Technical Report

UCAM-CL-TR-153
ISSN 1476-2986

Number 153

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Efficient data sharing

Michael Burrows

December 1988

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

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

This technical report is based on a dissertation submitted September 1988 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

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

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

ISSN 1476-2986

Preface

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

This dissertation is not substantially the same as any that I have submitted or am currently submitting for a degree, diploma or any other qualification at any other university.

MICHAEL BURROWS
DECEMBER 9, 1988

Trademarks

Acknowledgments

I am indebted to many who gave me advice, criticism and their time during the course of this research. I would like to thank:

- David Wheeler, my supervisor. His suggestions and comments were always provocative and stimulating. He broadened my interests to encompass many areas of computer science outside the topic of my research, for which I am grateful.
- Roger Needham, the head of the Computer Laboratory. His knowledge in the field of distributed systems was invaluable. His discussions were always interesting, and his suggestions always helpful.
- Martyn Johnson, the system manager of most Computer Laboratory machines. He put many valuable resources at my disposal, which often required large amounts of his own time. The Demand-Initialized Disc system described in Chapter 3 was based on his original idea.

I would like to thank the people who provided encouragement or suggested improvements to this dissertation. They include Steve Crawley, Paul Curzon, Joe Dixon, Andy Gordon, Stephen Harrison, Martyn Johnson, Ian Leslie, Sape Mullender, Roger Needham, Cosmos Nicolaou, Gianpaolo Tommasi, David Wheeler and John Wilkes .

I used the Andrew benchmark in measuring the performance of my caching systems. I should like to thank the author of the benchmark, M. Satyanarayanan of the Information Technology Center of Carnegie-Mellon University, for permission to use it, and to reproduce some of his results for comparison with my own.

The work was supported by a studentship from the Science and Engineering Research Council.

Summary

As distributed computing systems become widespread, the sharing of data between people using a large number of computers becomes more important. One of the most popular ways to facilitate this sharing is to provide a common file system, accessible by all the machines on a network. This approach is simple and reasonably effective, but the performance of the system can degrade significantly if the number of machines is increased. By using a hierarchical network, and arranging that machines typically access files stored in the same section of the network, it is possible to build very large systems. However, there is still a limit on the number of machines that can share a single file server and a single network effectively.

A good way to decrease network and server load is to cache file data on client machines, so that the data need not be fetched from the centralized server each time it is accessed. This technique can improve the performance of a distributed file system, and is used in a number of working systems. However, caching brings with it the overhead of maintaining *consistency*, or cache coherence. That is, each machine in the network must see the same data in its cache, even though one machine may be modifying the data as others are reading it. The problem is to maintain consistency without dramatically increasing the number of messages that must be passed between the machines in the network.

Some existing file systems take a probabilistic approach to consistency, some explicitly prevent the activities that can cause inconsistency, while others provide consistency only at some cost in functionality or performance. In this dissertation, I examine how distributed file systems are typically used, and the degree to which caching might be expected to improve performance. I then describe a new file system that attempts to cache significantly more data than other systems, provides strong consistency guarantees, yet requires few additional messages for cache management.

This new file system provides fine-grain sharing of a file concurrently open on multiple machines in the network, at the granularity of a single byte. It uses a simple system of multiple-reader, single-writer locks held in a centralized server to ensure cache consistency. The problems of maintaining client state in a centralized server are solved by using efficient data structures and crash recovery techniques.

Contents

List of Tables

List of Figures

Glossary

This list defines abbreviations and some technical terms used in the text. A most important set of distinctions is retained throughout this dissertation: clients are machines, users are people and applications are programmes that run on clients.

Andrew A file system built at the Information Technology Center of Carnegie Mellon University [Howard 88].

application A programme running on a machine on behalf of a *user*.

cache A local copy of remote data, which can be accessed more readily than the original. To store data in a cache.

callback In *Andrew*, a guarantee that the *file server* will contact a *client* if a file is modified. Equivalent to holding a read-*token* in MFS.

client A machine that calls upon a *server* to perform some operation, such as storing a file. Compare with *user*.

CFS The Cedar File System [Schroeder 85], developed at Xerox PARC.

coherence Cache coherence is synonymous with cache *consistency*.

consistency A caching system is said to be consistent if an ordered set of data accesses always produces the same results under the caching system as it would have done without caching.

DID Demand-Initialized Disc system. A caching system built on top of RVD. Described in Chapter 3.

file server A machine that allows *clients* to store and retrieve data files by name.

host A computer with access to a network.

LRU Least Recently Used.

MFS Mike's File System. A caching file system described in Chapter 3.

metadata Information associated with files, but not contained within them. E.g. names, permissions information, etc.

NFS Sun Microsystems' Network File System [Sandberg 85].

RFS AT&T's Remote File Sharing [Bach 87].

RPC Remote Procedure Call. Invocation of a procedure at a *server* by a *client.*

RVD The Remote Virtual Disc system, developed at MIT [Treese 88].

server A machine that offers some service to *clients*, such as storing files, giving the time of day etc. Normally, one server is able to handle requests from many clients.

Sprite A distributed operating system built at the University of California at Berkeley [Nelson 88].

token In *MFS*, a logical permit to *cache* data, used to maintain cache consistency. Possession of a read-token allows the data to be cached for reading; a write-token allows both reading and writing. Tokens are managed by the *token server.*

token server A *server* that keeps track of *tokens*, reclaiming them from clients when they are needed by others.

user A person who initiates activities on a computer. The programmes that he runs are *applications.* Compare with *client.*

Chapter 1

Introduction

This dissertation considers the use of caching in distributed file systems. It argues that effective caching techniques can improve file system performance without sacrificing consistency guarantees or reliability.

This chapter introduces the notion of a distributed file system. It discusses how caching can improve the performance of such a system, the problems of cache consistency, and the techniques available for maintaining consistency. The chapter concludes by stating the goals of the dissertation and summarizing the contents of each chapter.

1.1 Distributed File Systems

Distributed file systems have been an active area of research for many years. Their importance is obvious—they are a focal point for information storage, retrieval, naming and sharing in many distributed computing systems. Their importance grows as the size and popularity of distributed systems increases.

In many cases, distributed file system interfaces are based on the interfaces of time-shared file systems. Some file systems (e.g. the Cedar File System [Schroeder 85]) are different in style, but such systems are rare, and none has gained wide acceptance.

The basic properties of file systems are well established—they allow application programmes to store data files and retrieve them by name. The storage is usually non-volatile, and usually larger in size than the main memory of a machine. File names are often textual and human readable. There are several reasons for placing data in a file system:

- The data will survive machine crashes.
- The data can exceed the size of main memory.
- Files can be given mnemonic names for easy reference by users.
- Files can be retrieved by programmes and users other than the ones that generated them.

It is possible to build a distributed system in which each machine has its own local file system for the storage of its own data. However, experience with such systems suggests that this situation is far from ideal. One result is that each machine is limited by the size of its own store, rather than the total amount of storage in the system. Worse, the data can only be named and accessed from a single machine—it cannot be shared between machines.

In the normal course of using a computer system, there is potential for a great deal of file sharing. Users share copies of standard system files, and share their own files amongst their colleagues. Normally, the sharing is between a group of people working in related areas, though sometimes there is a need to share information in a much larger group. In most cases, file sharing does not involve concurrent access to the data.

File transfer protocols allow a user to retrieve data stored on remote machines, but this is inconvenient for any system larger than a handful of machines. This approach leads to multiple copies of files being scattered through the system, with no guarantees that the copies are *consistent* with one another. Checking the consistency of a copy against a master copy can be expensive, particularly for files shared between a large number of machines. Keeping track of the copies, and locating the most recent version soon becomes tedious.

A solution to these problems is for machines to allow direct access to their file systems by other machines on the network. A *client* machine can then access the files of a *file server*. Transparent file access can be provided by placing both local and remote files in the same name space. The most important aspects of distributed file system design include:

- naming—how users and machines identify files;
- file location—who or what controls it, and how;
- internal file structure;
- the programmer's interface;
- access control.

These issues have been explored in many different file system designs, but are not central to the arguments presented here. This dissertation concentrates on three areas associated with the basic notion of accessing data in a distributed system:

- performance;
- consistency in the presence of file sharing—what guarantees are provided at what granularity;
- failure properties—what happens when a machine, network or disc fails.

In particular, later chapters consider the use of *caching* to increase performance, and how this affects file system consistency and failure properties.

1.2 Caching

Caching is an extremely powerful technique in distributed systems. The principle is simple: copies of commonly used information are held close to the clients that need it, so that operations on it can take place without interactions with remote machines. The effect on performance can be dramatic. In a later chapter, we shall see that a file system cache on a UNIX [Ritchie 78] machine can reduce the number of disc accesses by a factor of ten or more. This is undoubtedly a useful gain, but it comes at the cost of additional complexity in the clients, the servers and the network protocols that are used between the two.

The main problem with caching is the need to maintain cache consistency (sometimes called cache coherence) in the face of *write-sharing*. For example, if two clients on two different machines each cache a copy of a file, and then one updates its local copy, the other must update its cache before it can read its copy once more. Nelson [Nelson 88] distinguishes two forms of write-sharing:

Sequential write-sharing takes places when a file is read and written on multiple machines, but is never open[1] simultaneously on more than one.

Concurrent write-sharing takes places when a file is simultaneously open for reading and writing on more than one machine.

The problems of write-sharing have been tackled in a number of differing ways, which are summarized below.

1.3 Avoiding Inconsistency

1.3.1 Immutability

Immutable data items have the property that once they have been created, they cannot be modified. Immutable objects have many advantages in a distributed environment. In particular, a cache of immutable objects does not require any maintenance to remain consistent. Since the value of an object cannot change, two copies of the same object can never differ.

In the Cedar File System, files are viewed as a set of versions of immutable objects [Schroeder 85]. Each time a modification is required, a new object is created. This approach is reasonably efficient in most situations since files are almost always written in their entirety [Nelson 88], and old versions of files can be removed when not required. Of course, there are always some files that are incrementally modified, such as logs and databases, but these are often accommodated by other services [Brown 85].

Unfortunately, it is not possible for a caching system to take full advantage of the properties of immutable files unless files can be identified uniquely. If a file's

[1]This assumes that an application declares its intention to read or write a file by *opening* the file. Related accesses to the file are grouped between bracketing *open* and *close* operations.

identifier can be changed, the same identifier may be used in two different places to refer to different files. The property required is that the file is immutable with respect to its identifier; if an identifier refers to any file at all, it always refers to the same file. This permits files to be deleted, provided their identifiers are never reused.

The Cedar File System includes version numbers in file names to allow the allocation of a new name to each new version of a file. Remote files are typically accessed through a version management system that retrieves and saves consistent sets of files [Schmidt 82]. Since the version management system keeps track of file version numbers on behalf of the user, the full name of each file is used when it is accessed, and the binding between name and contents is preserved. However, users are accustomed to referring to files by an unqualified name, without quoting explicit version numbers or timestamps. Cedar will automatically choose the highest numbered version when no explicit version number is specified. This invalidates the assumption that files are immutable with respect to their names, since the same (unqualified) name can now refer to many different files. Cache consistency again becomes an issue—the mutable directory must be up to date in order to correctly find the latest version of a particular file. The system must do the same amount of work to detect inconsistency of the directory as would be required for the file itself, if the file were mutable.

From this we can conclude that cache consistency is a problem even when using immutable files. Moreover, adding support for cache consistency of mutable directories allows the possibility of other mutable files, such as log files and databases.

1.3.2 Disallowing Sharing

There are two necessary conditions for inconsistency in a distributed caching system: sharing and modification. Inconsistency is impossible if data is immutable, or if it is not shared. Chapter 3 describes a file system in which shared files are immutable, and writable files cannot be shared. This policy ensures cache consistency, but imposes undesirable constraints on the users.

1.3.3 Maintaining Cache Consistency

Cache consistency in a distributed system can be maintained by a set of algorithms and protocols that communicate the actions of one client to other interested clients. Several possibilities exist, depending on the constraints placed on the design. The most important considerations are:

- the delays observed by a client when reading and writing files;
- the granularity of sharing and caching required;
- the load imposed on the clients and servers when a file is accessed;
- state required in the clients and servers about current clients;

- behaviour in the presence of machine failure and/or network partition.

Two popular strategies for controlling cache consistency are:

- validate before use—the client interrogates a master copy of the object before using the value in its cache (e.g NFS [Sandberg 85]);
- invalidate on write—a centralized server undertakes to inform clients when their caches are about to become inconsistent (e.g. Andrew [Howard 88]).

The first strategy requires very little server state, which simplifies crash recovery slightly. However, its performance is generally poor, especially when the number of clients is large.

Invalidation schemes require the server to remember what each client has cached, and this complicates crash recovery. However, the performance tends to be better, both because no network accesses are needed when the cache is accessed, and because the server is not burdened with a large number of validation requests.

Hybrid schemes are also possible. In some file systems, such as RFS [Bach 87] and Sprite [Nelson 88], clients validate files as they are opened, and are informed when concurrent write-sharing takes place.

1.3.4 Conclusion

The sections above have considered restricting file modification and file sharing in order to deal with the problems of cache inconsistency. Neither solution is ideal. Immutable files cannot support certain types of file access, and do not fully solve the problems of maintaining consistency when files are accessed by unqualified names. Restricting the ways in which files can be shared will inconvenience groups of users working with related files, and will force distributed applications to use other techniques for data sharing.

Cache consistency algorithms allow for the possibility of mutable files, and sharing, at some cost in complexity.

1.4 Hypothesis

This dissertation aims to show that:

- The performance of distributed file systems can be greatly improved by caching copies of remote files on client machines.
- The consistency of these copies can be maintained in an efficient and practical manner, even in the face of sharing at the granularity of individual reads and writes of a few hundred bytes each.
- Efficient crash recovery is possible without excessive loss of file data, even though the caching techniques presented make heavy use of server state.

- The performance of the resulting file system degrades slowly as the number of active clients per server increases.

The arguments are presented without particular reference to the other aspects of file system design. I claim that the caching systems described could be made to work with a range of naming schemes, replication mechanisms and file structures.

1.5 Synopsis

Chapter 2 presents measurements taken on a UNIX system and combines them with information gathered by other researchers. This information guides the implementation decisions made in other chapters.

Chapter 3 presents a simple caching system, and describes its implementation. The performance of the system under an artificial load, and experience with actual use of the system in a medium-sized environment is discussed. The shortcomings of the system are presented.

Chapter 4 presents a prototype implementation of a distributed file system that overcomes the difficulties described above. By storing information about the state of each client in a centralized place, the system provides:

- fine-grain sharing;
- good consistency guarantees;
- good performance;
- good scaling.

Algorithms for replication and crash recovery are described.

In Chapter 5, the behaviour of the file system is compared with a standalone UNIX system. Also, the performance of the system under an artificial load is analysed and compared with existing file systems.

Related work is presented in Chapter 6, which examines the caching systems of several existing file systems. The advantages and drawbacks of each system are discussed and compared with the prototype. The chapter shows that the proposed file system differs substantially from the others described.

Chapter 7 summarizes the main conclusions of the dissertation.

Chapter 2

Measuring a File System

When designing a new file system, it is important to know how it will be used; one of the best ways of predicting this is to examine existing file systems. This chapter describes how the file system of a time-shared UNIX system was monitored to obtain information about file usage and sharing. Since others have also traced file systems in considerable detail, results are presented only in brief. The results of this chapter are intended to give the reader a reasonable idea of the file system activity on a UNIX system in a university research environment. Important data reported by other researchers are summarized.

2.1 The Measurements

The file system examined was that of a time sharing UNIX machine running Berkeley 4.2BSD. The machine was a VAX-11/750, which typically had 10 users logged in during the day. Two types of measurement were performed:

- static measurements on existing files;
- dynamic measurements of how processes manipulate files.

The static analysis involved examining a snapshot of file system state. It provided information about typical file sizes, numbers of files, and the number of long-lived files that are read and written frequently. The dynamic analysis involved collecting information from the operating system as it was running. It provided detailed file usage information, including sufficient information to drive simulations of file caching schemes.

