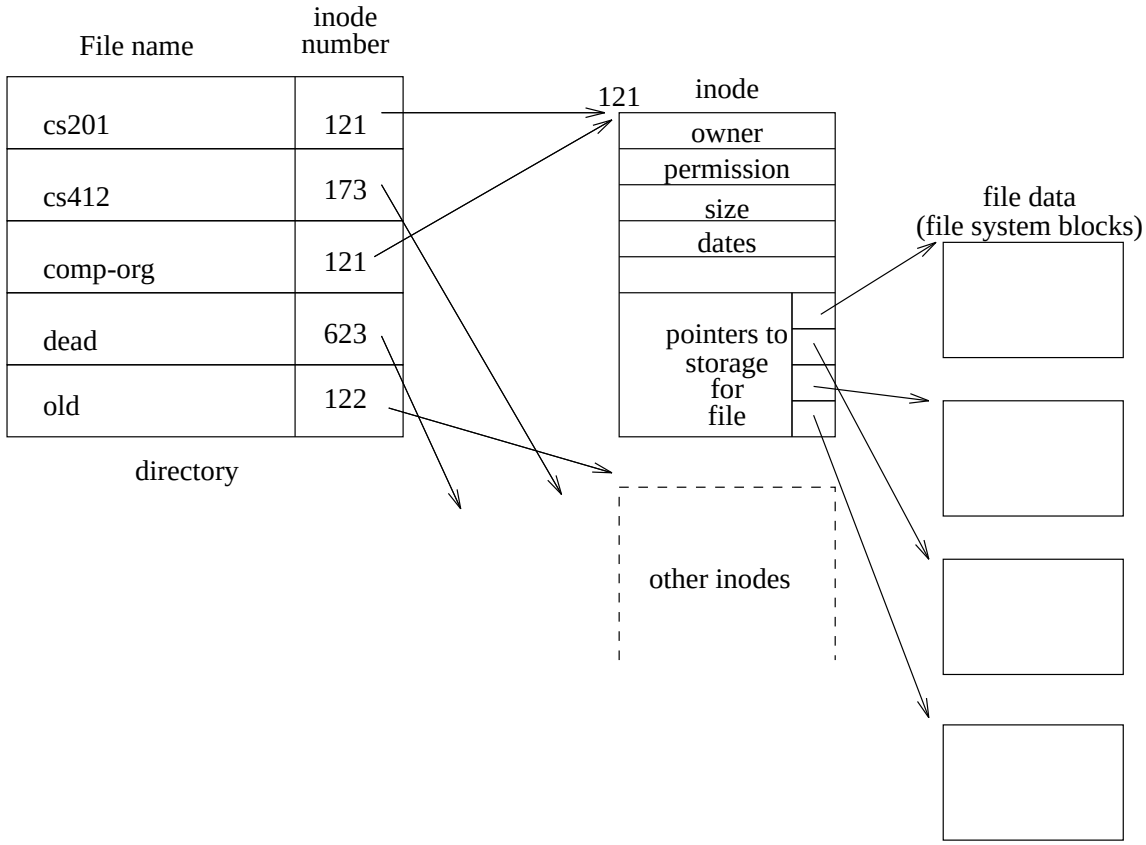


Figure B.3: Minix Directory Structure

B.4 Directories

A Minix directory is a file which contains a list of files and sub-directories. Each entry in the list contains two components, a name and an inode number. The name is the name of a file or another directory and the inode number is an index into the inode table. Inodes (described in section B.5 of this appendix) contain information about files.

Because a directory is structured in the same way as a file it also has an inode (which is listed in the directory that is the parent to the current one). A bit in the inode is set to indicate that the inode is for a directory not a normal file.

B.5 Inodes

Inodes store most of the information about a file. The exceptions are the name (or names) of the file, the file data and for a large file some of the pointers to the data in the file. The role of an inode is shown in figure B.3. Each inode has the structure shown in figure B.4.

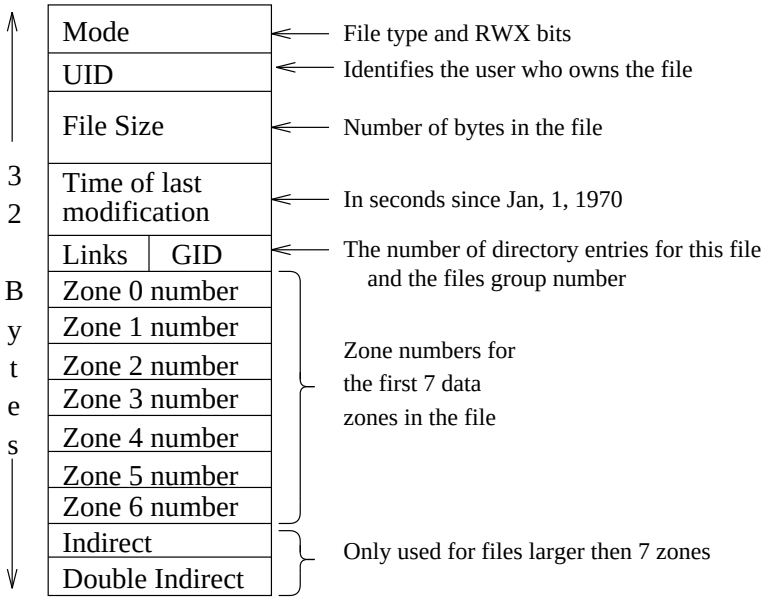


Figure B.4: Minix Inode structure
(adapted from Fig 5-32 in [215])

B.6 Bit Maps

There are two bitmaps used in a Minix file system. The first contains a bit for each inode in the inode table at the start of the disk. The second contains a bit for each data block on the disk.

In both cases the appropriate bit in the bit map is set to indicate that an item is in use. When a new block is to be allocated the block bit map is searched for a 0 bit. The search will start from the bits for a block already allocated to the file to which the new block will be added so that, where possible, the blocks of a file are stored close together. When a 0 bit is found it is set to 1 and the block number associated with the bit is returned to the routine requesting the new block.

B.7 Block Cache

The Minix block cache is used to reduce the number of accesses to the disk. It is also used as a place for a block that is in use to be stored. For example if data is being added to a file the last block in the file will be stored in the cache while the data is written into it.

The blocks in the cache are linked into a double-linked list with the block that has not

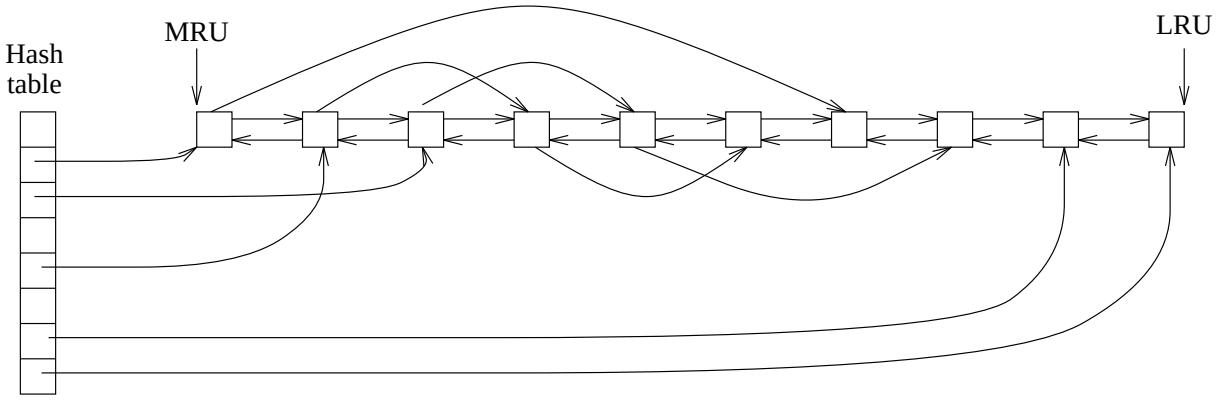


Figure B.5: Minix Cache Structure
(adapted from Fig 5-33 in [215])

been used the longest at one end of the list and the most recently used block at the other. When a block must be removed from the cache the block that has not been used for the longest will be chosen for replacement.

When the file system requires a block the cache is searched to see if the block resides there. If it does not the block is fetched from the disk and stored in the cache. In either case a pointer to the cache entry that contains the block is returned to the calling routine.

To allow the cache to be quickly searched for a block a hash table and hash-chain is maintained. The structure of the cache is shown in figure B.5.

Appendix C

Protocol Headers

C.1 IP

The IP protocol header fields (see figure C.1) are described below. The way the field is used in BB-NFS is also described.

Version The version number of this implementation of the IP protocol.

The current version number is 4.

IHL

The length of the IP Header.

The DFS protocol does not use any options so the header length is always 5.

Type of Service

This field contains a set of options that influence the priority and optimisation of the datagram traffic.

DFS does not require its datagrams to be allocated different priorities. All datagram are sent as routine priority, the probability of repeated datagrams to the same destination is selected along with a preference for speed over reliability. As a consequence of these preferences the Type of Service field is set to 1.

Total Length

The length of the IP datagram.

All DFS PDUs are either 30 bytes or 1052 bytes long, depending on whether they carry a data block (write requests and read replies) or whether they only carry the header (read requests and write replies).

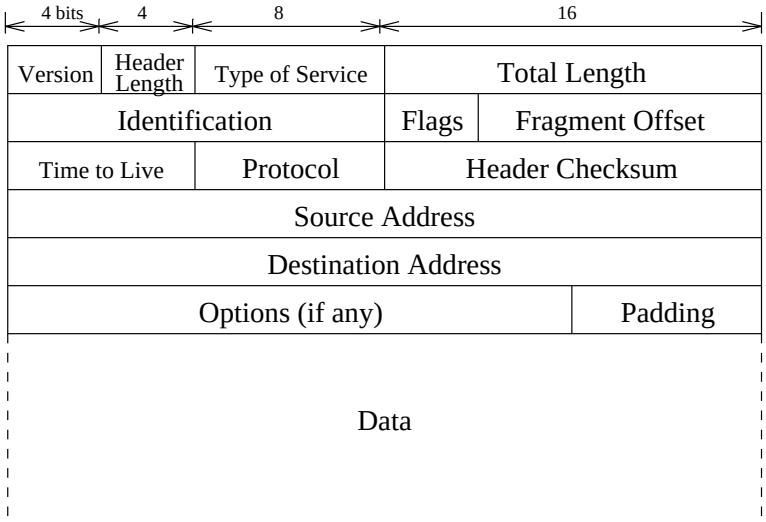


Figure C.1: The IP Header

Identification

A number that identifies a fragment with a particular IP datagram.

Because all DFS PDUs are small, and most networks can carry at least 1500 bytes, fragmentation is not required or allowed. As a consequence this field is always 1.

The interworking of DFS within UDP and IP, without fragmentation has been tested on a national and international basis, including traffic through the USA, Australia and New Zealand.

Flags

Bit 1 is reserved and must be 0. Bit 2 specifies whether fragmentation is allowed. Bit 3 is set to indicate the this is the last fragment.

Because fragmentation is not required all bits are 0.

Fragment Offset

Which fragment of the original datagram is carried by this packet.

Because fragmentation is not required this field is always 0.

Time to Live

A hop count, used to limit the number of nodes that an IP datagram can pass through. This field is a safeguard against datagrams looping forever in malformed networks.

For testing a value of 60 has been used. In the production system this value will be increased.

Protocol

The higher layer protocol that this datagram is carrying. Normally either TCP or UDP.

This field is 17, specifying the UDP protocol.

Header Checksum

Because all fields of the header, except total length, are constant, the checksum for the two different length datagram can be calculated at the time communication with the destination is initiated. Two IP headers that vary only in the length and checksum fields are then created and stored in memory with space for the UDP header and DFS PDU to immediately follow the header.

Source address

The IP address of the sending node.

Destination Address

The IP address the datagram is intended for.

Options

A variable length field including debugging, performance and security options.

No options are used.

C.2 UDP

The UDP protocol header contains the following fields (see figure C.2).

Source Port

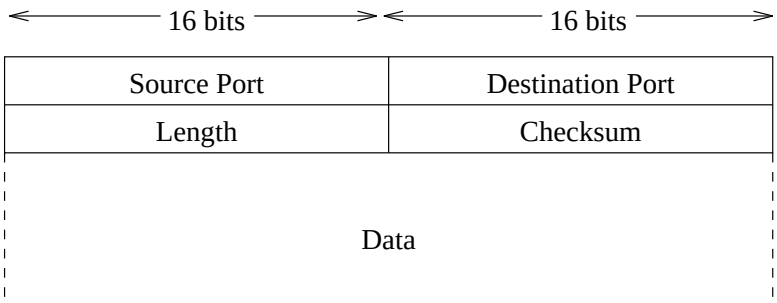


Figure C.2: The UDP Header

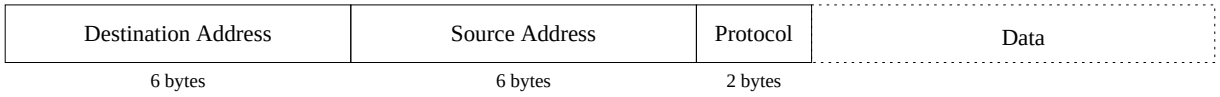


Figure C.3: The Ethernet Headers

This is the UDP port number that the communication originates from.

The port number chosen for the DFS protocol is 1200.

Destination Port

This is the UDP port number that the communication is directed to.

The port number chosen for the DFS protocol is 1200.

Length

The length of the UDP datagram, including the header.

For DFS this is either 1062, if a disk block is being carried by the datagram or 38 if no disk block is being carried.

Checksum

The checksum field is optional. If it has the value 0 no checksum is computed at the receiving machine.

DFS uses the checksum facility of the underlying network and consequently the UDP checksum is 0.

C.3 Ethernet

Figure C.3 shows the format of the Ethernet frame header. The frame has the following fields:

Source Address

The 6 byte Ethernet address of the sending device. Each network interface card is allocated a unique address.

Destination Address

The 6 byte Ethernet address of the receiving device.

For a particular association between a client and a server the source and destination addresses are fixed and can be laid down in memory when the association begins.

Protocol

The identity of the higher layer protocol being carried by this frame. The presence of this field allows a single Ethernet to carry mixed traffic for a number of higher level protocol (e.g. the IP protocol from TCP/IP and Novels IPX protocol).

BB-NFS uses the Ethernet version of the frame. Because its protocol is encapsulated in IP this field always `0x0800`.