2.1.1 Static Measurements

The file system of a timesharing UNIX machine was scanned, and file size, last-accessed time and last-modified time were recorded. Samples were taken in the evening of 14 consecutive days. Table 2.1 shows the distribution of files and data by file size. Table 2.2 shows the distribution of files and data by last-accessed and last-modified time.

	File Size							
	0–4K	4–16K	16–64K	64–256K	256–512K	512K–1M	1M+	Total
Files	47544	12387	4129	635	65	49	54	64863
Data (Kb)	52243	100409	118647	74827	20824	35526	95988	498466

The table shows the distribution of files and file data according to files size. Figures are the means of measurements taken at 6pm on 14 consecutive days.

Table 2.1: Distribution of Files and File Data by File Size

Accessed/Modified within	Files Accessed	Data Accessed (KB)	Files Modified	Data Modified (KB)
0–10 min	198	2833	35	878
10–30 min	181	2380	31	904
0.5–1 hr	252	3337	34	708
1–2 hrs	369	5656	62	1621
2–4 hrs	489	6309	66	2377
4–6 hrs	169	2002	71	536
6–8 hrs	225	2730	52	910
8–10 hrs	59	1085	18	227
10–12 hrs	31	256	10	108
0.5–1 day	798	7074	161	2954
1–2 days	1434	16563	353	6219
2–4 days	2841	27915	755	10406
4–8 days	5466	43806	2684	22644
8+ days	52351	376514	60531	447968
Total	64863	498460	64863	498460

The table shows the distribution of file last-accessed times and last-modified times for files and file data. Higher rows show data in more recently accessed or modified files. Figures are the means of measurements taken at 6pm on 14 consecutive days.

Table 2.2: Distribution of Files and Data by Access/Modified Time

Most of the information revealed by the tables is well known:

- Most files are small—73% are less than 4 kilobytes.
- A large fraction of file data is contained in fairly large files—69% of all file data is held in files longer than 16 kilobytes. 19% of all file data is held in files longer than 1 megabyte.
- Most files and most file data are not accessed or updated frequently. In 8 days:
  - 81% of files (76% of file data) are not accessed;
  - 93% of files (90% of file data) are not modified.
- Over a short period, more files are read than written, and more file data is read than written.
- File system activity varies greatly with time of day—it peaks in the middle of the afternoon, and dips at night.

In addition, 80% of directories were 512 bytes long; 98% were 4 kilobytes or less.

2.1.2 Dynamic Measurements

Dynamic measurements are harder to perform than static measurements:

- The volume of data is high.
- The measurements should not degrade the performance of the system significantly.
- Dynamic measurements often require modifications to the existing operating system.

The method of gathering information was simple. The UNIX kernel was modified to record various events, such as name lookups and system calls, in a circular memory buffer. This buffer was periodically scanned by a user process that read the information and recorded it in a file. The buffer was scanned often enough to ensure that all data was captured. The fraction of CPU time occupied by the tracing mechanism was measured to be less than 2% in normal use.

Two distinct sets of measurements were made. The first recorded file name lookups, together with a flag indicating whether the file was being created. The second recorded more detailed information about a number of system calls, including most file system operations, but not including individual reads and writes. In each case, several traces were taken over a number of days.

Instances of file sharing were detected by recording the user identifier of the process generating each entry in the traces. By modelling the system activity as a collection of independent login sessions, it was possible to estimate how much sharing would take place between single-user workstations carrying the same workload.

Name Lookups

Name lookup traces were taken on three consecutive days during normal working hours. Overnight traces were biased by the large number of lookups by processes scanning the entire file system to remove unwanted temporary files. Table 2.3 summarizes the results, which do not include any results from overnight traces, except where noted.

Mean lookups/second	4
Max lookups/second (mean over 60s)	22 (245 overnight)
Unsuccessful lookups	18% of all lookups
Regular files	92% of accessed files
Files accessed by > 1 user	10% of regular files accessed
Newly created files	5% of regular files accessed
Shared, newly created files	0.5% of regular files accessed

The table shows the results of tracing name lookups on a time sharing UNIX machine over a period of three days during working hours. A regular file is a normal file, rather than a device file or a directory.

Table 2.3: Results of Name Lookup Tracing

Instances of file sharing between users were detected by recording the user identifier of each process that did some name lookup. The results show that:

- Lookups are quite frequent.
- A significant number of file name lookups fail.
- A significant number of files are accessed by multiple users.
- Few files are read by other users soon after being created.

Further analysis of the data revealed that almost all instances of shared access to newly created files were due to interactions between user processes and background system processes, rather than between two users' processes. All such accesses could be explained by one of the following:

- network statistics files, written by the *rwhod* process;
- news, written by news processes;
- mail, written by the mail delivery system;
- the line printer system reading files to be printed;
- jobs submitted to be executed at a later date;
- modifications to the password database;
- reading the new *message of the day*, updated by the system manager.

In addition, a number of files appeared to be shared between two user identifiers because a system administrator switched to a privileged account in order to install a file that had just been created. In a distributed environment, each of these services could be localized to a single machine, or managed by a separate subsystem, such as a mail system. The design of each individual service will dictate whether these activities actually require shared access to files.

Some examples of shared access remained:

- installation of a new programme common to a group of users;
- users working in common source directories;
- exclusive locking of a user resource by creating a lock file.

The combination of these amounted to only 5% of all newly created files that were shared, or 0.025% of all regular files accessed.

System Call Traces

Traces were taken over a period of three days, during working hours. For each system call executed, the following information was recorded:

- the type of call;
- the user identifier of the process;
- the *i-number* of the file(s) affected;
- the size of the file.

Additional data was included for the *open* system call to indicate whether the file was opened for writing. Rough timing information was available from the process reading the data from the kernel buffer, but timing information was not recorded in each trace entry. The following operations were not recorded by the trace in order to reduce its size: *read*, *write*, *lseek* (change position within a file).

Operation	Frequency	Comments
stat	198333	Find file information
open	120310	Open/create a file
close	120300	Close an open file
fstat	84753	Find file information for open file
exec	21082	Execute a programme
chdir	17177	Change working directory
unlink	11414	Remove a file
access	8708	Check access permissions
chmod	3269	Change access permissions
link	2905	Create a second name for a file
flock	1042	Advisory file locking
chown	973	Change file owner
symlink	662	Create a symbolic link
readlink	588	Read a symbolic link
fsync	221	Flush file modifications to disc
utimes	58	Artificially set last-modified time
rename	29	Rename a file
mkdir	28	Make a directory
rmdir	28	Delete a directory
ftruncate	11	Truncate an open file
fchmod	3	Change permissions on open file
fchown	0	Change owner of open file
mknod	0	Make a new device file
truncate	0	Truncate a file

The table shows the number of file system operations performed by the kernel over three days during working hours.

Table 2.4: Frequency of File System Operations

Table 2.4 gives the frequency of file system operations recorded during the traces. The table shows that file information is frequently requested (the *stat*

call), probably by the directory listing programme. Most file system activity is based on *open*, *close*, *stat* and *fstat*.

The traces do not include paging activity, I/O due to directory operations, or accesses to file system metadata. Ousterhout estimates that these non-file accesses may account for as much as 50% of all disc accesses [Ousterhout 85]. It is more difficult to obtain information about these accesses because of the complexity of the UNIX file system code. Disc I/O statistics can be gathered, but these include the effects of in-memory caches, such as the buffer cache and the name cache.

A summary of results obtained from the system call trace is shown in Table 2.5. Comparison with the figures for the name lookup traces shows that many files are modified after being created. Additionally, write-sharing of existing files is more common than write-sharing of newly-created files. Experience with UNIX indicates that this is not a surprising result; most UNIX editors and compilers overwrite existing output files rather than creating new ones.

Mean file operations/second	7
Max file ops/second (mean over 60s)	93
Regular files	85% of accessed files
Files accessed by > 1 user	13% of regular files accessed
Files written	39% of regular files accessed
Write-shared files	8.5% of regular files accessed

The table shows the results of tracing file system calls on a time sharing UNIX machine over a period of three days during working hours. Files were considered to be write-shared if they were accessed by at least two users, and opened for writing by at least one user.

Table 2.5: Results of System Call Tracing

Despite the significant amount of write-sharing detected by the system call trace, the types of write-sharing found were identical to those causes of write-sharing described above. Some additional examples of source code sharing were discovered, as well as two random access databases which described a programming environment. In all, no more than 10% of the write-sharing detected by the traces fell into this category. This represents only 0.9% of all regular files accessed. The conclusion is that write-shared access to files is rare compared with single-user access.

Despite the evidence that write-sharing is rare on a time-shared system, this does not imply that the same will be true of a distributed system. The assumption that users are confined to a single machine is approximately correct in many existing distributed systems, but may be quite wrong in situations where the number of machines far exceeds the number of users. In such an environment, distributed applications (such as distributed compilations) may be commonplace, and these are likely to cause additional write-sharing, even though they are performing a task for only one user. Moreover, operating systems that allow processes to migrate from one machine to another can artificially introduce write-sharing, even though only one process is accessing each file. These observations suggest that write-sharing is likely to increase as distributed applications become more widespread.

Cache Simulation

The system call traces were used to drive a simple cache simulation. Information on individual reads and writes was not available, so whole file caching was simulated. Separate caches were provided for each user in order to simulate an environment where each user has one machine. The caching policy was as follows:

- File reads and writes caused the file to be cached.
- File writes caused data to be invalidated in other caches.
- Modifications were written-through to the file server without delay.
- The cache replacement policy was least-recently-used.

The simulation assumed that caches contain no useful data at the start of a login session. As a result, the miss ratio is likely to be worse than might be expected if previous cache contents were retained. Several interesting results were obtained:

- The maximum amount of cache space used in any login session examined was 15 megabytes.
- The average cache space used in a single login session was 4 megabytes, though a significant number of sessions could use as much as 10 megabytes.
- The mean read miss rate in a single login session with an unlimited cache was 36%. It ranged from 81% in short sessions, to 8% in long sessions.
- 21% of data accesses in login sessions were writes (not including system processes).

The results of other published cache simulations concentrated on the behaviour of the system as a whole, rather than considering the interactions between the individual users. The results of some of these studies are summarized below.

2.2 Other Work

Satyanarayanan [Satyanarayanan 81] and Smith [Smith 81] describe static analysis of file systems, with similar results to those given above. They also discuss the variation of file access patterns and file lifetimes with other parameters, such as file size and file type.

Smith [Smith 85] describes cache simulations based on traces from a collection of large mainframes. He analyses cache performance for a variety of cache designs, varying cache management policies and block sizes. Two important conclusions are:

- Cache block size should be increased as cache size increases. However, he warns that increasing cache block size increases the penalty for cache misses.

- Prefetching data one block ahead can significantly improve cache hit ratios in environments where sequential reading is common.

Smith's data was taken at the granularity of a disc track (about 7 kilobytes), and did not allow him to distinguish reads from writes. This prevented analysis of the effect of delayed write policies.

Ousterhout [Ousterhout 85] describes a more detailed dynamic analysis of the 4.2BSD file system, also based on system call traces. Individual reads and writes were not recorded, but *lseek* system calls were recorded, allowing the experimenters to determine which parts of each file had been accessed. Moreover timing information was included with each trace entry, permitting analysis of the performance of delayed writes. The traces do not include non-file I/O traffic, such as paging and accesses to metadata. Ousterhout's most interesting conclusions for the purposes of caching are:

- Average file system data rates are small (< 1 kilobyte/second/user).
- Peak file system data rates can be large (> 100 kilobytes/second/user).
- Most new data is destroyed soon after being written (25% after 30 seconds, 50% after 5 minutes).
- 70% of file accesses are whole-file transfers, accounting for 50% of the data accessed.
- A 4 megabyte cache can eliminate between 65% and 90% of all disc accesses, depending on delayed write policy.
- With a 4 megabyte cache, a block size of 16 kilobytes is optimal.

2.3 Conclusions

The results presented in this chapter are encouraging from the point of view of caching, if the data obtained from time-sharing systems is applicable to distributed systems. The most positive points are:

- Most file accesses are reads.
- Most newly written data is discarded soon after writing.
- Individual users access only a few megabytes of data in a single login session.
- Useful cache sizes are easily within the bounds of small discs, and medium sized semiconductor memories.
- Write-sharing is not common.
- Files are often small, and accessed as a single unit.

- The average disc bandwidth required by a user is small.

Unfortunately, the situation is not perfect:

- The use of databases is rare in the traced environment, but is common in the commercial computing world. The level of write-sharing observed may be artificially low.
- Write-sharing is likely to increase as distributed applications become more common.
- A high proportion of file data is held in a few large files, which are slow to transfer over a network.
- The peak disc bandwidth expected by a user is very much higher than the average.
- Paging and access to file system metadata may account for a large fraction of all disc accesses.

These results suggest that caches can significantly reduce the amount of network file system traffic in a programme development environment by adding caches of between 4 and 20 megabytes. Whole file caching seems attractive, but this can increase the access latency when a file is first touched. The presence of a few shared access databases indicates that a finer granularity would be more appropriate. Smith's results indicate that cache block sizes of 8 kilobytes to 64 kilobytes are likely to be best, depending on the size of the cache. Prefetching and delayed writes are important in decreasing the cache miss ratio.

Chapter 3

Demand-Initialized Discs

A simple caching scheme was developed to improve the performance of an existing file system and to gain experience with large caches. This chapter describes the previously existing system and how caching was added. Performance figures are presented and combined with experience from actual use.

3.1 Introduction

The Demand-Initialized Disc system (DID) was originally proposed by M.A. Johnson [Johnson 85] to overcome the problem of maintaining the local file systems of the machines in a *processor bank* [Needham 82].

In a processor bank, machines are typically allocated to users for short periods, and users usually have exclusive use of machines while they are allocated. Users rarely request access to a named machine; instead they request some service, such as a machine running UNIX. A management system allocates processors to users; it chooses a processor to run the requested operating system, loading the operating system if necessary. An interesting feature of the processor bank philosophy is that users often have great control over the software of the machines that they use, and are permitted to make arbitrary changes to it while the machine is allocated to them. As a result, only the hardware of a machine can be trusted after a machine has been returned to the pool; all of the software must be reloaded or checked.

A processor bank was being constructed, consisting of a number of MicroVAX computers, each equipped with a 70 megabyte disc. The most popular operating system on the machines was UNIX, though VMS was supported in a limited way. A pool of machines running UNIX was kept up to date by periodically running utilities to scan the local file system of each machine, and replace any system files that had been modified or removed. Users could then be allocated any machine, without encountering any differences in the installed software. User files were accessed via the Remote Virtual Disc (RVD) system developed at MIT by the Laboratory for Computer Science, and later enhanced as part of the Athena Project [Greenwald 85] [Treese 88].

The RVD system consists of a disc block server and a client device driver that reads and writes disc blocks via a simple request-response protocol. The

device driver behaves just like any disc driver, and is indistinguishable from other disc drivers at the file system level, except in its performance. An active client maintains a number of connections to the disc block server; each one corresponds to a separate *virtual disc*, consisting of a range of disc blocks on the server's disc. These virtual discs can be formatted as UNIX logical volumes, and mounted as part of the file system name space.

Each user is allocated a single virtual disc to store his own files. Since the UNIX file system was not designed to allow the sharing of writable file systems between machines, a virtual disc containing a UNIX file system cannot be mounted for writing on more than one machine at a time. This prevents users from sharing files effectively, and is a serious disadvantage of this approach. Read-only sharing is possible, but complicates even the most trivial updates of any files on virtual discs.

Apart from the inherent problems of the RVD system, the processor bank suffered from several serious problems, affecting both performance and management of the system:

- The time taken to scan discs for unauthorized changes was growing alarmingly as the number of machines and the amount of installed software increased.
- As more users began to use their virtual discs, the RVD servers became heavily overloaded. Even though system utilities and temporary files were stored on the local disc, the load during peak periods was becoming unacceptable.
- Users would often leave files on the local discs of the machines, preventing the space from being used by others. Worse, these files were never backed up, since the discs were not intended for permanent storage.
- As the set of installed files changed, the algorithm for detecting whether a file system was out of date became quite baroque, since the file systems varied slightly between machines, due to minor differences in system configuration.
- As more utilities and system libraries were installed, the local discs began to run out of space.

The problems of detecting file system inconsistencies can be avoided by storing system files on a single file server, shared by all the other machines. Unfortunately, this aggravates the problems of file server load, and leaves the problem of the minor differences needed between the various machines. The software can be altered to eliminate these differences, but this involves many changes, which imply additional software maintenance overheads.

The original suggestion which led to DID used a section of the local disc to store system files, but initialized it on demand from a read-only RVD. At the first access of a disc block, it was read from the remote server, and then written to the corresponding block on the local disc. Thereafter, all reads and writes to that block accessed only the local disc. Modifications could be made, but would

never be written back to the file server. This allowed a user to make arbitrary modifications to a machine while using it, but ensured that a clean file system could be obtained by rebooting.

I extended this idea to provide a general purpose cache for all RVD access, including user partitions. Under this scheme, cache blocks held data from any part of any file system. Blocks were thrown out of the cache and replaced by other blocks when necessary. As before, blocks written to system areas were written only to the cache, but a different update policy was needed for user partitions. Modifications to user RVDs were written back to the file server, as well as being stored in the cache, ensuring that both the local and the remote copies of the file were up to date.

DID is a very simple system, and does not provide facilities for sharing mutable files between groups of machines simultaneously. Its major benefit is its simplicity; since there is no possibility of sharing mutable files, no mechanism is required to guarantee consistency. Moreover, the bulk of the new code is confined to a single UNIX device driver that caches disc blocks fetched from one disc on another disc. This simplicity has allowed DID to act as an experiment in caching remote files, providing an opportunity to measure the effectiveness of client caching in a real, working environment.

3.2 The DID Device Driver

Most of the work in the caching system is done by a virtual disc driver that sits between the UNIX block I/O layer [Thompson 78] and the other disc drivers. Although the block I/O system treats this virtual disc driver as it would any other device driver, the driver has no physical device of its own. Instead, it calls on other device drivers to read and write blocks as required. The device driver can manage a number of logical drives simultaneously, each one appearing as a separate logical volume.

During use, each logical drive has two other devices associated with it. These are the *cache device* and the *initialization device* of the logical drive. Almost any random access device can be specified as cache and initialization devices, but in practice the cache device is usually a local disc, and the initialization device an RVD.

3.2.1 Caching Policies

The DID device currently supports two caching strategies:

write-to-cache: Modified blocks are written only to the cache device. Such blocks are *dirty*, and can never be flushed from the cache unless the logical drive is dismounted.

write-through: Modified blocks are written both to the cache device and to the initialization device. Blocks are never left dirty in this mode.

In both cases, blocks are read from the cache device if they exist there, and read from the initialization device otherwise. Whenever a block is read from the initialization device, it is written to the cache device. When necessary, unmodified blocks are thrown out of the cache to make way for new data, using an LRU replacement policy.

The write-through strategy was provided rather than a delayed write-back scheme because the UNIX I/O system already has a buffer cache in main memory that introduces a delay of up to 30 seconds on all writes. Increasing this delay would noticeably decrease the robustness of the file system in the face of potential client crashes.

3.2.2 Data Structures

Figure 3.1 shows the data structures of the DID device driver. For each logical drive, the driver stores the internal device numbers of the initialization and cache devices. For each distinct cache device, a hash table maps logical block numbers onto cache block numbers. Logical block numbers correspond exactly to the block numbers on the initialization device, but cache blocks may be allocated in any order. Each cache device can hold blocks from a number of different logical devices simultaneously.

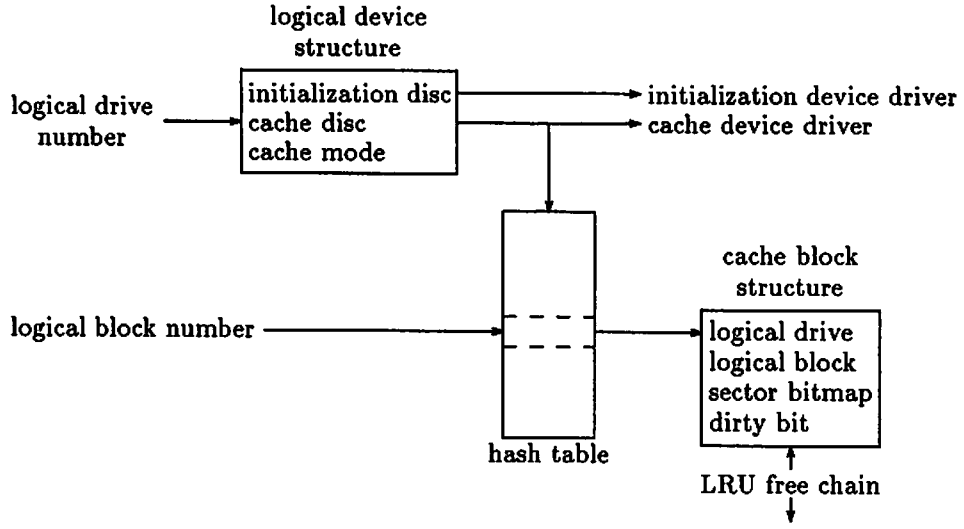


Figure 3.1: DID device driver data structures

The cache device is divided into 8 kilobyte blocks. This size corresponds to the normal unit of file system access on the machines. If smaller transfers are requested, a whole 8 kilobyte block is taken in the cache, but only the requested sectors are read from the initialization device. Each cache block is described by an entry in a table of *cache block structures*, which identify the logical device and block number of the data cached in the block. It also contains a bitmap of sectors, indicating which of the sectors within the block are valid, and a *dirty bit*, which is set if the cached data has been modified, and should not be discarded.

Cache block structures for clean blocks are held on a chain which is maintained in least-recently-used order. If the cache is full, the last block on the chain is discarded whenever a new block is cached. If the DID driver ever finds that all cache blocks are dirty, it will cause a *panic*, printing a diagnostic and rebooting the machine. This event has never occurred in actual use, though it could be caused by major modifications to system files in the cache.

3.2.3 Handling I/O Requests

The DID device driver can split a logical transfer into a number of physical transfers in order to satisfy a request. This is necessary when only part of a logical request has been cached, or when consecutive logical blocks have been cached in different parts of the cache device. Transfers are chained together using a callback mechanism in the UNIX I/O system, which allows an arbitrary kernel routine to be called when an I/O request has completed.

Each logical transfer request received by the device driver is split into four classes of physical transfer, which are performed in the following order:

1. reads from the cache device.
2. reads from the initialization device.
3. writes to the initialization device.
4. writes to the cache device.

Some optimizations are made to reduce the number of physical transfers based on the assumption that device latency is high compared to transfer times. For example, if a read can be only partially satisfied from the cache, and the copy on the initialization disc is up to date, the entire transfer is read from the initialization disc and subsequently written to the cache disc. However, the UNIX buffer cache system ensures that almost all transfers are 8 kilobytes in length, and are aligned on 8 kilobyte boundaries, so the utility of these optimizations is questionable.

The device driver does not attempt to overlap I/O requests on the two devices; doing so would have made the driver more complex. As a result, the observed response time is often greater than that of RVD, since two I/O operations are sometimes required, rather than one. This effect is reduced to some extent by performing some of the additional transfers asynchronously. In particular, when a block is fetched from the file server, the block is placed in the in-memory buffer cache, but it is not written to the cache device. Instead, the block is flagged so that it will be written back to the cache device when it is thrown out of the buffer cache. This has the effect of delaying the write to the cache device until no application programmes are waiting for the transfer to complete. Unfortunately, this technique cannot easily be applied to disc blocks fetched by the demand paging system, so these events always result in two I/O transfers.

The present driver still supports the option of directly mapping cache blocks to logical blocks on a one-to-one basis. That is, disc block number N is always

cached in the Nth block of the cache. This scheme has the advantage of being extremely simple:

- The presence of a valid block on the cache disc is indicated by a single bit.
- Cache block address calculations are trivial.
- Contiguous ranges of cached blocks can be transferred from the cache disc in a single operation.
- Cache block replacement can never occur.
- Dirty bits are not required to mark blocks that have been modified. There is always enough space for all blocks to be present on the cache disc simultaneously.

Unfortunately, this option presupposes that the cache disc is big enough to hold every block on every file system currently mounted, even though only a fraction of those blocks will ever be cached. For this reason, its use is extremely limited.

3.3 Performance

3.3.1 Benchmark

The performance of the DID system was measured with the Andrew benchmark [Howard 88]. This benchmark exercises a file system with a sequence of directory and file operations. The running time of the benchmark and the load on the file server give an indication of the performance of the file system. The benchmark is split into five phases:

1. MakeDir: Make a number of directories.
2. Copy: Copy an existing set of files to another directory.
3. ScanDir: Traverse the new directory, examining the status information of each file.
4. ReadDir: Read every byte of each file.
5. Make: Compile the files.

Both the existing set of files and the copy reside on the file system under test. When Howard *et al* ran the benchmark for the Andrew file system, all of the utilities and temporary files used by the benchmark resided on local file systems. Unfortunately, few DID systems have local file systems; in the results presented here, all utilities and temporary files were on file systems accessed via DID, even when the file system under test was local, or a non-cached RVD. In order to minimize the additional overhead of running the benchmarks in this way, all the

utilities were placed in the caches of the machines under test before the benchmark was run.

The benchmark was run simultaneously on a number of machines, all of which used the same file server. This server contained the original copy of the files copied in phase 2 of the benchmark, as well as the destination directories for each of the clients. Each instance of the benchmark generates one unit of load, which Howard claims is roughly equivalent to the load generated by five users. In fact, the simulated load generates a greater proportion of writes than one would expect from normal users.

All the machines used in the tests, including the file server, were MicroVAX-IIs, each with 5 megabytes of memory. All tests were run three times in a row; means and standard deviations are given for the measurements.

3.3.2 Results

The presentation of these results is based on the style of Howard in his description of the Andrew file system, for ease of comparison. Relative timing figures for Andrew and NFS are included from that paper for the purposes of comparison. Absolute comparisons are misleading because of differences in machine types.

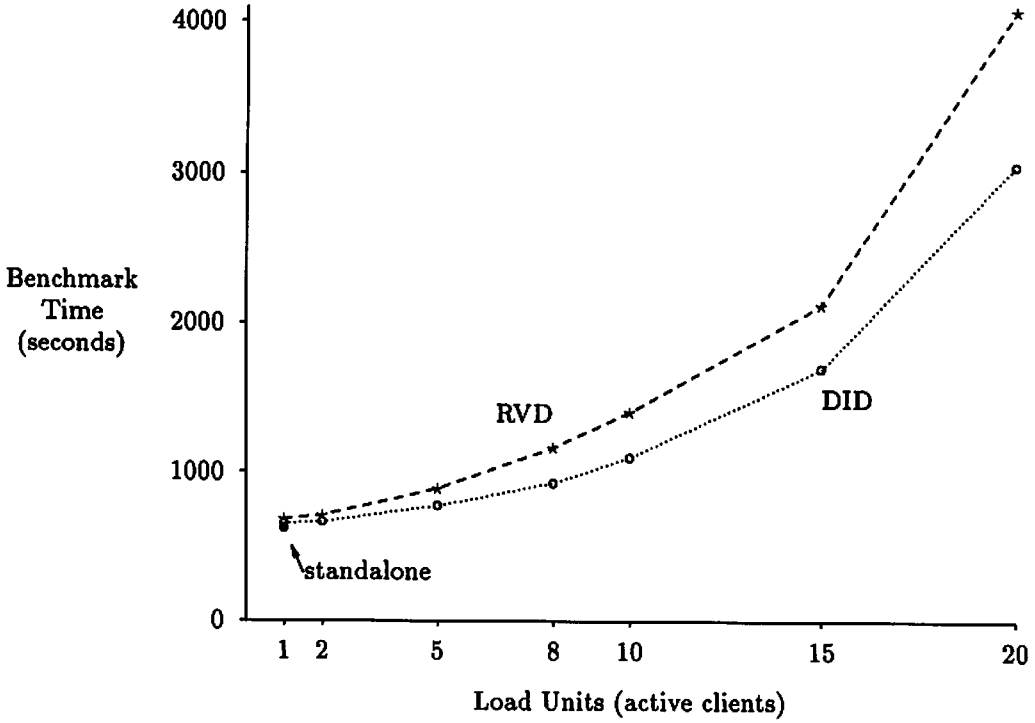
File System	Load	Overall Time	Time for Each Phase				
			MakeDir	Copy	ScanDir	ReadAll	Make
Standalone	1	615 (2)	7 (1)	33 (1)	30 (1)	56 (0)	489 (3)
DID	1	649 (6)	12 (1)	50 (2)	32 (1)	59 (1)	496 (4)
	2	660 (2)	13 (1)	59 (3)	32 (1)	59 (0)	499 (3)
	5	771 (10)	26 (1)	122 (7)	34 (1)	60 (1)	542 (11)
	8	925 (7)	45 (5)	203 (3)	38 (2)	62 (1)	594 (3)
	10	1099 (68)	59 (11)	281 (7)	37 (2)	61 (1)	698 (73)
	15	1696 (33)	87 (3)	543 (55)	37 (2)	63 (2)	997 (49)
	20	3050 (86)	126 (11)	1349 (124)	42 (4)	66 (6)	1507 (95)
RVD	1	680 (4)	7 (2)	34 (2)	33 (1)	56 (2)	550 (7)
	2	706 (7)	10 (0)	54 (1)	33 (4)	62 (4)	551 (3)
	5	888 (7)	26 (2)	122 (1)	36 (2)	90 (8)	624 (6)
	8	1165 (2)	44 (2)	217 (9)	38 (2)	131 (3)	747 (12)
	10	1404 (20)	54 (0)	295 (19)	42 (2)	164 (1)	877 (35)
	15	2124 (61)	85 (2)	563 (45)	52 (4)	251 (2)	1414 (341)
	20	4075 (29)	117 (1)	1773 (81)	69 (2)	346 (7)	1838 (108)

This table shows the elapsed time of the benchmark as a function of load. All timings are in seconds; the figures in parentheses are standard deviations. Part of this data is reproduced in Figure 3.2.

Table 3.1: Running Time of Benchmark for DID

Table 3.1 shows timings for the Andrew benchmark, for various file systems and loads. The standalone file system is the local file system of the machine. DID and RVD are systems accessing an RVD file server, with and without caching respectively. The table shows that DID scales well in phases of the benchmark that involve reading files, but badly when files are created or written. RVD scales badly when files are read or written. At low loads, RVD outperforms DID when

writing files; DID writes data to the file server and the local disc sequentially, which delays the application longer than the RVD case. The total running times for the benchmark are shown again in Figure 3.2.