Appendix D

Modifications to Minix

This appendix lists the areas of Minix that were modified as part of the project. Brief notes on the nature of the changes are included. Further details of the modification can be found at:
`http://www.cs.waikato.ac.nz/~tonym/bb-nfs/mods/mods.html`

- **Instrumentation.** Add instrumentation (including new system call to generate statistics).
- **Block maps.** Lock block map blocks (i.e. double and triple indirect blocks) before use to ensure coherence.
- **Block Status.** Blocks can be locked stale or have a short lease in addition to being clean and dirty. Change all assignments to block status (clean, dirty, stale etc.) to bit assignments and all tests to bit tests.
- **Stale Blocks.** Blocks go stale after sitting in the cache too long. This limits how long non-coherent data can sit in the cache.
- **Lease Revoke.** When a lease revoke arrives, return lease immediately if not in use or as soon as finished.
- **Lock Cache.** Allow locked blocks to sit in cache, return lock and block from sync or when a revoke arrives.
- **File System Transactions.** Keep a copy of a changed block in the cache (or optionally in a separate block store –provided for taking measurements). Also keep dirty field and number of users (counts) in case transaction aborts. Dirty field is used to determine if block was locked before this file system transaction. Also keep

Name	Use
reads in	The number of read requests entering the cache.
no read	Blocks requested from the cache with no prior content.
lock writes	Number of unlocking writes leaving the cache.
lock read	Blocks returned to the cache that are locked.
read hit	Block read requests that are satisfied from the cache.
read out	Read requests that are not satisfied by the cache.
write in	The Number of write requests entering the cache.
lock write	Unlocking writes when the block is still in use. This is an error condition.
lock reuse	Lease requests satisfied by the lease cache.
write through	Blocks that are written through the cache. These are blocks that are crucial to the correct operation of the file system. Because Unstable Atomic Update is safer than write through, write through does not occur when it is in use.
flushed	Blocks flushed from the cache by the 30 second cache flush procedure included in Minix and typical of Unix systems.
pre-fetch	The number of block pre-fetched.
stale refresh	Data blocks that required refreshing, because they had been in the cache beyond the valid time for file data (see section 6.6.)
dup	Blocks duplicated as part of the transaction mechanism.
unlocked revoke	Block revoke requests received.

Table D.1: Instrumentation Counters

inode table, fp table, flip table and original message as they were at the start of the transaction. Also remember `susp_count` and `reviving`. A block in the cache that is not currently in use but has been in use this transaction must not be reused.

When a file system aborts, abort any locks obtained this transaction (requires a special transaction flag to server), recover any blocks that have been modified from their copies, recalculate the bufs_in_use counter, restore the inode, file pointer and flip tables from their copies and the original message from its copy. Restore susp_count and reviving from their copies. Record which blocks are dirty at the start of the (repeated) transaction. Inodes that are changed during the transaction have their content replaced from the copy in the disk block (that will probably be in the cache) when they are fetched again, this seems unnecessary because we restore the inode anyway.

- **Unstable Atomic Update.**

  End transaction flag sent by sync. Quiet flag used to know when not to end a transaction at a sync. If not quiet and sync has no blocks to write, a special IO transaction is sent to mark the end of the file system transaction. Quiet is reset when a read or a write happens and set when a sync happens. Also used to determine whether the file system should be flushed on a sync.

  Flushall modified so that when it is called from within the file system, it does not write out blocks that have changed this transaction. This avoids problems when the transaction is aborted.

- **Coherence.** Locked read always gets data from the server (unless already locked). Add NO_REUSE as a flag to put_block used to stop block being cached. Added an extra parameter to the new_block call, to specify whether the new block should be locked. Invalidate a block when a zone is freed. This makes sure it is not written back to the disk even if it is modified. It is also removed from the cache.

- **Unlockall system call.** A system call that returns all leases (added for testing the effect of the lock cache).

- **Piggy-back lease returns.**

  Add code to delay detect lease return on clean block. Delay lease return until a message is sent to the server.

- **Short Term Leases.** The short term lease flag is set in the original message, or when a revoke arrives for a lease that has been needed this transaction. A flag records that one or more short term leases are held and they are returned at the end of the transaction.

- **Communications.** Add ability to send an Ethernet message. Add remote device driver which detects that device is remote from minor device number, constructs the Ethernet message including message ID, adds any piggy-back locks waiting to be sent, and sends message to the Ethernet task. Wait for reply from Ethernet task. Ethernet task will send and attempt to resend after a timeout if no reply. If there is no reply after repeated attempts, return to the remote device driver. and the transaction aborts. If a reply is received, check to see if it is a revoke rather than the reply to the original message. If it is a revoke, then return lease if free and wait for the original message reply.

- **Log Messages.**

  All kernel messages are logged from putc. Minor device 20 installed as a logging device. Any writes to this device are logged.

- **Inode coherence.** Lock and unlock inodes. Lock block that contains the inode. Must check that another inode is not already locked in the block, if so, don't re-lock.

  All inodes are locked when fetched and unlocked at the end of the transaction. When a block is fetched any unlocked in core inodes are updated because they may have out of date changes (another client may have locked the inode, changed it and then unlocked it again).

- **Mtime.** Time management in inode, when an inode is read use the newer of the times in the inode on disk and the time in the in-core inode (if it was already in core).

- **Remote Delete.** Add one to number of links when an inode is opened. This ensures it is not deleted on one client when it is still in use on another (required Unix semantics). This introduces a problem when client crashes with open files. FSCK fixes this.

- **Errors Corrected and Minor Changes.** Corrected error in bracketing in put_inode call in.

  rw_block made private to cache.c (not used elsewhere)

  Remove unnecessary get_inode, put_inode pair when deleting a directory entry.

  Reduced size of inode and flip structures to improve performance in transaction mechanism.

- **Bitmaps.** A copy of the bitmaps is kept in memory. When a new block has to be allocated, this bitmap is searched for a block that was free last time the bit map was correct. Once one is found, the block of the bit map is locked, making it coherent.

The bit is then checked again and if it still indicates a block is free, the block is allocated.

Before a bit is set to free, the bitmap block containing the bit is locked. This ensures that the memory copy is coherent and protects the change from the concurrent update.

- **Boot Code.** The boot menu program was modified, to use bootp and arp to discover the IP address of the client and the Ethernet address for the server. Options were also added to select the IP address of the server and set Ethernet card parameters.

- **Server.**

  The Unix server code is not included in the above list because it was not implemented as a modification to Minix.

Bibliography

[1] V. Abrossimov and M. Rozier. Generic virtual memory management for operating system kernels. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 123–136, Litchfield Park, Arizona, USA, December 1989.

[2] V. Abrossimov, M. Rozier, and M. Gien. Virtual memory management in Chorus. In *Progress in Distributed Operating Systems and Distributed Systems Management*, Berlin, April 1989. Springer Verlag.

[3] M. Accetta, G. Robertson, M. Satyanarayanan, and M. Thompson. The design of a network based central file system. Technical Report CMU-CS-80-134, Computer Science Department, Carnegie-mellon University, August 1980.

[4] G. T. Almers, A. P. Black, E. Lazowska, and J. D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, 11(1):43–59, January 1985.

[5] R. Ananthanarayanan, S. Menon, A. Mohindra, and U. Ramachandran. Experiences in integrating distributed shared memory with virtual memory management. *ACM SIGOPS Operating Systems Review*, 26(3):4–26, July 1992.

[6] D. P. Anderson and D. Ferrari. The DASH project: An overview. Technical Report UCB-CSD-88-405, Computer Science Division, University of California, Berkeley, Berkeley, California, 94720, February 1988.

[7] D. P. Anderson and S-Y Tsou. The DASH local kernel structure. Technical Report UCB-CSD-88-463, Computer Science Division, University of California, Berkeley, Berkeley, California, 94720, November 1988.

[8] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[9] F. Armand, M. Gien, F. Herrmann, and M Rozier. Revolution 89 or distributing Unix brings it back to its original virtues. Technical Report CS/TR-89-36.1, Chorus

Systèmes, Ft. Lauderdale, FL, USA, October 1989. *Also in* Proceedings of the Workshop on Experiences with Building Distributed (and Multiprocessor) Systems.

[10] Y. Artsy and R. Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, 22(9):47–56, September 1989.

[11] W. Aspray. *John von Neumann and the origins of modern computing.* MIT Press, 1990.

[12] E. H. Baalbergen. Parallel and distributed compilations in loosely-coupled systems: A case study. In *Proceedings of the Workshop on Large Grained Parallelism*, 1986.

[13] E. H. Baalbergen. Design and implementation of parallel make. *Computing Systems*, 1(2):135–158, 1988.

[14] H. E. Baalbergen, K. Verstoep, and A. S. Tanenbaum. On the design of the Amoeba configuration manager. *ACM SIGSOFT Software Engineering Notes*, 14(7):15–22, November 1989.

[15] M. J. Bach, M. W. Luppi, A. S. Melamed, and K. Yueh. A remote-file cache for RFS. In *Proceedings of the USENIX 1987 Summer Conference*, pages 273–279, Phoenix, Arizona, USA, June 1987. USENIX.

[16] M. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and Ousterhout J.K. Measurements of a distributed file system. In *Proceedings of the 13^{th} ACM Symposium on Operating System Principles*, Pacific Grove, California, USA, October 1991. ACM.

[17] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. A distributed implementation of the shared data-object model. In *Proceedings of the First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 1–19, Ft. Lauderdale FL (USA), October 1989. USENIX.

[18] H. E. Bal and A. S. Tanenbaum. Distributed programming with shared data. In *IEEE Conference on Computer Languages*, pages 82–91. IEEE, 1988.

[19] H. E. Bal, R. van Renesse, and A. S. Tanenbaum. Implementing distributed algorithms using remote procedure call. In *Proceedings of the 1987 National Computer Conference*, pages 499–505. American Federation of Information Processing Societies, 1987.

[20] A. Baldi and L. Stefanelli. HARKYS – A new network file system approach. In *Proceedings of the EUUG Autumn 1986 Conference*, pages 163–173, Manchester, UK, September 1986. European Unix Users Group.

[21] D. Balenson. Privacy enhancement for Internet electronic mail: Parts i-IV. Technical Report RFCs 1421-1424, Internet Standards, February 1993. *Also published as DEC Technical reports.*

[22] A. Barak and A. Litman. MOS: A multicomputer distributed operating system. *Software Practice and Experience*, 15(8):725–737, August 1985.

[23] J. F. Bartlett. A NonStop kernel. *ACM SIGOPS Operating Systems Review*, 15(5):22–29, December 1981.

[24] J. M. Bernabêu, P. W. Hutto, M. Y.A. Khalidi, R. J. Ahamad, W. F. Appelbe, P. Dasgupta, R. J. LeBlanc, and U. Ramachandran. The architecture of Ra: A kernel for Clouds. Technical Report GIT-ICS-88/25, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, USA, 1988.

[25] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[26] K. Birman and T. A. Joseph. Exploiting replication. Technical Report TR 88-917, Department of Computer Science, Cornell University, Ithaca, NY, USA, June 1988.

[27] K. Birman and K. Marzullo. A brief overview of the ISIS distributed programming toolkit and the Meta distributed operating system. Technical report, Cornell University, NY, USA, 1989.

[28] K. Birman and K. Marzullo. The ISIS distributed programming toolkit and the Mesa distributed operating system. *Sun Technology*, pages 90–104, summer 1989.

[29] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(1):260–273, April 1982.

[30] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[31] P. Biswas and K. K. Ramakrishnan. File access characterization of VAX/VMS environments. In *The 10^{th} International Conference on Distributed Computing Systems*, pages 227–234, Paris, FRANCE, May 1990. IEEE.

[32] P. Biswas, K. K. Ramakrishnan, D. Towsley, and C.M. Krishna. Performance benefits of non-volatile caches in distributed file systems. Technical Report UM-CS-1994-019, Computer Science Department, University of Massachusetts, Amherst, MA, 1994.

[33] A. Black, N. Hutchinson, E. Jul, and H. Levy. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):78–86, January 1987.

[34] G. S. Blair, J. R. Mariana, J. R. Nicol, and D. Shepherd. A knowledge-based operating system. *The Computer Journal*, 30(3):193–200, June 1987.

[35] D. G. Bobrow, J. D. Burchfield, D. L. Murphy, R. S. Tomlinson, B. Beranek, and Newman Inc. TENEX, a paged time sharing system for the pdp-10. *Communications of the ACM*, 15(3):135–143, March 1972.

[36] U. M. Borghoff and K. Nast-Kolb. Distributed systems: A comprehensive survey. Technical Report TUM-I8909, Techn. Univ. Müchen, Munich, Germany, November 1989.

[37] M. R. Brown, R. G.G. Cattell, and N. Suzuki. The Cedar DBMS: A preliminary report. In *Proceedings of the ACM International Conference on Management of Data*, pages 205–211. ACM, April 1981.

[38] M. R. Brown, K. N. Kolling, and E. A. Taft. The Alpine file system. *ACM Transactions on Computer Systems*, 3(4):261–293, November 1985.

[39] D. Brownbridge, L. Marshall, and B Randell. The Newcastle Connection or UNIXes of the world unite! *Software Practice and Experience*, 12(12):1147–1162, December 1982.

[40] W.A. Burkhard, B.E. Martin, and J-F Pâris. The Gemini replicated file system test-bed. In *Proceedings of the Third Internation Conference on Data Engineering*, pages 441–488. IEEE, February 1987.

[41] L-F. Cabrera and D. D.E. Long. Swift: Using distributed disk striping to produce high I/O data rates. *Computing Systems*, 4(4):405–436, 1991.

[42] R. Campbell, N. Islam, and P. Madany. Choices, frameworks and refinement. *Computing Systems*, 5(3):217–257, 1992.

[43] R. Campbell, N. Islam, D. Raila, and P. Madany. Designing and implementing Choices: An object-orientated system in C++. *Communications of the ACM*, 36(9):117–126, September 1993.

[44] J. L. Carrol, D. E. Darrell, and D. E. Long. Block-level consistency of replicated files. In *The 7^{th} International Conference on Distributed Computing Systems*, pages 146–153. IEEE, 1987.

[45] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[46] G. A. Champine and D. E. Geer. Project Athena as a distributed computer system. *IEEE Computer*, pages 40–50, September 1990.

[47] D. R. Cheriton. UIO: A uniform I/O interface for distributed systems. *ACM Transactions on Computer Systems*, 5(1):12–46, February 1987.

[48] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[49] D. R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9^{th} Symposium on Operating Systems Principles*, pages 129–140, Bretton Woods, N. H. USA, October 1983. ACM.

[50] S. Y. Cheung, M. Ahamad, and M. H. Ammar. Multi-dimensional voting: A general method for implementing synchronization in distributed systems. In *The 10^{th} International Conference on Distributed Computing Systems*, pages 362–369, Paris, FRANCE, May 1990. IEEE.

[51] Chorus Systèmes. Technical overview of the CHORUS system: The integration of real-time, distributed processing and Unix. Technical Report CS/TR-89-10.2.0, Chorus Systèmes, Marc Shapiro, INRIA, 1989.

[52] S. Chutani, O. T. Anderson, L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. In *Proceedings of the Winter 1992 USENIX Conference*, pages 43–60. USENIX, January 1992.

[53] C.T. Cole, P.B. Flinn, and A.B. Atlass. An implementation of an extended file system for Unix. In *Proceedings of the Summer 1985 Usenix Conference*, pages 131–149, Portland, June 1985. Usenix.

[54] D. Comer. *Internetworking with TCP/IP -Volume I.* Prentice Hall, 1995.

[55] D. Comer. *Internetworking with TCP/IP -Volume III.* Prentice Hall, 1995.