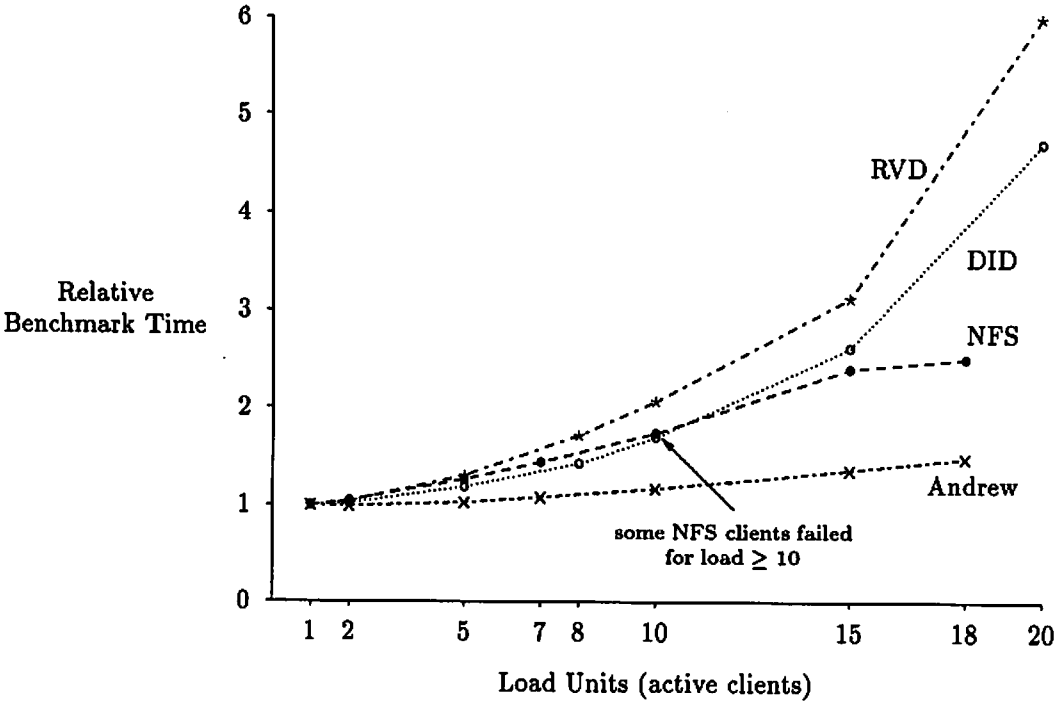


The running time of the benchmark is shown for DID, RVD and the standalone file system.

Figure 3.2: Running Time of Benchmark for DID

Figure 3.3 shows the time taken for the benchmark relative to the time taken at a load of 1. This graph is the best measure for how well the system scales when running the benchmark. Figures for the Andrew file system and NFS are included for reference. The Andrew and NFS clients were Sun-3/50s with 4 megabytes of main memory, of which they used 10% for the buffer cache. The MicroVAXes also use 10% of their memory for the buffer cache, and so had the advantage of larger memory caches. However, the file server used by the Andrew and NFS systems was a Sun-3/160, which is approximately twice as fast as a MicroVAX-II.

This graph shows how badly DID fares overall when compared with other systems. This result was surprising, given the simplicity of the RVD protocol, and the size of the DID cache. Later tests show that the reason for the poor performance is a combination of the write-through policy of the DID cache, the number of blocks touched when a UNIX file is created, and badly tuned network protocols.



The relative running time of the benchmark is shown for DID, RVD NFS and Andrew. All times are relative to the running time at a load of 1. Figures for NFS and Andrew are taken from the measurements of Howard *et al.* [Howard 88]. Note that some NFS machines failed to complete the benchmark at loads of ten and higher. Timings for NFS and Andrew used Sun-3/50 clients and a Sun-3/160 server. The Andrew clients each had 4 megabytes of main memory. Each load unit represents one client running the Andrew benchmark.

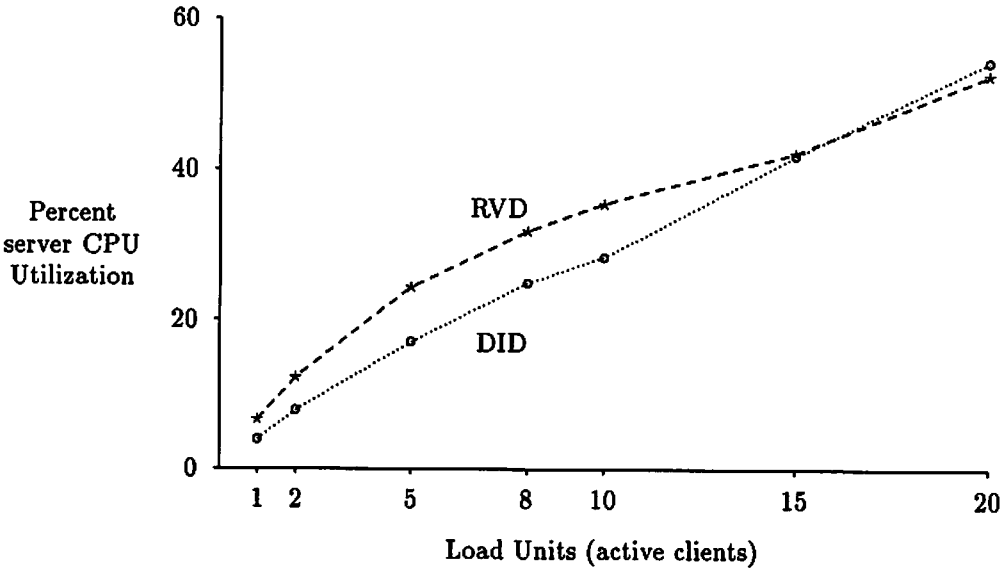
Figure 3.3: Relative Running Time of Benchmark

Table 3.2 shows the server CPU utilization as a function of load for DID and RVD. This information is shown again in Figure 3.4, which shows the reduction in load due to the caching. The figures seem to indicate that the server CPU is not heavily loaded during the test, but they are actually misleading, as a later graph will show.

File System	Server CPU Utilization (%) by Load (active clients)						
	1	2	5	8	10	15	20
DID	4.0 (0.2)	7.9 (0.2)	17.0 (0.3)	24.9 (0.2)	28.4 (0.7)	41.7 (2.3)	54.1 (2.5)
RVD	6.7 (0.1)	12.3 (0.1)	24.4 (0.1)	31.9 (0.1)	35.5 (0.5)	42.1 (2.0)	52.3 (1.1)

This table shows the server CPU utilization during the benchmark for various loads. The figures in parentheses are standard deviations. This data is reproduced in Figure 3.4.

Table 3.2: RVD and DID Server CPU Load during Benchmark



The file server CPU utilization is shown for various load values. Each load unit represents one client running the Andrew benchmark.

Figure 3.4: RVD and DID Server CPU Load During Benchmark

Table 3.3 shows the number of I/O transfers that took place on each client during the benchmarks, showing how the number varies with load. The first two rows show the number of read and write requests made on the DID device driver by the UNIX block I/O system. The subsequent rows show how many I/O transfers were performed to and from the local disc and the file server disc.

One might expect that each machine would require the same number of transfers regardless of the load, but in reality this number varies because of the delayed write action of the UNIX buffer cache. For example, the number of writes increases with load because temporary files exist for longer periods, and are more likely to be written to disc. The number of reads increases slightly with increased load

Transfer Type		Data Transfers by Load							Standard deviation
		1	2	5	8	10	15	20	
request	read	3013	2815	2900	2982	3044	3355	3428	268
	write	2450	2434	2580	2748	2916	3608	4710	75
local	read	2865	2666	2755	2836	2899	3206	3283	268
disc	write	2599	2583	2725	2893	3060	3757	4855	75
file	read	149	149	146	145	145	149	145	6
server	write	888	885	898	918	930	1013	1080	31

This table shows the number of data transfers that took place on each machine during the benchmark as a function of load. The maximum standard deviation for the figures in each row is also given. The table shows the number of read and write requests from the UNIX block I/O system, the number of transfers to and from the local disc, and the number of transfers to and from the file server.

Table 3.3: Number of Data Transfers During Benchmark

because more cache blocks are thrown out of memory by the activities of system housekeeping processes during longer runs.

An interesting feature of the benchmark is the fraction of transfers that are writes. At a load of one, the benchmark generates 45% writes, but this number grows to 66% at a load of 20. Of all requests that go to the file server, 86% are writes at a load of 1, and 88% are writes at a load of 20. The number of writes to the file server is higher than one would expect. This is due to the large number of distinct blocks that must be modified when a file is created. The file's i-node, its directory entry and its data are stored in different blocks. Moreover, when a file is read, i-node blocks are asynchronously written in order to update the last-accessed time.

Transfer Type		Data Transfers		
		Synchronous	Asynchronous	Total
request	read	585700	66673	652373
	write	79606	227788	307394
local	read	560934	64623	625557
disc	write	104374	229838	334212
file	read	24768	2050	26818
server	write	28477	49982	78459

This table shows the number of data transfers that took place on group of 7 machines over several days of normal use. It shows the number of read and write requests made on the DID device driver, the number of transfers to and from the local disc, and the number of transfers to and from the file server. Asynchronous transfers are read-aheads and delayed writes.

Table 3.4: Number of Data Transfers During Normal Use

The transfer profile exhibited by the Andrew benchmark is not typical of that for normal users. Table 3.4 shows data collected from a group of 7 machines over a period of a few days of normal use. Only 32% of all requests were writes, and only 74% of all file server accesses were writes. The DID cache achieves a 4% read

miss ratio in normal use over a long period. The vast majority of writes are to temporary files, and are not written back to the file server.

Table 3.5 shows the occupancy of the DID cache during normal use. Of the 23 machines sampled, none had actually thrown anything out of their 40 megabyte caches. The maximum cache space used was 38 megabytes, but the mean only 21 megabytes. The mean amount of space occupied by dirty blocks was 9 megabytes. These figures indicate that a cache size of 40 megabytes is larger than necessary. A twenty megabyte cache would probably have been adequate in almost all cases.

	Cache occupancy (megabytes)			
value	minimum	maximum	mean	std dev
space in use	6.9	33.8	18.1	7.1
space dirty	2.2	18.8	9.3	4.8
wasted space	1.5	7.5	2.6	1.3
maximum used	7.1	38.3	21.3	9.4

The table shows the minimum, maximum and mean cache occupancy for a sample of 23 machines. The figures for cache space in use, and dirty cache space are instantaneous values. The table also shows the instantaneous value of space wasted due to internal fragmentation of the 8 kilobyte cache buffers, and the maximum cache occupancy since the machine was booted. The machines had been running for an average of 11 days since the last reboot.

Table 3.5: Occupancy of the DID Cache

Table 3.6 and Table 3.7 show file access latency—the time taken to open a file, read a single byte and close the file again. Table 3.6 shows that the time for RVD to access a file on an unloaded server is close to the time to access a local disc. DID takes twice as long to read and cache a file for the first time. The table shows the poor performance of NFS, which is almost as slow as DID when accessing a file for the first time, even though both file server and client are running on faster machines. The large time to retrieve data from the NFS memory cache may be due to additional context switches introduced by the NFS *block I/O daemon* [Sandberg 85]. The Andrew times include the time necessary to switch to the address space of the user-space cache manager. DID and RVD gain significantly from the benefits of the UNIX buffer cache; Andrew and NFS suffer from the problems of context switching to a management process each time a file is accessed.

Table 3.7 shows how the file access latency varies with file size for DID, RVD and Andrew. Howard also gives figures for NFS, but the access times are independent of file size. The times for DID and RVD increase up to the file system block size, and then become constant. The times for Andrew continue to increase, since Andrew always transfers the entire file before proceeding.

The figures obtained from timing the benchmark are quite surprising. DID performed quite badly, despite reasonably good access latency, and low read miss rates in normal use. The write-through cache is liable to generate a large number of writes during the benchmark, but this does not fully explain the observations, which seem quite contrary to normal experience with the system.

System	File cached	Data location	File Access Latency (ms)
Standalone (MicroVAX)	-	local disc	27 (0.2)
	-	local memory	3.2 (0.1)
DID	no	file server	59 (1)
	yes	local disc	27 (0.2)
	yes	local memory	3.2 (0.1)
RVD	no	file server	36 (1)
	yes	local memory	3.2 (0.1)
Standalone (Sun-3/160)	-	local disc	23 (0.5)
	-	local memory	1.7 (0.1)
NFS	no	file server	54 (1)
	yes	local memory	10.3 (0.1)
Andrew	no	file server	160 (35)
	yes	local memory	16 (0.5)

This table shows the file access latency in milliseconds of a standalone system and for DID, RVD, NFS and Andrew with a single client accessing a server. The figures in parentheses are standard deviations.

The files were all 3 bytes long. The latency is the time to open the file, read one byte and close the file. The Andrew times are taken from the paper by Howard *et al* [Howard 88]. The DID and RVD times were measured with MicroVAX-IIs as both client and server. The Andrew figures are for a Sun-3/50 client using a Sun-3/160 server. The NFS times were measured with a Sun-3/160 client and server. The Sun-3/50 is approximately 1.5 times faster than a MicroVAX-II; the Sun-3/160 is approximately twice as fast as a MicroVAX-II. Standalone times from both machine types are given for comparison.

Disc access times are variable depending on head positioning. Tests were performed by opening 1000 files in the same directory. The UNIX file system will ensure that the files are stored close to one another, so the times are optimistic.

Table 3.6: File Access Latency of DID

File Size (bytes)	File Access Latency (ms)		
	DID	RVD	Andrew
3	59 (1)	36 (1)	160 (35)
1113	75 (1)	47 (1)	148 (18)
4334	121 (5)	64 (1)	203 (29)
10278	153 (4)	81 (1)	310 (54)
24576	154 (2)	82 (1)	515 (142)

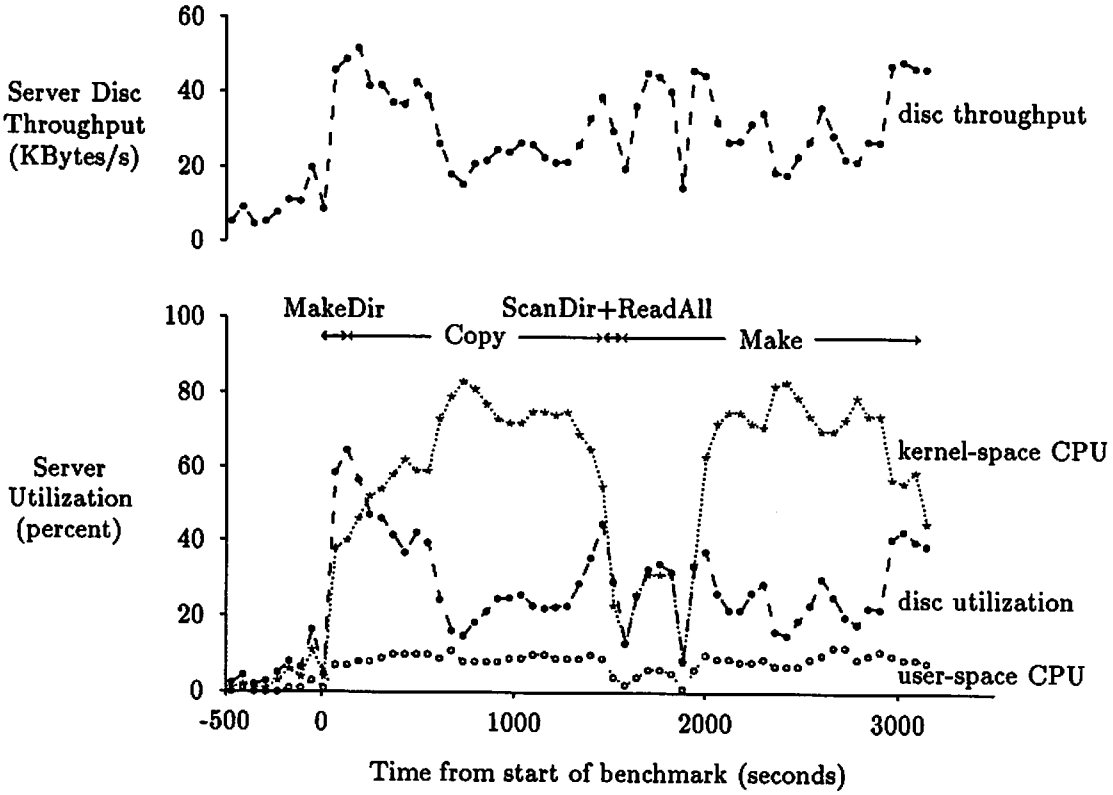
This table shows the latency of file access in milliseconds as a function of file size for DID, RVD and Andrew for files that have not yet been cached. Figures in parentheses are standard deviations. The test conditions are the same as in Table 3.6. The DID and RVD transfer block size was 8 kilobytes; no more than 8 kilobytes will be transferred regardless of file size. Andrew always transfers the whole file.

Table 3.7: File Access Latency by File Size for DID

In order to discover the cause of the problem, the server was monitored while 20 clients ran the benchmark to determine the following:

- disc throughput in kilobytes per second (upper graph);
- disc utilization (fraction of the time spent seeking or transferring data);
- kernel-space CPU utilization;
- user-space CPU utilization.