[56] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, 1994.

[57] B. Croft and J. Gilmore. Bootstrap protocol (BOOTP). Technical Report RFC951, Internet Standard, September 1985.

[58] M. D. Dahlin, C. J. Mather, T. E. Wang, R. Y. Anderson, and D. A. Patterson. A quantitative analysis of cache policies for scalable network file systems. Technical Report UCB//CSD-94-798, University of California, Berkeley, Berkeley, California 94720, February 1993.

[59] M. D. Dahlin, R. Y. Wang, T. E. Anterson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, 1994.

[60] P. Dasgupta, R. C. Chen, S. Menton, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J.M. Bernabêu-Aubân, P. W. Hutto, M. Y.A. Khalidi, and C. J. Wilkenloh. The design and implementation of the Clouds distributed operating system. *USENIX Computing Systems Journal*, 3(1), 1990.

[61] P. Dasgupta and R. J. LeBlanc. The Clouds distributed operating system. *IEEE Computer*, November 1992.

[62] E. E. (Jr) Davit. Computing from the communication point of view. *Advances in Computers*, 10:109–128, 1970.

[63] J. D. Day and H. Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334–1345, December 1983.

[64] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. Logical disk: A simple new approach to improving file system performance. Technical Report MIT/LCS/TR-566, MIT Laboratory for Computer Science, April 1993.

[65] C. N. R. Dellar. A file server for a network of low cost personal microcomputers. *Software Practice and Experience*, 12:1051–1068, 1982.

[66] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, August 1981.

[67] G. Dickson and A. Lloyd. *Open Systems Interconnection: Computer Communications Standards and GOSIP Explained.* Prentice Hall, 1992.

[68] E. W. Dijkstra. Co-operating sequential processes. In Genuys F., editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

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

[70] F. Douglis, M. F. Kaashoek, J. K. Ousterhout, and A. S. Tanenbaum. A comparison of two distributed systems: Amoeba and Sprite. Technical Report CS91, University of California Santa Cruz, Santa Cruz, September 1991.

[71] C.S. Ellis and R.A. Floyd. The Roe file system. In *Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems*, pages 17–19, Clearwater Beach, Florida, USA, October 1983. IEEE.

[72] Y. Endo and C. Small. The case for in-kernel tracing. Technical Report TR-34-94, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, December 1994.

[73] K. P. Eswaran, J. N. Gray, R. A. Lorie, and Traiger I.L. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[74] R. Finkel, M. Solomon, D. Dewitt, and L. Landweber. The Charlotte distributed operating system: Part IV of the first report on the Crystal project. Technical Report 502, Computer Sciences, University of Wisconsin, Madison, USA, July 1983.

[75] J. C. French, T. W. Pratt, and M. Das. Performance measurement of the concurrent file system of the Intel iPSC/2 hypercube. *Journal of Parallel and Distributed Computing*, 17(1-2):115–121, Jan and Feb 1993.

[76] M. Fridrich and W. Older. The Felix file server. *ACM SIGOPS Operating Systems Review*, 15(5):37–44, December 1981.

[77] M. Fridrich and W. Older. HELIX: The architecture of a distributed file system. In *The 4^{th} International Conference on Distributed Computing Systems*, pages 422–431. IEEE, 1984.

[78] N. H. Garnett and R. M. Needham. An asynchronous garbage collector for the Cambridge File Server. *ACM SIGOPS Operating Systems Review*, 14(4):36–40, October 1980.

[79] K. Geihs and U. Hollberg. Retrospective on DACNOS. *Communications of the ACM*, 33(4):439–448, April 1990.

[80] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7^{th} ACM Symposium on Operating System Principles*, pages 150–162, Asilomar, California, USA, 1979. ACM.

[81] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, March 1988.

[82] D. S. Gill, S. Zhou, and H. S. Sandhu. A case study of file system workload in a large-scale distributed file system. Technical Report CSRI-296, Computer Systems Research Institute, Toronto University, Toronto, ON, M5S 1A4, March 1994. *Also in* SIGMETRICS May 1994 (extended abstract).

[83] A. C. Goldstein. The design and implementation of a distributed file system. *DIGITAL Technical Journal*, 5:45–55, September 1987.

[84] A. Gozani, M. Gray, S. Keshav, V. Madisetti, E. Munson, M. Rosenblum, S. Schoettler, M. Sullivan, and D. Terry. GAFFES: The design of a globally distributed file system. Technical Report UCB/CSD/87/361, University of California, Berkeley, June 1987.

[85] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, December 1989. *Also in* Proceedings of the 12^{th} ACM Symposium on Operating System Principles.

[86] M. Guillemont. Real time in a distributed computing environment. Technical Report CS/TR-90-65.0.1, Chorus systems, Marc Shapiro, INRIA, December 1990.

[87] R. G. Guy. Ficus: A very large scale reliable distributed file system. Technical Report CSD-910018, Computer Science, University of California, Los Angeles, June 1991.

[88] F. Halsall. *Data communications, computer networks, and Open Systems*. Addison-Wesley, 1992.

[89] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. Technical Report UCB/CSD-93-744, CSD, University of California, Berkeley, Berkeley, California 94720, USA, 1993.

[90] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM SIGOPS Operating Systems Review*, 27(5):29–43, December 1993.

[91] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *IEEE Transactions on Computers Systems*, 13(3):274–310, August 1995.

[92] J. H. Hartman and J.K Ousterhout. Zebra: A striped network file system. In *USENIX Workshop on File Systems*. USENIX, May 1992.

[93] J. S. Heidemann. Personal communication. February 1996.

[94] J. S. Heidemann, T. W. Page, R. G. Guy, and G. J. Popek. Primarily disconnected operation: Experiences with Ficus. In *Proceedings of the Second Workshop on Management of Replicated Data*. IEEE, November 1992.

[95] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[96] G. Heiser, K. Elphinstone, S. Russell, and J. Vochteloo. Mungi: A distributed single address-space operating system. Technical Report SCS&E-9314, School of Computer Science and Engineering, The University of New South Wales, November 1993.

[97] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart. Availability and consistency tradeoffs in the Echo distributed file system. In *Proceedings of the 2^{nd} Workshop on Workstation Operating Systems*, pages 49–53, Pacific Grove, CA, USA, September 1989. IEEE.

[98] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[99] R. C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–195, September 1972.

[100] J. H. Howard, M. L. Kazar, S. G. Menees, D.A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[101] C. Huang and V. O.K. Li. Missing-partition dynamic voting scheme for replicated database systems. In *The 9^{th} International Conference on Distributed Computing Systems*, pages 579–586, Newport Beach, CA USA, June 1989. IEEE.

[102] R. A. Hyman. *Charles Babbage: Pioneer of the Computer*. Oxford University Press, 1982.

[103] ISO. File transfer, access and management (FTAM). Technical Report ISO8571, ISO, 1988-93.

[104] ITU. Information technology - open systems interconnection - presentation protocol definition. Technical Report X.226, International Telecommunication Union, Place des Nations, 1211 Geneva 20, Switzerland, April 1995.

[105] ITU. Information technology - open systems interconnection - presentation service definition. Technical Report X.216, International Telecommunication Union, Place des Nations, 1211 Geneva 20, Switzerland, April 1995.

[106] P. Janson, L. Svobodova, and E. Maehle. Filing and printing services on a local area network. In *Proceedings of the* 8^{th} *Data Communications Symposium*, pages 211–219. IEEE, 1983.

[107] C. Juszczak. Improving the performance and correctness of an NFS server. In *Proceedings of the 1989 Winter USENIX Conference*, pages 53–63, San Diego, February 1989. USENIX.

[108] M. F. Kaashoek, A. S. Tanenbaum, S. Flynn Hummel, and H. E. Bal. An efficient reliable broadcast protocol. *ACM SIGOPS Operating Systems Review*, 23:5–19, October 1989.

[109] M. F. Kaashoek, R. van Renesse, H. van Staveren, and A. Tanenbaum. FLIP: An internetwork protocol for supporting distributed systems. *ACM Transactions on Computer Systems*, 11(1):75–106, 1993.

[110] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S-T. Tu, and E. R. Zayas. DEcorum file system architectural overview. In *Proceedings of the Summer 1990 Usenix Conference*, pages 151–164, Anaheim, California, USA, June 1990. USENIX.

[111] C. A. Kent. Cache coherence in distributed systems. Technical Report 87/4, Digital Western Research Laboratory, Palo Alto, California, USA, December 1987.

[112] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[113] J. Kohl and C. Staelin. HighLight: Using a log-structured file system for tertiary storage management. In *USENIX Winter 1993 Conference Proceedings*, pages 435–447, San Diego, CA, USA, January 1993. USENIX. *Also in* UCB technical report S2K-92-16.

[114] N. P. Kronenberg, H. Levy, and W. Strecker. VAXclusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[115] N. P. Kronenberg, H. M. Levy, W. D. Strecker, and R. J. Merewood. The VAXcluster concept: An overview of a distributed system. *DIGITAL Technical Journal*, 5:7–21, September 1987.

[116] P. Kumar and M Satyanarayanan. Log-based directory resolution in the Coda file system. Technical Report CMU-CS-91-164, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, 1991.

[117] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transaction on Database Systems*, 6(2):213–226, June 1981.

[118] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[119] L. Lamport, R. Shostak, and M. Paese. The Byzantine Generals problem. *ACM Transactions in Programming Languages and Systems*, 4(3):382–401, 1982.

[120] B. W. Lampson. Atomic transactions. *Lecture Notes in Computer Science. Distributed Systems – Architecture and Implementation*, 105:246–265, 1981.

[121] B. W. Lampson. Hints for computer systems design. *ACM SIGOPS Operating Systems Review*, 15(5):33–48, 1983.

[122] B. W. Lampson and R. F. Sproull. An open operating system for a single-user machine. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 98–105, Asilomar, California, USA, December 1979. ACM.

[123] P. J. et. al. Leach. UIDs as internal names in a distributed file system. In *Proceedings of the* 1^{st} *symposium on the Principles of Distributed Computing*, pages 34–41. ACM, 1982.

[124] P. J. et. al. Leach. The architecture of an integrated local network. *IEEE Selected areas of Communications*, 1(5):842–856, November 1983.

[125] C. Lever. Using DFS without DCE/LFS. Technical Report CITI-94-2, Center for Information Technology, University of Michigan, 519 West William Street, Ann Arbor, MI 48103-4943, January 1994.

[126] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–374, December 1990.

[127] E. J. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988. *Also in* Proceedings of the 11th Symposium on Operating System Principles. ACM. Austin Texas. Nov 1987.

[128] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, Shrira L., and Williams M. Replication in the Harp file system. *ACM SIGOPS Operating Systems Review*, 25(5):226–238, October 1991.

[129] A. Litman. DUNIX –A distributed Unix system. *ACM SIGOPS Operating Systems Review*, 22(1):42–50, January 1988.

[130] G. W.R Luderer, H. Che, J. P. Haggerty, P. A. Kirslis, and W. T. Marshall. A distributed Unix system based on a virtual circuit switch. *ACM SIGOPS Operating Systems Review*, 15(5):160–168, December 1981.

[131] B. Lyon. External data representation specification. Technical report, Sun Microsystems Inc., Mountain View, CA, USA, 1984. *Also* RFC1014.

[132] B. Lyon. Sun remote procedure call specification. Technical report, Sun Microsystems Inc., Mountain View, CA, USA, 1984. *Also* RFC1050.

[133] R. Macklem. The 4.4BSD NFS implementation. *Unix BSD4.4 on-line documentation SMM:06-2*, 1993.

[134] P. W. Madany, R. H. Campbell, V. Russo, and D. E. Leyens. A class hierarchy for building stream-oriented file systems. In *ECOOP '89*, pages 311–328, Nottingham, UK, 1989.

[135] D. Major, G. Minshall, and K. Powel. An overview of the Netware operating system. In *Proceedings of the 1994 Winter USENIX Conference*, pages 355–372, Berkeley California USA, January 1994. USENIX.

[136] T. Mann, A. Hisgen, and G. Swart. An Algorithm for Data Replication. Technical Report 46, Digital Equipment Corporation Systems Research Center, June 1989.

[137] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. L. Scott. Using simple page migration policies to reduce the cost of cache fills in coherent shared-memory systems. Technical Report 94.TR535, Rochester University, June 1994.

[138] S. J. Martin, J. M. McCann, and D. R. Oran. Development of the VAX distributed name service. *DIGITAL Technical Journal*, 9:9–15, June 1989.

[139] K. Marzullo and F. Schmuck. Supplying high availability with a standard network file system. In *The 8th International Conference on Distributed Computing Systems*, pages 447–453, San Jose, CA, 13-17 Jun 1988. IEEE.

[140] R. M. Metcalfe and D. R. Boggs. Ethernet: Packet switching for local computer networks. *Communications of the ACM*, 19(7):105–117, July 1976.

[141] Sun Microsystems. RFC1094: NFS: Network file system protocol specification. Technical report, Internet Standard, March 1989.

[142] J. G. Mitchell and J. Dion. A comparison of two network-based file servers. In *Proceedings of the 8th ACM Symposium on Operating System Principles*, Pacific Grove, California, USA, December 1981. ACM.

[143] A. Mohindra and U. Ramachandran. A survey of distributed shared memory in losely-coupled systems. Technical Report GIT-CC-91/01, College of Computing, Georgia Institute of Technology, Atlanta, GA, USA, January 1991.

[144] A. W. Moore. Operating system and file system monitoring: A comparison of passive network monitoring with full kernel instrumentation techniques. Master's thesis, Monash University, 1995.

[145] A. W. Moore, A. J. McGregor, and J. W. Breen. A comparison of system monitoring methods, passive network monitoring and kernel instrumentation. *ACM SIGOPS Operating Systems Review*, 30(1):16–38, January 1996.

[146] S. Mullender and A. Tanenbaum. A distributed file service based on optimistic concurrency control. In *ACM SOSP*, December 1985.

[147] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba –A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[148] S.J Mullender. Distributed operating systems: The state-of-the-art and future directions. In *Proceedings of the EUTECO 88 Conference*, pages 57–66, Vienna, Austria, 1988. North-Holland.

[149] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[150] R. M. Needham and M. D. Schroeder. Authentication revisited. *ACM SIGOPS Operating Systems Review*, 21(1):7, January 1987.

[151] M. N. Nelson and J. K. Ousterhout. Copy-on-write for Sprite. In *Proceedings of the Summer Usenix '88 Conference*, pages 187–201, San Francisco, CA. USA, June 1988. USENIX.

[152] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1987.

[153] W. G. Nichols and J. S. Emer. Design and implementation of the VAX distributed file service. *DIGITAL Technical Journal*, 9:16–28, June 1989.

[154] J. R. Nicol, G. Blair, and J. Walpole. Operating system design: Towards a holistic approach? *ACM SIGOPS Operating Systems Review*, 21(1):11–19, January 1987.

[155] J.R. Nicol, G. Blair, J. Malik, and J. Walpole. COSMOS: An architecture for a distributed programming environment. *Computer Communications*, 12(3):147–157, 1989.