The results are given in Figure 3.5.



The upper graph shows the variation of server disc throughput during one run of the benchmark with 20 clients running DID. The lower graph shows the variation of server disc utilization and CPU utilization for the same run. Two curves are plotted for CPU utilization; one shows the time spent in user-space, the other shows time spent in kernel-space. All values are averaged over 60 second intervals.

Figure 3.5: Server throughput, disc and CPU utilization during benchmark

The user-space CPU utilization is low throughout the period of the benchmark, even though the RVD server runs entirely in user-space. The disc throughput never exceeds 50 kilobytes a second, even though the disc utilization reaches nearly 70% at that time. This indicates that the disc is seeking a great deal. The disc throughput and utilization curves follow one another closely, showing that the fraction of transfers requiring disc seeks is uniform across the benchmark. The drop in disc utilization in the middle of the Copy phase is quite unexpected.

The most interesting feature of the graph is the relationship between the kernel-space CPU utilization and the disc utilization. The two curves are almost perfectly symmetrical about the 50% utilization line whenever the CPU utilization exceeds this level. This result was so surprising that the disc utilization figures were considered faulty, until their correspondence with throughput figures was observed. The unexpected drop in disc throughput during the Copy phase can now be explained. The clients are trying to write data at an enormous rate, and the file server is swamped with large data packets. The server is unable to keep up with the clients, and does not acknowledge requests before replying, so clients send their requests repeatedly. Investigation showed that the server receives about 150 requests a second, 90% of which are retransmissions of previously received packets. Few packets are lost, but clients take no account of server load when deciding when to retransmit. The networking code in the UNIX kernel on the server machine is quite inefficient, and consumes between 70% and 80% of the processor when receiving 1 kilobyte packets at this rate. In order to verify these results, a simple adaptive client retry strategy has been implemented. The number of retransmissions was reduced to 10% of all packets arriving at the server, and disc throughput was increased to 60 kilobytes per second.

All the curves show noticeable drops in activity just after the Copy phase, and at the beginning of the Make phase. The first dip is due to the ScanDir and ReadAll phases, which perform no local accesses. The second dip is due to the compilation of one extremely large module, which keeps all the clients CPU bound for over a minute. The three peaks in kernel-space CPU utilization correspond to the benchmark writing two large library files and a linked object file.

3.4 Experience

DID has been in service for two years, supporting a growing user population from many groups in the department. The number of machines using it has grown to 25, and is slowly increasing. The average number of active users varies between 20 and 60 during working hours. There are over 400 separate RVDs, of which less than 100 are normally mounted. There are three main file servers holding user and system files, though only one of them actually serves system files at any given time.

All the machines have local discs, but very few of them use the discs for permanent file storage. On most machines, the local disc is used only for caching and swapping. Each machine has only one cache area, usually about 40 megabytes in size, which is used to cache every RVD currently mounted on the machine. It was found that using multiple cache areas served no useful purpose, and merely complicated the allocation of local disc space.

Machines are booted over the network, and access all of their files across the network. Even the boot servers and file servers, though able to boot themselves independently, switch to the network version of the file system during their boot sequences to ensure that all machines have identical views of the system directories.

3.4.1 Cache Location

The access time of a disc-based cache is typically tens of milliseconds, which may be longer than the time needed to fetch the data from a fast file server. Given the high latency of disc storage, it may be better to use a smaller memory-based cache rather than a disc-based cache. Other file system designs, such as AT&T's RFS [Bach 87], cache remote files in main memory, taking no more than a few megabytes of cache area. The results of Nelson [Nelson 88], indicate that only a relatively small memory-based cache is sufficient to achieve good performance in a small distributed system.

In fact, DID already benefits from some main memory caching due to the UNIX buffer cache. The idea behind using a large disc cache is to minimize load on the file server, rather than to satisfy client requests quickly. From this point of view the location of the cache is immaterial. If large amounts of main memory were available, it would certainly have been used in preference to disc storage. Fortunately, the design of DID allows for any cache device, including a RAM disc, to be used in place of the local disc. This functionality could be provided more efficiently, though less flexibly, by modifying the UNIX buffer cache to support additional write-back policies. Braunstein has investigated file cache management schemes for machines with large memories [Braunstein 88]. He found that current buffer cache algorithms do not perform well when the size of the cache is increased to several tens of megabytes, and developed new algorithms that improve performance considerably.

One particular advantage of a disc-based cache that is not exploited in DID is that of caching data across client crashes. A fuller discussion of this topic can be found in Section 4.5.5.

3.4.2 General Performance

The observed performance of DID has been very encouraging. One file server is able to provide all of the files needed by 25 machines, while using about 5% of the CPU of the file server. During peak periods of load, the CPU utilization may rise to as much as 10%, but this is rare. This can be compared with previous experience with the unmodified RVD system, which normally required over 30% of the CPU on each file server to satisfy only 8 client machines. This was despite the fact that all system utilities and libraries were held on clients' local discs, rather than on the file server.

Later optimizations have improved the performance of the RVD system used in the Athena Project [Treese 88], but these improvements have not yet been incorporated into the DID system.

Most of the time, the performance of DID is not significantly different from that of a local file system. However, the difference is clear under certain conditions:

- repeated synchronous writes;
- recovery after power failure;

- logging in.

DID always flushes synchronous writes to the file server, which performs no delayed writing or optimizations. UNIX generates synchronous writes whenever directories or other structural information is updated, and when the buffer cache becomes full. Therefore, DID can be slow when called upon to write files that are much larger than the buffer cache, or when creating and deleting a large number of files at once. The most obvious solution to these problems is a greater use of delayed writes, but in the case of DID, this would require considerable changes to the UNIX kernel. The next chapter describes a file system that makes heavy use of delayed writes.

After a power failure every machine in the system reboots, and reads a large number of system files from the same file server. The current RVD server has no in-memory disc block cache, so each request generates a disc access[1]. This approach is acceptable in normal use, because the combined caching potential of the clients is huge compared to the total memory of the file server. However, after a power failure the effect on the server is quite dramatic; for approximately 15 minutes after the power has been restored, the system is unusably slow. As machines complete their boot sequences, the system becomes faster, and within a couple of minutes, the previous level of performance has been restored. This effect could be reduced by having the server cache the blocks of system files, or by using multiple servers for the system file space.

In the processor bank, users' home directories are volatile objects that reside only in the caches of the machines being used. Users mount and use their own cached partitions for the long term storage of files. As a result, the login procedure must create home directories and initialize individual setup files for users whenever they log in to a newly booted machine. This procedure often involves creating several files per user, and is noticeably slower than a normal login, but the delay is bearable. The real problem arises when large numbers of users log in at once, as happens when the machines are used for teaching classes. This problem is best solved by eliminating the restriction that files cannot be writably shared across machines. This too is addressed by the file system described in the next chapter.

3.4.3 Crash Recovery

Server crashes require almost no action for recovery. When the server restarts, it is contacted by its previous clients, which reconnect and start using their virtual discs as though the server had never crashed. There is no possibility of losing data due to a server crash, since it does not use delayed writes to disc.

Client crashes can cause virtual discs to be left in an inconsistent state, just as in the case of standalone UNIX file systems. This is partly due to the choice of the disc block as the unit of naming and transfer in the client-server protocol. If the unit of transfer were some part of a file, rather than a disc block, servers

[1]The RVD server deliberately bypasses the UNIX buffer cache on the server in order to avoid the problems of losing modified data during a crash

could be responsible for the internal consistency of their own discs. Fortunately, the problems caused by this are not insurmountable.

It is normal to run the programme *fsck* [Kowalski 78] when booting a machine after a crash to check for and correct file system inconsistencies. Since the system file space is served read-only, it cannot change, and therefore never needs checking. This greatly improves the startup time of clients, which can be booted in about a minute if the appropriate servers are not overloaded. A standalone MicroVAX system requires almost three minutes to boot after a crash. A flag was added to the start of each file system to indicate when *fsck* need not be run. The flag is set when the file system is cleanly dismounted, and reset whenever it is mounted.[2] Since only a small number of the RVDs are mounted at a given time, most of the file systems do not require checking after a crash. Moreover, since most RVDs are only a few megabytes in size, each can be checked in a few seconds when it is next mounted.

3.4.4 Lack of Sharing

The inability to share writable files with RVD is a great disadvantage and was expected to cause many problems. In fact, although users have encountered problems, few of them caused serious inconvenience. Undoubtedly, users would behave in a different way if sharing were more convenient, but very little sharing seems essential for effective use of the system. In only two or three cases are users forced to share machines so that they can share a single writable file space.

A more serious problem is in updating file spaces that are always mounted on many machines, such as the system file space. In these cases, update is normally accomplished by writing a new version of the file space to a separate RVD, and then swapping the names of the RVDs. Clients remain connected to the original copy, but pick up the new copy when they reboot or remount the file system. Most virtual discs can be dismounted and remounted without rebooting, but when system utilities are updated, every machine must be rebooted. It is not necessary to reboot all machines together; usually it is done over a period of one or two days. Important changes can be made to the caches of machines, either by hand, or automatically at reboot.

System files are clearly difficult to update under this scheme, but even this has its advantages. The university environment inevitably invites some amount of tampering and abuse of the machines. It is comforting to know that users can only change cached copies of system utilities and that unauthorized modifications are undone when the machine next reboots.

3.4.5 Backup

The issue of backup, which is important in all file systems, is simplified by DID's use of many small file spaces which can be backed up individually. Since each virtual disc is small, it can be copied in a relatively short time, so backups can

[2]This technique is used in several variants of UNIX, and has since been added to ULTRIX.

occur even during the middle of the day without availability dropping significantly. Only one or two RVD partitions are unavailable at any time, and they are almost always released within a few seconds.

Manual management of tapes and disc partitions would be a dreadful task for a human operator. The most important features of the RVD backup system are that tape allocation is automatic, and that it needs human intervention to change tapes only once per day. The backup system currently writes about 180 megabytes of data to tape each day, writing mostly incremental changes from previous dumps, and guaranteeing to backup all new data each day. It has over twenty 90 megabyte tapes under its control.

One problem of the RVD backup system is that it cannot access an RVD while it is mounted for writing by a client machine. This becomes dangerous if a user habitually leaves RVDs mounted for very long periods. If the backup system is unable to access an RVD after repeated attempts over a number of hours, it sends an electronic mail message to warn the owner of the disc that the backup has been missed. To alleviate this problem, RVDs are automatically dismounted whenever the RVD is no longer being accessed by any processes, and all of the processes belonging to the owner of the RVD have finished.

Ideally, we would like to backup a partition while it is in use, but it is difficult to ensure that the file system structures are in a consistent state in a system based on disc block access. A heavy-handed approach would be to take a copy of the disc during a period of low activity. If this copy proves to have only minor inconsistencies that can be corrected without losing data, a backup would be taken from the copy. If the number of inconsistencies was too great, the copying process could be repeated. Experience with the state of UNIX file systems after crashes indicates that this approach is likely to work in almost all cases, but it cannot be recommended for subtlety.

The Andrew file system also makes use of fairly small *volumes* to reduce the complexity of operations such as backup and storage allocation [Sidebotham 86]. Backup of volumes is accomplished by making a read-only replica, from which the backup is taken. The internal consistency of the replica can be guaranteed by the file server, since it alone controls the disc data structures.

3.5 Summary

The DID caching system is a simple, effective scheme for improving the performance of accesses to remote discs. By using large caches held on the local discs of client machines, it achieves performance close to that of standalone systems under normal conditions, while allowing many more machines to use a single file server than was previously possible. Other notable features are:

- large caches (approximately 40 megabytes), which give a read miss rate below 5% in normal use;
- cache held on the client local disc;

- all system utilities and libraries served to 25 machines by a single file server;
- small user file spaces which reduce the problems of space allocation and backup;
- improved scalability;
- remote files are accessed via the standard UNIX buffer cache.

Some disadvantages stem from DID's use of disc blocks, rather than files as the unit of transfer:

- Write-sharing of files is impossible.
- Client crashes can cause file system inconsistencies.
- Copies of active file systems cannot be taken for backups easily.

Performance problems under heavy load are caused by a combination of effects:

- The cache write-through policy cannot spread bursts of updates over a longer period.
- Clients are delayed during synchronous writing of file system metadata.
- A large number of separate disc writes are generated for each newly created file.
- Reading a file generates disc write traffic to update the last-accessed time.
- Clients use inappropriate retry policies when writing to congested servers.

Chapter 4

A Caching File System

This chapter describes the design and implementation of a prototype file system that makes heavy use of caching to achieve its performance. The emphases are on caching as much data as possible, minimizing communication with centralized servers, and on preserving fine-grain sharing semantics. Delayed writes are used to reduce the amount of data to be written to the file server.

The following sections describe the assumptions, the basic algorithms employed, how they have been implemented in MFS (Mike's File System), and some features that could be added to MFS to improve it. The behaviour and performance of the prototype are presented in Chapter 5.

4.1 Assumptions

The design of MFS assumes various properties of client machines and their workload. In the light of the observations of Chapter 2, the workload is assumed to consist mainly of access to small files, which are not typically write-shared. However, some write-sharing is assumed to arise from distributed compilations, global log files, and *ad hoc* databases used by applications such as mail systems and line printer systems.

The key assumptions about clients are:

- Each client has a reasonable amount of memory or disc storage that may be used as a cache. This will be at least a few megabytes, but may be a few tens of megabytes.

- Clients are not considered trustworthy—users may have total control over their workstations.

- The number of clients is large compared with the number of servers.

Another important assumption is that users want high performance wherever possible, and prefer remote file access to resemble local file access in all respects, except perhaps failure semantics.

4.2 Caching Algorithm

The cache consistency algorithm used in MFS is very similar to various *directory* schemes suggested for maintaining memory consistency on shared memory multiprocessor systems [Tang 76] [Censier 78]. These protocols use a single directory to record which caches contain which data and whether the cached data can be modified. When a cache misses, a check is made against the directory to see whether the data is in another cache, and whether it has been or can be modified in that cache. If sharing is about to take place in a way that would cause cache inconsistency, the other cache(s) are contacted and forced to flush the dirty data, or invalidate the cache entries, as appropriate.

These protocols have not become popular with designers of shared memory multiprocessor caches, for a number of reasons:

- Centralized directory controllers are potential bottlenecks.
- A significant amount of memory must be added to distribute the directory to avoid bottlenecks.
- These protocols do not make use of the bus' ability to broadcast.

Broadcast can be used merely to invalidate other caches [Archibald 84], so reducing the total state required for the directory schemes, but the most popular approach is the snoopy cache [Archibald 86]. Snooping allows each cache to observe the behaviour of the other caches on the bus, permitting them to maintain consistency without additional messages or centralized directories. This technique requires storage proportional to the size of the caches, and has no central bottleneck other than the bus itself.

It seems that snoopy caches have an advantage in systems where every message passes every cache controller, since they make good use of the bandwidth taken up on the bus. Unfortunately, snoopy cache protocols require cheap, reliable broadcast for their operation. The broadcast primitives provided by most local area networks are not sufficiently reliable, and excessive use of broadcast can place an unwelcome load on all machines in a large network [Treese 88]. The overhead of examining file caches on client machines is high enough that snoopy cache techniques would probably degrade performance in most situations as most of the messages received by a client would have nothing to do with that client's cache contents. Unless file servers are highly replicated, there is little additional overhead in adding a centralized cache directory since all requests must come through a file server in any case. In addition, while the size of a directory is significant in a shared memory system, it is very small when compared to the size of a typical file system.

Given these observations, a directory technique for cache consistency seemed reasonable for a distributed file system, and so a very simple directory scheme was chosen for MFS. The algorithm can be viewed as multiple-reader, single-writer lock algorithm, with callbacks to request the release of conflicting locks. In order to avoid confusion with higher level file locking primitives, I will refer to these locks

as tokens. Assuming that an abstract *data item* is protected by a single token, the basic rules are as follows:

- For each data item, a centralized server is responsible for issuing and reclaiming read and write tokens.
- For each data item at a given instant, there may be
  - no clients holding tokens, or
  - exactly one client holding a write token, or
  - one or more clients holding a read token.
- The server maintains a list of exactly which clients hold which tokens.
- Clients may request tokens or release tokens by contacting the server.
- When a token is requested by a client, the server does not issue the token until it has contacted all clients with conflicting tokens and these tokens have been reclaimed.
- While a client is holding a read token for a data item, it is permitted to read a data item in its cache.
- While a client is holding a write token for a data item, it is permitted to read or write a data item in its cache.
- Data items may not be accessed unless the client holds an appropriate token.
- Before a client releases a write token, it must write the data item back to the server if it has been modified in its cache.

At most one client may have any data item cached for writing at any given time, and no client may have the data item cached for reading if it is concurrently cached for writing elsewhere. This is a sufficient condition for cache consistency, provided clients do not access data without holding the correct tokens. In fact, provided that no client is allowed to write a data item without holding a write token, some (faulty or malicious) clients can ignore the need to hold tokens for reading without harm to other clients.

The algorithm performs best when there is little conflict. When there is no conflict, each client will obtain tokens for the data that it uses, and no tokens need ever be reclaimed by the server. Since the evidence indicates that write-sharing is quite rare in file systems, the algorithm could be expected to work well.

4.3 Overview

Each client machine has a file cache, which is actually part of the local file system of the client machine. Cached files are placed in the normal directory structure as though they were local files. The cache manager for each machine is a privileged user-space process that manipulates the cache via privileged kernel calls. Applications proceed as though they were using a purely local file system. When an application system call requires some cache activity, the kernel suspends the application and sends a message to the cache manager via a private interface. The cache manager performs whatever cache operations are necessary, and then tells the kernel to continue the application programme.

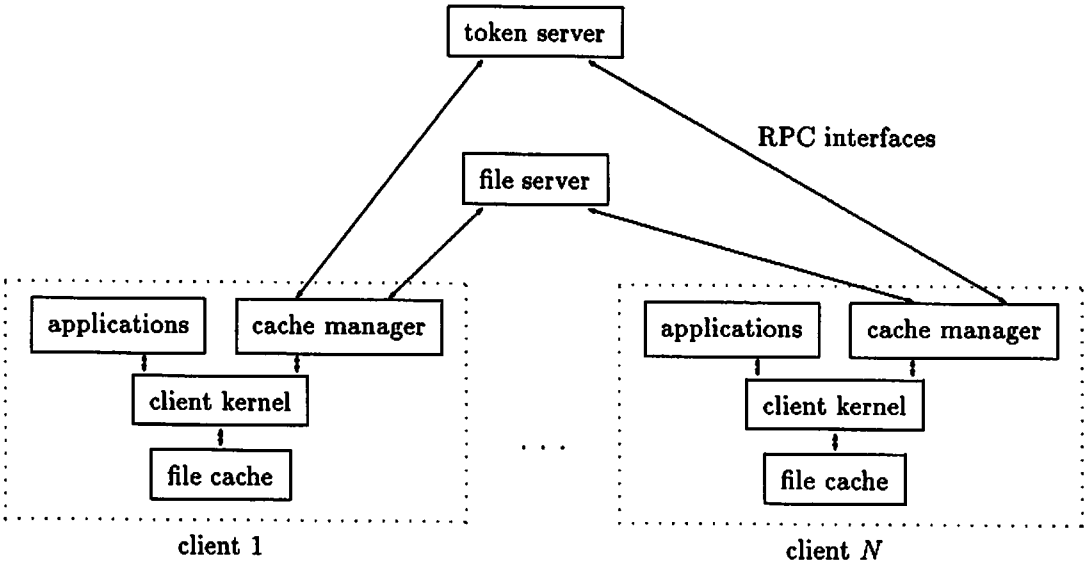


Figure 4.1: The structure of MFS

The cache manager contacts a file server whenever it needs to read and write files on behalf of an application. The token server is responsible for maintaining the set of tokens issued to clients, and to detect conflicts between the requests from the clients. In MFS, it is implemented as a separate server, and is called directly by the client cache manager. The token server calls the cache manager to reclaim tokens when token conflicts arise. Figure 4.1 shows the overall structure of MFS; the design of each of these components is discussed in later sections.

MFS was built using MicroVAX-II clients running ULTRIX, DEC's version of the UNIX operating system. Each client has a large cache held on a section of disc, usually around 20 megabytes in size. The token server runs on another UNIX machine, typically another MicroVAX. Any machine supporting Sun Microsystems' Network File System (NFS) [SUN 86a] can be used as a file server; experiments were done using an unmodified Sun-3/160 file server.

Sun Microsystems' Remote Procedure Call system [SUN 86b] was used for all inter-machine communication in MFS. The RPC protocol sits upon a datagram

protocol running on a 10 megabit/s Ethernet. Unfortunately, the current version of the RPC system, version 3.9, lacks several features necessary for practical distributed applications. The features that have been added in the implementation of MFS are:

- multiple client and server threads;
- detection of server failure during a long lived procedure call.

In addition, the RPC protocol does not provide at-most-once semantics, so all remote procedures had to be made idempotent.

4.4 The Token Server

The token server is responsible for maintaining the list of tokens held by each client. It issues new tokens on request, and reclaims tokens from clients when conflicts arise.

4.4.1 Separate Token Server

Ideally, the token server would be integrated with the file server. An integrated design has several advantages:

- The file server can efficiently enforce the rule that a client may not write files without possessing appropriate write tokens. This allows the server to prevent cache inconsistencies even in the face of malicious clients. If the file server and the token server are separated, extra messages must be sent to enforce the rule.
- The token server can check efficiently that clients have permission to read or write a file when they request tokens for it. This check is desirable in order to prevent a malicious client obtaining write tokens for various important files, simply to inconvenience other clients. Once again, this can be achieved by sending extra messages.
- Token requests can be piggybacked on file server read and write requests. This significantly reduces the number of messages sent by the clients.
- File system availability is increased, since only one machine needs to be running instead of two. (This argument is less powerful in systems where servers are replicated.)
- The system failure model is simplified—it becomes quite unlikely that one of the two services is available without the other.

In spite of this, the MFS token server is separate from the file server. Implementation constraints made it difficult to modify existing file servers, and a prototype system was more easily built from existing components. In fact, most

of the problems of separating the token server from the file server can be overcome efficiently:

- An application cannot distinguish between an update made without a valid write token, and a set of consistent writes interleaved with the application's read system calls. In MFS, clients do not guarantee to hold tokens for longer than a single system call, and applications cannot gain any information about tokens, so the problem of inconsistent writes is unimportant.[1]
- It is not necessary for the token server to check file permissions for every token granted. Needham and Burrows point out [Needham 88] that it is sufficient for the token server to check permissions only when conflict arises. At all other times, the token server can grant tokens to malicious clients, since they can neither access the files, nor hold onto them to prevent a legitimate client from obtaining a token. This is considerably more efficient in situations where token conflict is rare.
- Although coded as a separate server, it is often possible to run the token server and the file server processes on the same physical machine. This reduces the probability that one server will crash independently of the other.
- A separate token server can be used with many existing file servers, merely by changing the file access checking stub used when resolving lock conflicts.

Of these arguments, the last was the most important for this implementation of MFS. A considerable amount of time was saved by using existing NFS file servers. However, the resulting system has a performance disadvantage over an implementation with a combined file and token service, because clients must communicate with each server separately.

The permissions-checking code to prevent malicious clients from acquiring tokens for arbitrary files has not been implemented in the current version of MFS. Instead, there is a limit on the length of time a client can hold a token after it has been requested by the token server. A client can extend this time beyond the normal RPC timeout period while it writes modified data back to the file server, but the period cannot be extended indefinitely. This technique increases the likelihood of forward progress, even in the presence of malice, albeit somewhat slower. Logging the identities of clients that repeatedly experience conflict is likely to be an effective means of detecting faulty clients in most situations. More demanding environments will require that tokens be access checked.

4.4.2 Token Server Interface

The MFS token server provides client machines with the abstraction of a *token group*, which contains a set of tokens that can be independently issued and reclaimed by the server. A token group is named by a 288 bit identifier, and can

[1] In fact, applications could distinguish these two cases by repeatedly examining both the data and the file modification times on two separate clients, but such behaviour is extremely unlikely.

protect up to 2^{32} data items independently. The intention is that one token group should be used for each file, and each byte of the file should be protected by a separate token within the group. The token group identifier is large enough to contain an NFS file handle (256 bits) and a file server network address (32 bits) to make it unique across all systems. This identifier could be shortened considerably by making use of redundancy in the internal structure of the NFS file handle.

The interface allows a range of tokens within a single token group to be obtained or released in a single, atomic operation. This allows clients to obtain blocks of tokens in the same way as they obtain blocks of bytes from the file server. The token server does not impose any particular block size since the optimal block size may vary from file server to file server, or from client to client. Since the token server treats each of the 2^{32} tokens within a token group as independent, clients using different block sizes can interwork without fear of inconsistency, provided they agree on what each token protects. There is no provision for atomically obtaining tokens from more than one token group, or atomically obtaining more than one range within the token group.

The token server exports only one remote procedure call:

```
(status, session_id) := TOKEN (token_group_id, token_type,
                               token_range, client_id)
```

The `token_group_id` identifies the token group from which tokens are obtained. The `token_type` specifies whether the tokens should be READ_TOKENs or WRITE_TOKENs. The special type VOID_TOKEN is used to discard tokens. Two 32 bit unsigned integers make up `token_range`, which specifies the range of tokens to be affected within the group. The `client_id` identifies the client making the call—this is just a callback address for the server to use when requesting the client to give up the token.

The call returns a status code, which is zero if there has been no error. The only errors possible are RPC system errors and communication errors, such as timeouts. That is, the token server *always* grants a token eventually unless some system failure prevents it from doing so. Even if the token server implemented the access control checks described above, the server would always respond positively in order to prevent malicious clients from obtaining information about other clients' file access patterns [Needham 88]. The server ignores conflicts with tokens held by the same client on the grounds that the client can manage its own tokens without assistance. This behaviour causes the call to be idempotent, which simplifies the requirements of the RPC protocol. The `session_id` is also returned. This is a 32 bit identifier which changes only when the server fails to contact the client. This identifier allows a client to detect when it has been out of contact with a server. When this happens, it should refresh any tokens that the server may have released.

The server expects each client to export one procedure call that is used to reclaim tokens when a conflict is detected:

```
(status, remaining_tokens) := RELEASE_TOKEN (token_group_id,
                              token_type, token_range)
```

The server passes a `token_group_id`, `token_type` and `token_range`, and expects the client to release any tokens it holds that conflict with the request. The client replies with a status code and a list of ranges of tokens that it still holds in `token_range`, after it has removed conflicting tokens. This allows the client to degrade tokens from write to read to avoid conflicts, rather than removing all tokens in the range. The server checks that all conflicts have been removed, and that no new tokens have been claimed by the client. If the client cannot be contacted, all its tokens are revoked. If the client claims tokens that it did not previously hold or fails to resolve the conflicts, all the tokens held by the client in `token_range` are revoked.

No attempt has been made to make this interface secure against a determined attack. In a secure implementation, it would be essential to encrypt all messages so that one client could not impersonate another. Of course, the file server would be protected by its own access checks, but malicious or faulty clients could convince the token server to give other clients an inconsistent view of the file system, even though the file server still held correct data.

4.4.3 Data Structures and Algorithms

The data structures used to hold the information for a token group are extremely important, since the storage requirements of the token server and clients must not be excessive, even when the server has issued a large number of tokens.

The basic requirements of the data structure are as follows:

- Each token group can protect 2^{32} data items independently.
- Tokens are usually obtained in large, contiguous ranges.
- Many tens of clients should be able to hold tokens from the same token group simultaneously.
- Token group identifiers are very sparse, and only a few tens of thousands are likely to be in use at any given time.
- Token groups will be created and deleted as files are created and deleted, so both these operations must be reasonably efficient.

The data structures used in the MFS token server are shown in Figure 4.2. A hash table maps token group identifiers to *token group structures*. Each token group structure contains a single mutual exclusion lock, and a pointer to a variable length array of *holder structures*, one for each client holding any tokens in this token group. Each holder structure contains a pointer to a *client structure* and pointer to a *token array* which holds the state of each of the 2^{32} possible tokens in a compressed form.

There is only one client structure for each client known to the token server. This structure contains the client's name, its network address, its current session identifier, and a flag indicating whether the client is up or down. When the

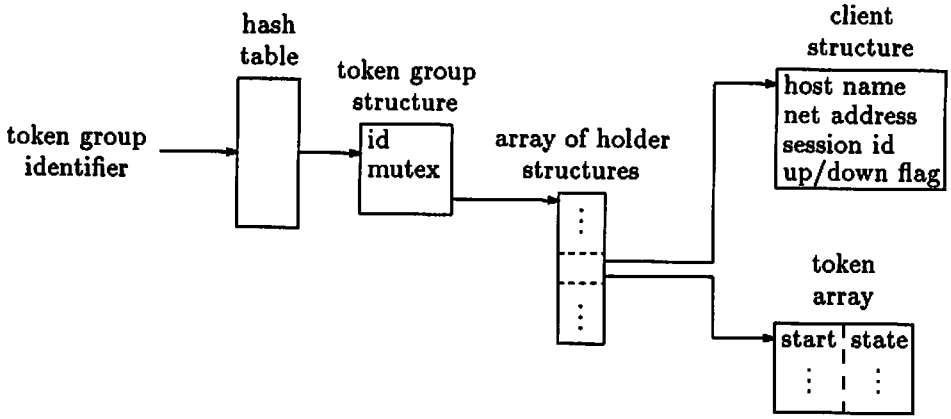


Figure 4.2: Token server data structures

server fails to contact a crashed client when reclaiming a token, it marks the client as down in the client structure. This has the effect of immediately revoking all tokens owned by that client—the data structures themselves are garbage collected asynchronously. When the client contacts the server once again, a new client structure is created, with a new session identifier.[2] This scheme ensures that the server is never delayed more than once for a crashed client, and that a client will lose no tokens by being temporarily inaccessible if there are no conflicting token requests.

An alternative scheme would be for the client structure to hold a timestamp giving the last time that the client contacted the server. Clients would poll the server periodically to ensure that the timestamp was never very old. The server would behave as before, except that it would revoke the client's tokens immediately if the client had not contacted the server for more than one RPC timeout. This improves the service seen by a client, since it is less likely to encounter a delay when a token must be reclaimed from a crashed client. Clients must periodically refresh their sessions to achieve this effect, but the cost of this is small.

The token array provides an efficient representation for the tokens in a token group. Tokens will usually be allocated in ranges rather than independently, so the state of the tokens is held as an ordered list of the start points of ranges. The possible states for a token are VOID, READ and WRITE. The VOID state indicates that no token has been issued. For example, if a client holds WRITE tokens for the first 8 kilobytes of a file and READ tokens for the third 8 kilobytes, the array would have four entries:

```
    0   WRITE
 8192   VOID
16384   READ
24576   VOID
```

[2]Old client structures may still exist for the client if they have not yet been garbage collected, but they are ignored by the routines that traverse the server data structures.

The last range is assumed to extend to $2^{32}-1$. As well as being fairly compact, this data structure is easy to manipulate, and can be searched rapidly. Three basic operations are used:

1. `SETRANGE (token_array, start, end, state)`

   Set all tokens in `token_array` from `start` to (end-1) to `state`. This can be accomplished by inserting array entries for the `start` and `end` points, removing all entries between the two, and possibly removing one redundant entry from each end of the range.

2. `ranges_and_states := CONFLICTS (token_array, start, end, state)`

   Return a list of ranges in `token_array` between `start` and (end-1) that have states conflicting with `state`. `WRITE` is defined to conflict with `READ` and `WRITE`. `READ` conflicts only with `WRITE`.

3. `got_tokens := GOTTOKEN (token_array, start, end, state)`

   Return `true` if all points in `token_array` in the range `start` to (end-1) have a state higher than or equal to `state`. `VOID` is defined to be the lowest state and `WRITE` the highest.