[156] B Nitzberg and V. Lo. Distributed Shared Memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[157] D. Notkin, A. P. Black, E. D. Lazowska, H. M. Levy, J. Sanislo, and J. Zahorjan. Interconnecting heterogeneous computer systems. *Communications of the ACM*, 31(3):258–273, March 1988.

[158] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *ACM SIGOPS Operating Systems Review*, 23(1):11–28, January 1989.

[159] J. K. Ousterhout. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[160] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer Conference, 1990*, Anaheim, California, USA, June 1990. USENIX.

[161] J. K. Ousterhout, H. Da Cosra, D. Harrison, Kunze J.A., Kupfer M., and Thompson J.G. A trace-driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the 10^{th} ACM Symposium on Operating System Principles*, pages 15–24, Orcas Island, WA, USA, December 1985. ACM.

[162] T. W. Page, R. G. Guy, and G. J. Popek. Architecture of the Ficus scalable replicated file system. Technical Report CSD-910005, Department of Computer Science, University of California Los Angele, March 1991.

[163] J. Y.C. Pang, D. S. Gill, and S. Zhou. Implementation and performance of cluster-based file replication in large-scale distributed systems. Technical Report CSRI-274, Computer Science Research Institute, University of Toronto, Toronto, Ontario, Canada, M5S 1A1, August 1992.

[164] J-F Pâris. Voting with witnesses: A consistency scheme for replicated files. In *The 6^{th} International Conference on Distributed Computing Systems*, pages 606–612, Boston, Mass., May 1986. IEEE.

[165] J.-F. Pâris. Voting with bystanders. In *The 9^{th} International Conference on Distributed Computing Systems*, pages 394–401, Newport Beach, CA USA, June 1989. IEEE.

[166] J.-F. Pâris and W. F. Tichy. STORK: An experimental migrating file system for computer networks. In *Proceedings of the IEEE INFOCOM '83*, pages 168–175, San Diego, CA, USA, April 1983. IEEE.

[167] D. S. Jr. Parker, G. Popek, G. Rudisin, Stoughton D. A., Walker B. J., Walton E., Chow J. M., Edwards D., Kiser S., and Kline C. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[168] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, pages 109–116, Chicago, IL, USA, June 1988. ACM.

[169] W. H. Paxton. A client based transaction system to maintain data integrity. In *Proceedings of the 7^{th} ACM Symposium on Operating System Principles*, pages 18–23, Asilomar, California, USA, December 1979. ACM.

[170] R. Pike, D. Presotto, R. Pike, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. Technical Report 92-1-07, AT&T Bell Labs, Murray Hill, New Jersey 07974, USA, January 1992.

[171] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Proceedings of the EUUG Spring '88 Conference*, pages 15–21, London, April 1988. European Unix Users Group.

[172] C. B. Pinkerton, E. D. Lazowska, D. Notkin, and J. Zahorjan. A heterogeneous distributed file system. In *The 10^{th} International Conference on Distributed Computing Systems*, pages 424–431, Paris, FRANCE, May 1990. IEEE.

[173] D. Plummer. Ethernet address resolution protocol: Or converting network protocol addresses to 48-bit Ethernet address for transmission on Ethernet hardware. Technical Report RFC826, Internet Standard, November 1982.

[174] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A network transparent, high reliability distributed system. In *Proceedings of the 8^{th} ACM Symposium on Operating System Principles*, pages 169–177, Pacific Grove, California, USA, December 1981. ACM.

[175] J. Postel. User datagram protocol. Technical Report RFC768, DARPA, August 1980.

[176] J. Postel. Internet control message protocol. Technical Report RFC792, DARPA, September 1981.

[177] J. Postel. Internet protocol. Technical Report RFC791, DARPA, September 1981.

[178] S. Prabhakar, D. Agrawal, A. El Abbadi, and A. Singh. Tertiary storage: Current status and future trends. Technical Report TRCS-96-21, Computer Science, University of California, Santa Barbara, Santa Barbara, CA 93106, USA, 1996.

[179] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9, a distributed system. In *Proceedings of the Spring 1991 EurOpen Conference*, pages 43–50, May 1991.

[180] H. C. Rao and L. Peterson. Accessing files in an internet: The Jade File System. *IEEE Transactions on Software Engineering*, 19(6):613–624, June 1993.

[181] R. F. Rashid and R. Fitzgerald. The integration of virtual memory and interprocess communication in Accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[182] D. P. Reed. Naming and synchronisation in a decentralized computer system. Technical Report MIT/LCS/TR-205, MIT, 1978.

[183] D. P. Reed and L. Svobodova. *SWALLOW: A Distributed Data Storage System for a Local Network*, pages 355–373. North-Holland, 1981.

[184] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and Popek G. J. Resolving file conflicts in the Ficus file system. In *The 1994 USENIX Conference*, June 1994.

[185] P. Reiher, T. Page, G. Popek, J. Cook, and S. Crocker. Truffles –A secure service for widespread file sharing. In *Proceedings of the Privacy and Security Research Group 1994 Workshop on Network and Distributed System Security*, 1994.

[186] M. Richards, A. Aylward, P. Bond, R. Evans, and B. Knight. TRIPOS: A portable operating system for mini-computers. *Software Practice and Experience*, 9(7):513–526, July 1979.

[187] D. M. Richie and K. Thompson. The Unix time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[188] A.P. Rifkin, M.P. Forbes, R.L. Hamilton, M. Sabrio, S. Shah, and K. Yueh. RFS architectural overview. In *Proceedings of the Summer 1986 USENIX Conference*, pages 248–259, Atlanta Georgia, June 1986. USENIX.

[189] M. Rosenblum and J. K. Ousterhout. The LFS storage manager. In *The USENIX Summer 1990 Conference*, pages 315–324, Anaheim, California, USA, June 1990. USENIX.

[190] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, Pacific Grove, California, USA, December 1991. ACM.

[191] L. Rowe and K. Birman. A local network based on the Unix operating system. *IEEE Transactions on Software Engineering*, 8(2):137–146, March 1982.

[192] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Gullemont, F. Herrmann, C. Kaiser, S. Langlois, Léonard P., and Neuhauser W. Overview of the CHORUS distributed operating systems. Technical Report CS/TR-90-25, Chorus Systèmes, Marc Shapiro, INRIA, April 1990.

[193] J. H. Saltzer, D. P. Reed, and D.D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[194] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the 1985 Summer USENIX Technical Conference*, pages 119–130, Portland, Oregon, USA, June 1985. USENIX.

[195] H. S. Sandhu and S. Zhou. Cluster-based file replication in large-scale distributed systems. In *Proceedings of the ACM SIGMETRICS and Performance 1992 Conference*. ACM, April 1992. *Also in* Computer Systems Research Institute, University of Toronto technical report CSRI-255, Jan 1992.

[196] A. Sandoz. Achieving high availability in a replicated file system by dynamically ordering transactions. In *The 10th International Conference on Distributed Computing Systems*, pages 432–439, Paris, FRANCE, May 1990. IEEE.

[197] M. Satyanarayana, M. Howard, J. H. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC distributed file system: Principles and design. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 33–50, Orcas Island, Washington, USA, December 1985. ACM.

[198] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, August 1989.

[199] M. Satyanarayanan. *Distributed File Systems*, pages 353–383. ACM Press, 1993.

[200] M. Schroeder, D. Gifford, and R. Needham. A caching file system for a programmer's workstation. In *The Tenth ACM Symposium on Operating Systems Principles*, pages 25–34. ACM, December 1985.

[201] A. Shaw. Operating systems. *Communications of the ACM*, 31(3):255–257, March 1988.

[202] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16:57–69, August 1983.