All operations on token arrays are built from these primitive operations, each of which can be implemented with reasonable efficiency in just a few tens of lines of code. An alternative data structure would have been a linked list of start positions of regions. This would have made insertion and deletion more efficient at the cost of increased search time. Fortunately, long lists have been sufficiently rare to make this consideration unimportant.

A TOKEN call

When the token server receives a new TOKEN call, it behaves as follows:

- Find the correct token group structure by hash table lookup. If the token group structure does not exist, create it.
- For each holder structure in the token group:
  - If the associated client structure is flagged as down, remove the holder structure from the array and ignore it.
  - If the holder is for the current client, stamp the new token range into the token array.
  - If the holder is another client that holds conflicting tokens, fork a new thread to reclaim those tokens with a RELEASE_TOKEN call.
  - If a client does not respond to a RELEASE_TOKEN call, flag it as down in the client structure.
  - If a client responds but does not release all conflicting tokens correctly, or in a reasonable time, treat the tokens as released anyway.

- Wait for all forked threads reclaiming tokens to terminate.
- If no holder structure is found for the current client, create one, and insert the new token range in the token array.
- If no (non-deleted) client structure exists for the client, create one and initialize it with a new session identifier.
- Return the session identifier to the calling client.

In fact, a small optimization could be made to this algorithm, though this has not been done in the current implementation. If another client is found to hold write tokens for the requested range, then no other clients need be considered, since there can be no other outstanding tokens for that range. Similarly, if read tokens have been requested, and another client is found to hold read tokens for the same data, no further clients need be considered. This would be beneficial in situations where a large number of read tokens had been issued for a single file, as might be the case for standard system files.

As multiple threads are accessing the server data structures, locks are needed to serialize modifications. Most of these locks need only be held for short periods of time, while a structure is being modified or examined. However, since clients can delay releasing their tokens by several seconds, the token arrays will be held locked for significant periods. In the current implementation, a single mutual exclusion lock is used for all the data structures associated with a token group, causing non-conflicting token requests in the same group to be needlessly serialized. More concurrency could be obtained by placing a mutual exclusion lock on each token in the group. This could be done efficiently by using a data structure identical to the token array itself to protect sections of the token array. This lock array would itself be protected by a single mutual exclusion lock, but this would be held only for short periods of time.

Since client machines cannot be relied upon to relinquish tokens that they no longer require, algorithms are needed to prevent the excessive growth of the token server's database. The main mechanism for releasing unwanted tokens has already been mentioned; all tokens held by a client are revoked if it fails to respond to a RELEASE_TOKEN call. If clients remain contactable for long periods, a server thread generates fictitious token conflicts with tokens that have been held for a long time. This crude approach causes some additional client load if the chosen token is actively being used, but the number of tokens that need to be revoked in this way is not large. If clients were to perform the task of garbage collection themselves, this mechanism should not be necessary.

4.4.4 Crash Recovery

Servers that maintain state about their clients require more complicated crash recovery mechanisms than stateless servers. This has caused server writers to avoid keeping track of state whenever possible. Nevertheless, the token server keeps a great deal of state about clients, going so far as keep track of everything

that each client is currently caching. It is therefore important to show that the token server can recover from crashes in a reasonable, efficient way. The strategies for crash recovery may vary, depending on whether the token server is replicated.

At the time of writing, MFS uses a very simple logging scheme to allow a single server to recover from a crash. Every so often, the server writes all its data structures to a checkpoint file. Each subsequent token request is then logged in a file which can be quickly read after a crash. The load on the token server is such that this scheme works tolerably during simple tests, but better crash recovery schemes are certainly possible.

Crash Recovery without Replication

If there is only one token server, there are two extreme solutions, and a range of compromises in between. One extreme is to record the state of the token database in non-volatile storage at all times. This may incur little overhead on machines with non-volatile memory, but it is likely to take several milliseconds if a disc must be accessed. Of course, only incremental changes need be written to disc in the form of a log, but each write necessarily includes the rotational latency of the disc.

The other extreme is to record only the names of active clients on non-volatile storage. When the server restarts, it attempts to contact each client to discover which tokens it held before the crash. The server waits until all clients have responded, or sufficient time has elapsed for the clients to notice the communications problem. After this time, the server can once again issue tokens in the normal way. This approach is simple, but relies on the clients replying honestly. A malicious client could steal a token from another machine holding a modified copy of the file, and hence temporarily prevent file updates. Additionally, it may take a long time to interrogate all the clients and re-establish all the tokens, particularly if some of the clients have crashed or are otherwise uncontactable. The danger is that an uncontactable client may still hold some tokens, unknown to the token server. Until the token server is sure that the client has noticed the problem, the server cannot issue new tokens for fear of conflict with existing ones.

A combination of these solutions is to write a checkpoint file and a log, but to write the log only every minute or so. In the event of a crash, the checkpoint and log can be read efficiently, and the additional tokens issued during the last few minutes can be obtained from the clients. The number of tokens to be collected from clients is likely to be small, and can be gathered using very few messages. Hybrid solutions are still vulnerable to the problems of fraud and uncontactable clients.

Unforgeable Tokens

Token fraud is unnecessary in MFS, because a malicious client can always write directly to the file server, thereby bypassing the token mechanism altogether. In most systems, the normal file access control mechanism will be sufficient, but secure environments may require protection to prevent malicious clients from causing cache inconsistency.

One possible technique is the use of *unforgeable tokens*. These are fairly simple to generate, given a hash function, $hash(x)$, which is:

- Efficient. $hash(x)$ must be easy to generate, given x.
- One-way. Calculating x must be computationally infeasible given $hash(x)$, even if some parts of x are already known.
- Collision-free. Given x, it must be computationally infeasible to find a y such that $x \neq y$ and $hash(x) = hash(y)$.

Functions of this type are used in cryptographic signatures [CCITT 87], and can be manufactured from encryption algorithms [NBS 77] [Wheeler 87]. Given such a function, a scheme for generating unforgeable tokens is simple. The token server keeps a secret number, X, known only to itself, and a sequence number, T. The sequence number is incremented each time a token is issued, so each new sequence number is greater than all other sequence numbers issued from this server. This property can be efficiently preserved across reboots by basing T on the time, provided the clock is never set back. If a reliable clock is not available, T can be written to non-volatile storage periodically, say every N increments, and during a reboot, T can be reset to be the last stored value plus N. When the server issues a token, it constructs S, the concatenation of X, T, the client identifier, the token type, the token group identifier, and the token range within the group:

$$S = X \cdot T \cdot client\_id \cdot token\_type \cdot token\_group\_id \cdot token\_range$$

The server calculates $hash(S)$ which is passed to the client, along with T. After a server crash, a client can present these values to the server, which can verify that the hash value is correct. The value T indicates when the token was issued in relation to all other tokens, allowing the server to determine which tokens were in use at the time of the crash, provided all trustworthy clients reclaim their tokens. A malicious client cannot claim to hold a token that was never issued, because it does not have X and so cannot calculate $hash(S)$. It cannot claim to hold a token that was once issued to another client (assuming clients can be reliably identified), because the *client_id* will be incorrect. If it claims that it still holds a token that was released due to conflict, the situation can be resolved by examining the sequence numbers. If there are no conflicts, a malicious client gains nothing by claiming a token. The file server should not allow any write tokens to be used until all clients have had a chance to claim their tokens, since another client may have a conflicting write token.

Generating $hash(S)$ can take a significant time; many cryptographic functions require tens of milliseconds of computation on current processors. At times of heavy server load, this may cause excessive delay. It may be possible to reduce the delay by adding appropriate hardware to the token server, since only the server needs to be able to calculate the hash functions. In fact, in a secure environment, every machine would probably require encryption hardware in any case. Faster scrambling algorithms can be used, such as Wheeler encryption [Wheeler 87] but

even these will occupy the server processor fully for a a substantial fraction of a millisecond.

A simple way to reduce the number of hash calculations is to make only write tokens unforgeable, leaving read tokens unprotected. Since read tokens are likely to be far more common, this simple approach will significantly reduce the workload on the server, without causing any loss in security. During the crash recovery procedure, the file server disallows writes. The token server collects old write tokens, but issues no new write tokens. It permits clients to obtain read tokens, but honest clients claim only tokens that they held before the crash. This allows limited read-only access during the recovery procedure. When the server has contacted each of its clients, or has tried for a sufficiently long period, it resumes normal operation. A malicious client can hold a read token until an old write token is found, but cannot alter the file system in any way, and so can only compromise the consistency of its own cache.

Optimizations

The extra security afforded by unforgeable tokens is small, and possibly better provided by other mechanisms. Logging provides better security guarantees, and may well be a better choice for a highly secure system. Here too, it is possible to distinguish read tokens and write tokens in order to avoid excessive overhead. If only write tokens are logged, the token server can immediately issue new read tokens after a crash. The only modification is that the server must not issue new write tokens until it is sure that all clients have detected the crash and have thrown away their old read tokens.

A good way for the token server to reduce the probability of delay while waiting for crashed clients is to keep a record of which clients are uncontactable. When the server reboots, it can disregard any clients that fail to respond if they were known to have crashed at the time the server went down.

Crash Recovery with Replication

Network services are replicated for a variety of reasons:

- Server load can be distributed across many machines.
- Each server can be close to its own clients in a large network, to avoid communication delays and network gateway congestion.
- Replicas in different partitions can sometimes continue to operate independently.
- Other servers can sometimes take over the work of a crashed server.

Distributing the load across many machines is a fairly trivial task in the case of the token server, since token groups can be divided among the available token servers. It is quite difficult to ensure that several *equivalent* instances of the token

service are available at different parts of the network, since the purpose of the token service is to allow conflicts to be resolved at one centralized point. However, there are techniques that would allow other servers to take over the work of a crashed server, and these will now be examined.

One solution to this problem is to allow the available token servers to vote over which token groups each will be responsible for whenever servers crash or reboot. Distributed voting strategies have been used for some time [Gifford 79], and by use of suitable algorithms can give high availability, even in the face of multiple failures [Barbara 86]. Given a voting mechanism, a group of active token servers can decide on a master, which can then allocate responsibility for token groups amongst the servers. A simple way to perform this allocation is to form a list of the servers that voted, and issue this list to all servers and clients. Clients can then hash the token group identifiers onto the array of active token servers. This approach has the disadvantage that management of every token is likely to be transferred when any server enters or leaves the server group. A better scheme is to hash the token group identifiers onto the list of all token servers, even those that are uncontactable. If the chosen server is down, the identifier is rehashed until an active server is found. This algorithm changes the list of tokens managed by each server as little as possible when servers enter and leave the voting group, but maintains an even distribution of tokens on servers at all times. A full redistribution must take place when new servers are introduced to the system, but this is usually less common than a server crash.

Assuming the token servers are able to distribute responsibility for the various tokens during a reconfiguration phase, two problems now remain.

- How can two active servers transfer the state of a token group?
- How can an active server obtain the state of a token group previously managed by an uncontactable server?

Of these, the former is straightforward. The two servers can engage in an atomic transaction (e.g as described by Paxton [Paxton 79]) that transfers the state of the token group from one to the other. The latter problem is more difficult.

Log-based crash recovery schemes are not useful if a replica server is to take over the work of crashed system, unless the new server can read the log. This is possible only if the crashed server logs its actions by sending messages to other token servers, or if discs are multi-ported and accessible from multiple token servers. Logging to other servers is often faster than logging to disc, since there is no rotational latency. Even so, the fastest request-response protocols are unlikely to take less than a millisecond or so, and this time must be added to the overall response time of the server. If servers log their actions to all other available servers, the total number of messages may be greatly increased; if they send their messages to a subset of the servers, the possibility remains that all servers with knowledge of a particular token's state could be down.

The amount of network traffic can be decreased by putting multiple messages in a single packet or using broadcast and multicast techniques. Even so, there is a

constant minimum delay between sending a log message and receiving a reply. As was described in the section on crash recovery without replication, the overheads of logging can be reduced at the expense of additional work during the crash recovery procedure.

It is unclear which of these recovery techniques is the best. Multiple server schemes have the advantages of high availability and good load sharing, and are likely to be the most effective. Further work is needed to determine the tradeoffs between solution complexity, availability and performance.

4.5 The Client

4.5.1 General Description

MFS clients have been implemented on a set of MicroVAX-II machines running ULTRIX. In order to minimize changes to the operating system kernel, most of the client code resides in a privileged user-space process called the *cache manager*. This process is responsible for keeping track of the contents of the cache, obtaining new tokens from the token server and reading and writing file data at the file server. It receives requests from the client kernel via a private interface, configured as a device driver.

The cached data is held in the local file system of the machine using the standard directory structure. It is manipulated by the cache manager using ordinary system calls augmented with a set of calls designed to optimize certain time consuming operations. As application processes access remote files, the kernel informs the cache manager of accesses to data that is not yet cached, and file modifications that must be written to the file server.

The kernel interface available to application programmes is unchanged, and the semantics have been preserved in almost all cases. The main exceptions to this rule are the failure semantics, and certain file properties, such as the file last-accessed time, which are difficult to preserve efficiently in a distributed environment.

4.5.2 Client Conventions

Since the token server and the file server are separate in MFS, no fixed interpretation is placed on a token by the token server. Clients must agree on a set of conventions to allow them to interwork reliably.

As was suggested in the description of the token server, the MFS client uses a separate token group for each file in the file system. Each byte of a file has a token associated with it; token 0 within a group protects byte 0 of the file, token 1 protects byte 1 etc. The file type, permissions, owner and group of the file are protected by a single token, number $2^{32} - 1$. The file type (normal, directory, etc) is known to be immutable for a particular file, but the permissions information can change at any time. Client machines cache the permission information in order to mimic the file access checks performed by the file server itself. These checks can be performed much faster locally than by contacting the server, but are not

needed from the point of view of security. A malicious client can always lie to its own users, regardless of where the access checks are normally performed. It is the final server check that limits the damage that can be caused by a malicious client; the local checks merely allow trustworthy clients to provide equivalent service at higher performance.

The last-modified time and last-accessed time are also read under the protection of this token, but are updated in an informal manner in order to avoid loss of performance on each read and write. The resulting consistency guarantees for this and other file information are discussed in Section 5.1.

The size of a file must be treated specially, since it can be modified either by truncating the file, or by writing bytes beyond the end of file:

- If the size of the file is to be read, the client must hold tokens for all bytes from the end of the file to token number $2^{32} - 2$.

- If the file is to be truncated, the client must hold write tokens for all bytes from the truncation point to token number $2^{32} - 2$

- If bytes are to be written to any part of the file, including past the end-of-file point, write tokens must be held for the range to be written.

These three rules are sufficient to ensure consistency of the size of the file. The first rule may require the client to iterate towards the correct file size, since it cannot know which bytes require locking until it knows the size of the file. If the file server and token server were combined, a single call could be introduced to avoid this potential iteration. In practice, clients can often make a reasonable guess about the file size and lock the correct region in one step.

Directories are locked in the same way as files, except that the directory contents are always locked as a single unit, by claiming tokens from zero to $2^{32} - 2$. Partial directory caching is certainly possible, but it is unlikely to yield a significant increase in performance. There are several reasons for this:

- Directories are typically small; the mean directory size on most UNIX systems is less than one kilobyte.

- Directory listing programmes always read the entire directory.

- Since UNIX directories are not sorted, the entire directory must be scanned for duplicates when a new entry is made.

If large directories existed which were never listed in full, it would be worth sorting the entries in some way (alphabetically, or by length) and assigning tokens to ranges within the sorted list. A client could then cache all file names beginning with *ca*, for example.

4.5.3 Kernel Modifications

Operating system kernels are often difficult to debug, so in MFS, the kernel changes were kept to a minimum. This approach reduced development time, but has undoubtedly led to a less efficient implementation of an MFS client; additional context switches are required whenever the cache manager is to be run. Nevertheless, since the cache manager is normally invoked only on cache misses, the performance of the system during normal use is close to that of a machine with an unmodified local file system.