[203] A. Siegel. Deceit: A flexible distributed file system. Technical Report TR 89-1042, Department of Computer Science, Cornell University, Ithaca, NY, USA, November 1989.

[204] A Siegel. *Performance in Flexible Distributed File Systems.* PhD thesis, Cornell University, May 1992. *Also in* tech. report no. TR-92-1266.

[205] W. E. Snaman and D. W. Thiel. The VAX/VMS distributed lock manager. *DIGITAL Technical Journal*, 5:29–44, September 1987.

[206] M. H. Solomon and R. A. Finkel. The ROSCOE distributed operating system. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 108–114, Pacific Grove, Calif, Dec 1979. ACM.

[207] V. Srinivasan and J. Mogul. Spritely NFS: Experiments with cache-consistency protocols. *ACM SIGOPS Operating Systems Review*, 23(5):45–57, December 1988.

[208] V. Srinivasan and J. C. Mogul. Spritely NFS: Implementation and performance of cache-consistency protocols. Technical Report 89/5, Digital Western Research Laboratory, Palo Alto, California, USA, May 1989.

[209] H. Sturgis, J. Mitchell, and J. Israel. Issues in the design and use of a distributed file server. *ACM SIGOPS Operating Systems Review*, 14(3):55–69, July 1980.

[210] T. E. Sutherland. Sketchpad: A man-machine graphical communications system. In *Proceedings of the Spring 1963 AFIPS Conference*, pages 329–346, 1963.

[211] L. Svobodova. File servers for network-based distributed systems. *ACM Computing Surveys*, 16(4):353–398, December 1984.

[212] D. Swinehart, G. McDaniel, and D. Boggs. WFS: A simple shared file system for a distributed environment. In *Proceedings of the 7^{th} ACM Symposium on Operating System Principles*, pages 9–17, Asilomar, California, USA, December 1979. ACM.

[213] C. D. Tait. Techniques for building highly available distributed file systems. Technical Report CUCS-497-87, Columbia University Department of Computer Science, New York, NY 10027 USA, March 1990.

[214] A. S. Tanenbaum. MINIX: A Unix clone with source code for the IBM PC. *;login:*, 12(2):3–9, March 1987.

[215] A. S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, 1987.

[216] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall, 1989.

[217] A. S. Tanenbaum and S. J. Mullender. An introduction to Amoeba. In *AMOEBA: Collected Papers*, pages 4–10, Amsterdam, The Netherlands, 1991. Department of Mathematics and Computer Science, Vrije Universiteit.

[218] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *The 6^{th} International Conference on Distributed Computing Systems*, pages 558–563, Boston, Mass., May 1986. IEEE.

[219] A. S. Tanenbaum and R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–469, December 1985.

[220] A. S. Tanenbaum, R. van Renesse, H. van Staveren, Sharp G.J., Mullender S.J., Jansen A.J., and G. van Rossum. Experiences with the Amoeba distributed operating system. Technical Report IR-194, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, July 1989.

[221] C. K. Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of the National Computer Conference*, pages 749–753. AFIPS, October 1976.

[222] M. Theimer, L. Cabrera, and J. Wyllie. QuickSilver support for access to data in large, geographically dispersed systems. In *The 9^{th} International Conference on Distributed Computing Systems*, pages 28–35, Newport Beach, CA USA, June 1989. IEEE.

[223] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transaction on Database Systems*, 4(2):180–209, June 1979.

[224] W.F. Tichy and Z. Ruan. Towards a distributed file system. In *Proceedings of the 1984 Summer USENIX Technical Conference*, pages 87–97, Salt Lake City, Utah, USA, summer 1984. USENIX.

[225] G. M. Tomlinson, D. Keefe, I. C. Wand, and A. J. Wellings. The PULSE distributed file system. *Software Practice and Experience*, 15(11):1087–1101, November 1985.

[226] G. W. Treese. Berkeley Unix on 1000 workstations: Athena changes to 4.3BSD. In *Proceedings of the Winter 1988 Usenix Conference*, Dallas, Texas, USA, February 1988. USENIX.

[227] A. R. Tripathi and N. M. Karnik. Trends in multiprocessor and distributed operating systems. Technical report, Department of Computer Science University of Minnesota, Minneapolis MN 55455, 1995.

[228] S. S. Turing. *Alan M. Turing.* W. Heffer, Cambridge, 1959.

[229] K. Uehara, H. Miyazawa, K. Yamamoto, S. Inohara, and T. Masuda. A framework for customizing coherence protocols of distributed file caches in Lucas file system. Technical Report TR94-14, Department of Information Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-Ku Tokyo, 113 Japan, December 1994.

[230] L. J. van Moergestel, H. E. Bal, M Kaashoek, R. van Renesse, G. J. Sharp, H. van Staveren, and A Tanenbaum. Amoeba on a multiprocessor. Technical Report IR-206, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, 1989.

[231] R. van Renesse and A. S. Tanenbaum. Voting with ghosts. In *The 8^{th} International Conference on Distributed Computing Systems*, pages 456–461, San Jose, CA, 13-17 Jun 1988. IEEE.

[232] R. van Renesse, A. S. Tanenbaum, and H. van Staveren. Connecting RPC-based distributed systems using wide-area networks. In *The 7^{th} International Conference on Distributed Computing Systems*, pages 28–34. IEEE, 1987.

[233] R. van Renesse, A. S. Tanenbaum, and A. Wilschut. The design of a high-performance file server. In *The 9^{th} International Conference on Distributed Computing Systems*, pages 22–27, Newport Beach, CA USA, June 1989. IEEE.

[234] R. van Renesse and H. van Staveren. Wide-area communication under Amoeba. Technical Report IR-117, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, December 1986.

[235] R. van Renesse, H. van Staveren, and A. S. Tanenbaum. Performance of the Amoeba distributed operating system. *Software Practice and Experience*, 19:223–234, March 1989.

[236] G. van Rossum. AIL –A class-orientated stub generator for Amoeba. In *Proceedings of the Workshop on Experience with Distributed Systems*, Heidelberg Federal Republic of Germany, September 1989. Springer Verlag.

[237] F. Vaughan, T.L. Basso, A. Dearle, C. Marlin, and C. Barter. Casper: a cached architecture supporting persistence. *Computing Systems*, 5(3):337–359, Summer 1992.

[238] V. L. Voydock and S. T. Kent. Security mechanisms in high-level network protocols. *ACM Computing Surveys*, 15(2):135–171, June 1983.

[239] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. *ACM SIGOPS Operating Systems Review*, 17(5):49–70, October 1983.

[240] B. J. Walker. Distributed Unix transparency: Goals, benefits and the TCF example. *CommUNIXations*, pages 14–30, July 1989.

[241] C. E. Walston. Information retrieval. *Advances in Computers*, 6:1–30, 1965.

[242] R. Y. Wang and T. E. Anderson. xFS: A wide area mass storage file system. Technical Report UCB//CSD-94-783, University of California, Berkeley, Berkeley, California 94720, December 1993.

[243] P.J. Weinberger. The version 8 network file system (abstract). In *Proceedings of the 1984 Summer USENIX Technical Conference*, page 86, Salt Lake City, Utah, USA, summer 1984. USENIX.

[244] B. B. Welch. Naming, state management, and user-level extensions in the Sprite distributed file system. Technical Report UCB/CSD 90/567, EECS, University of California, Berkeley, California 94720, April 1990.

[245] J. Wolf. The placement optimization problem: A practical solution to the disk file assignment problem. *ACM SIGMETRICS*, pages 1–10, May 1989.

[246] W. Wulf, E. Cohen, W. Corwin, A. Jone, R. Levin, C. Pierson, and F. Pollark. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.