Several changes were necessary to the file system code in order to allow the cache manager to take control of file system operations. Most importantly, a new device driver was added for communication with the cache manager. When a kernel process needs to invoke the cache manager, it calls a routine in the device driver with the parameters of the request. The device driver assembles the request into a packet, queues the packet for delivery to the cache manager and optionally puts the calling process to sleep, waiting for a reply. As the cache manager reads from the device, successive packets are read, and interpreted. If a kernel process is waiting for a reply, the cache manager writes to the device, giving the return value and a unique identifier for the kernel process. The device driver finds and wakes up the waiting process, passing it the return value. With this basic mechanism in place, it is possible to call the cache manager from any kernel process.

If the cache manager process aborts, or has not been started, waiting processes are allowed to continue, and the cache is treated as an ordinary part of the local file system.

The cache manager recognizes the following kernel requests:

LOOKUP A directory lookup requires a remote file or directory to be accessed. The kernel process waits until the specified file or directory has been cached.

READTOKEN, WRITETOKEN The kernel requires some read or write tokens for a file. READTOKEN has the side effect of caching at least the data protected by the token.

DIRWRITTEN, DIRREWRITTEN, DIRREMOVED Indicates that a name has been written, overwritten or removed from the given directory. The calling process does not wait.

CHMOD, CHOWN, UTIME, TRUNC Informs the cache manager that a file had its permissions or owner changed, its last-modified time set or has been truncated. The calling process does not wait.

FSYNC Waits until the cache manager has flushed to the file server all dirty data associated with the file.

MODIFIED Informs the cache manager that some file data has changed. The calling process does not wait.

A few additional calls are not yet supported by MFS. The most important is a request that free space be generated in the cache. This would be used whenever the kernel found the cache file system full. At present, MFS relies on keeping at least one free block in the cache at all times. The remaining unsupported calls are needed for application level advisory file locking. These operations could easily be supported using the token server to store locks, in the form of additional tokens.

A kernel process reading a file makes a READTOKEN call to ensure that the data is cached before it is read. Similarly, a process writing a file calls WRITETOKEN just before the write and MODIFIED just afterwards. All file system modifications are notionally bracketed between two calls; the first call obtains write tokens, and the second informs the cache manager of the modification. In order to avoid the majority of these calls, optimizations have been included for common cases.

Optimizations

Each file in the UNIX file system has a small data area associated with it, known as the *i-node*. The i-node stores information about the file that is not actually contained in the file, such as file permissions, ownership and size information. In MFS, a spare field in the i-node is used to store information about the status of the cached file. This field allows the kernel to distinguish the following cases:

- a normal, local file; not a cached file at all;
- a remote file about which nothing is known, except its name;
- a partially cached remote file, whose file type is known;
- a partially cached remote file, whose file type and attributes are known;
- a remote file that has been fully cached, and may be read.

If a client holds write tokens for an entire file, or has modified the file in the cache, these facts are also recorded in the i-node. Using this information, the kernel can avoid contacting the cache manager in many situations. Most notably, when a file is fully cached, reads can proceed at local file system speed. Also, when write tokens are held for the entire file, writes will proceed at full file system speed, and will be written to the file server asynchronously. By arranging to cache small files in their entirety, the cache manager is able to avoid an excessive number of calls from the kernel.

In addition to the essential kernel changes, a few extra system calls were added to allow the cache manager to perform its functions more efficiently. These permit the cache manager to manipulate the files and directories identified in the calls arriving from the kernel, without requiring the full pathname of each file.

4.5.4 The Cache Manager

The cache manager responds to three sources of external stimuli:

- requests from the client kernel;
- requests from the token server, reclaiming tokens;
- timeouts expiring, causing data to be written to the file server.

The cache manager is responsible for reading from and writing files to the file server, obtaining and releasing tokens on behalf of the client, and maintaining the client cache.

Data Structures and Algorithms

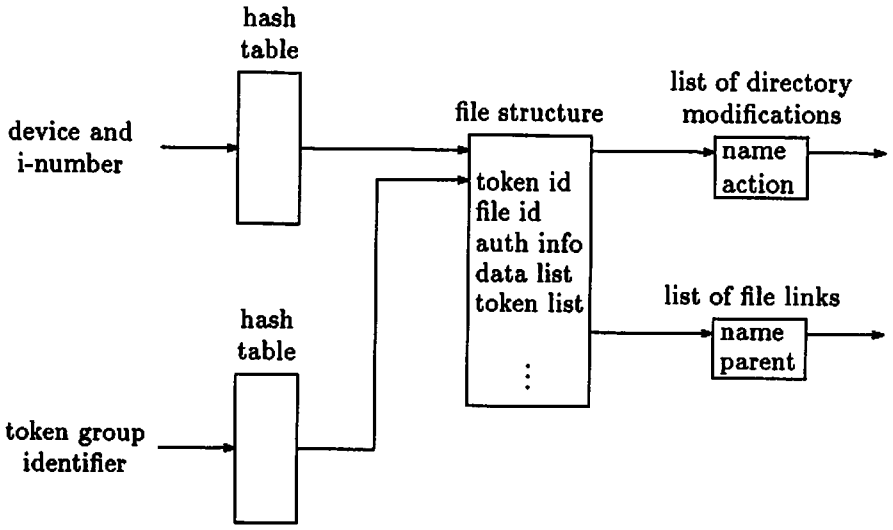


Figure 4.3: Cache manager data structures

The cache manager data structures are shown in Figure 4.3. For each file currently held in the client's cache, the cache manager maintains a *file structure*, which holds information about the file that cannot be held in the cache itself. This information includes:

- the file server and token server responsible for the file;
- identifiers that identify the file at:
  - the file server (i.e. the NFS file handle);
  - the token server (i.e. the token group identifier);
  - the client kernel (i.e. the i-number and file name);
- a list of all parent directories containing a reference to the file (this list is needed for maintaining the consistency of directories. See below.);

- for directories, a list of all names inserted or removed from the cached copy;
- authentication information for delayed writes to the file;
- the list of outstanding tokens;
- the list of bytes cached or modified.

The list of outstanding tokens, and the list of cached bytes are each held in a *token array* structure, described in Section 4.4.3.

File structures must be found both when the kernel requires assistance from the cache manager, and when the token server wishes to reclaim tokens. For this reason, two hash tables are required, one allowing lookups by local i-node number, and the other by token group identifier.

Kernel requests start new threads, which find the appropriate file structures and perform the requested operation. Many requests are to cache a new file, or a new part of some file. In this case, the cache manager obtains any tokens it needs, reads data from the file server, writes it to the cache, and returns control to the kernel once again. When handling partially cached files, some read requests are for data that has already been cached, and no action is required. When cache data is modified, the cache manager notes the fact and adds the file to a queue of files to be written to the file server. Writing is delayed in the hope that further operations by the client will overwrite or delete the modified data.

Token conflicts are received from the token server and are handled in a similar way. A new thread locates the file structure, and examines the conflicts between its tokens and the requested token. By following the links to parent directories, the cache manager builds a full pathname for the file in the local cache, and can then open and modify the file.[3] First, the i-node of the file is read, and modified to temporarily disable kernel access to the file. If the cached copy of the data has been modified, the cache manager writes the new data back to the file server, and marks the region clean in the file structure. The cache manager retains a read token if it is able to do so, but otherwise will remove the data from the cache. It updates the token array and data array for the file, then sends a reply to the token server.

Dirty data is usually written back in the order that it was modified, approximately 30 seconds after the data was written to the cache. A background cache manager thread steps through a list of modified data, slowly writing it back to the file server. The order of write-back is modified by requests from applications (the `fsync` system call), from other clients requesting conflicting tokens, or by file server consistency requirements.

[3] An additional system call to open a file by i-number would simplify this operation, but has not yet been implemented in MFS.

The file server affects the write-back policies of the cache in unexpected ways, some of which require extensive support in the cache manager. The most obvious of these is the case where file permissions are changed before dirty data has been written. When performing synchronous operations on behalf of a user, the cache manager can simply pass the user's authentication information to the file server. However, when delayed writes occur, it is not always obvious whose authentication information should be used. MFS keeps the authentication information of the last user to write to the file, and uses this to write data back. While holding dirty data, the cache manager must ensure that it is still able to write the dirty data back to the file, and it does this by holding a read token for the permissions information of the file. If this information is about to be changed, all dirty data is written back to the server before this token is released. This is one of a number of situations in which the side-effects of file system metadata must be taken into account when using delayed writes.

Directory Modifications Another case where token release causes seemingly unrelated write traffic is a result of the way files are deleted at an NFS file server. In UNIX, a file remains in existence as long as some process has the file open, or the file still has an entry in a directory somewhere. An NFS file server cannot tell if a file is still open, and so operates exclusively on the second criterion. If the last name is removed, the file disappears, even though an application may be accessing the file. This problem is solved by holding a read token on the directory containing the file when it is opened. As long as the file is open, the client must be prepared to invent a temporary, fictitious name for the file if the read token on the directory is reclaimed to prevent the file being deleted at the file server. Most NFS clients already perform this operation if the name is removed by the client that opened the file. The token mechanism allows *any* client to remove the file without harm, at some cost in complexity.

Another example of this effect can be observed in the following sequence of operations:

- Some application on client *A* executes `link("/foo/a","/bar/b")`, to create another name (`/bar/b`) for the existing file `/foo/a`.
- *A* obtains read tokens for `/foo` and write tokens for `/bar`.
- *A* then performs the link operation in its own cache, but does not write the changes back to the file server.
- A little while later, an application on client *B* executes `unlink("/foo/a")` to remove the original link.
- *B* requests write tokens for `/foo` in order to remove the name. *A* has only read tokens for this directory, and gives them up immediately.
- *B* then writes its changes back to the file server, which observes that the last link to the file has been removed, and it deletes the file.

- *A* finally decides to write its changes to the file server, but finds that it cannot create the name `/bar/b` because there is no file for it to name.

Other examples can be constructed using only one client, which renames a file and flushes the directory modifications in the wrong order. The solution used in MFS is to record additional information in the directory modifications list whenever a file is `linked` or `renamed`, and to flush changes to the modified directory if the read tokens on the source directory are reclaimed. Modifications to a directory are stored as a list of modification records. Each record contains a file name, and a state, indicating how the cached copy of the directory has changed since it was last written to the file server. It also contains a pointer to the file structure of the file, which allows each cached link to the file to be found. A modification record can have one of five states:

NULL: no modification.

ADD: the name has been added to the directory.

DELETE: the name has been deleted from the directory.

REPLACE: the original file has been deleted, and a new file put in its place.

CHECK: this entry has not been modified, but it has been used to generate other links. These links must be flushed to the file server before tokens are reclaimed for this directory.

In any directory, there are always either zero or one directory modification records for a given name. If there are no modification records, the name is in the NULL state.

There are three operations that can be performed on a name in a directory:

- *link*—create a directory entry elsewhere that refers to the same file;
- *remove* the name;
- *add* a name;

There is no *rename* operation; rename is built up from *add* and *remove* operations. Table 4.1 shows how these operations move entries between states.

Initial State	Action		
	link	*add*	*delete*
NULL	CHECK	ADD	DELETE
CHECK	CHECK	*error*	DELETE
ADD	ADD	*error*	NULL
DELETE	DELETE	REPLACE *or* CHECK	*error*
REPLACE	REPLACE	*error*	DELETE

Table 4.1: Directory Modification State Table

The *error* states are present because it is impossible to delete an entry that has already been deleted, and impossible to add an entry if one already exists.

The *add* operation generates CHECK state from DELETE state only if the file name being added is the one that was previously deleted. The entry enters CHECK state rather than NULL state because there is no record of whether this entry was used as the source for a *link* before the *delete*. This information could be stored, but the extra check involved is quite efficient.

The *delete* operation converts ADD to NULL, rather than CHECK, because the deleted entry is not on the file server anyway. If it had been there, the previous state would certainly not have been ADD. This state transition represents the case of a temporary file being created, then deleted entirely within the cache. No data need be written to the file server, unless some external event or timeout causes the modifications to be flushed.

When a read or write token for a directory is reclaimed by the token server, the cache manager scans the list of directory modifications. Even if the directory was only read, there may be entries in CHECK state. For each directory entry in CHECK, DELETE or REPLACE state, all other links to the file are found by examining the list of links attached to the file structure. If another link to the file is being created in another directory, that change is flushed to the file server immediately to ensure the continued existence of the file. Entries in ADD state can be flushed without additional checks, since they will never delete data, but the cache manager checks each link in order to optimize `rename` operations. Many ADD/DELETE pairs can be reduced to a single file server `rename` operation. New files that have not previously been written to the file server are assigned token identifiers only when they are created at the file server. The client must be sure to obtain write tokens for the file before allowing other clients to examine the directory and access the file.

In rare cases, it is possible to find a circular chain of entries to be flushed. Consider the case where files `a` and `b` have been swapped, using a sequence of `rename` operations with a temporary file name. The modification records for both `a` and `b` will be in REPLACE state, but neither operation can proceed since each requires that the other be flushed first. This situation is resolved by inventing a new temporary file name for one of the files. A DELETE record is added to the list of modifications to ensure that this name will be deleted eventually. No other client will ever see the name, since it is always deleted before the directory's tokens are released. It seems clumsy to invent a file name to circumvent this problem, particularly since a temporary file name must have been supplied by the application when creating the circularity. Unfortunately, the optimizations designed to avoid writing temporary files to the file server also lose the route by which circularities are reached.

The checks described here serve to complicate the cache manager further than was originally expected. Unfortunately, it is difficult to avoid them, particularly if similar optimizations are to be performed. It may be possible to ignore these checks when the number of links to a file is sufficiently large that there is no danger of it being deleted. However, the number of files with more than one link is very

small, so the potential gain here is negligible. The danger of accidentally deleting a file could be removed by using file lifetime guarantees, rather than insisting that files were always held in directories. This leaves the possibility that files could be left inaccessible if a client were to crash at an inopportune moment.

Cache Replacement Policies

MFS clients rarely need to throw data out of their caches because each cache is 40 megabytes in size. For this reason, little work has been done on improving the cache replacement algorithm. The current algorithm is first in, first out (FIFO). A better choice would be to throw out data on a least-recently-used (LRU) basis, but accurate usage information is difficult to obtain. The client code is specifically designed to eliminate the involvement of the cache manager in most file operations, so the cache manager is largely unaware of file access patterns. Each cached file has its own last-accessed time information, but this can only be found by a costly scan of the cache file system.

The problems of building paging virtual memory systems without *referenced-bits* are well known. A simple technique that achieves a cache miss rate close to that of an LRU policy, but requires considerably less usage information is the *clock* method, which can be modified to work well even when referenced-bits are not available [Babaoglu 81]. The basis of the method is to place each page in a logical circular list, and to sweep a pointer, *the hand*, around the list, examining each page in turn. On each revolution of the hand, pages that have not been referenced since the last revolution are removed from memory. The speed with which the hand is moved around the list can be varied according to the demands on the available space. If sufficient free space remains, the hand need not move at all; during periods of heavy demand on the memory system, the hand may be moved quickly to find additional replacement candidates. If reference information is available, the referenced-bit of each page is examined as the hand passes over it. If the bit is clear, the page can be removed; if it is set, it is cleared before the hand moves on. On systems without referenced-bits, reference information can be obtained by putting pages into a *reclaimable* state. Reclaimable pages are marked as invalid in the page table, even though they are still in memory. When such a page is touched, a page fault occurs, but instead of initiating a transfer from disc, the fault handler merely marks the page as valid, and thus records that the page has been referenced. The action of the sweeping hand is to mark pages as reclaimable, if they are not already in that state, and to remove pages that have not been reclaimed since the last sweep of the hand.

This algorithm can easily be applied to the MFS cache manager. Files that are only partially cached already cause calls into the cache manager, which can easily record references in the file structure. Files that are fully cached can be marked as partially cached in their i-nodes. On a subsequent reference, the kernel will check that the necessary part of the file is indeed in the cache. At this point, the cache manager can record that the file has been referenced, and mark the file as fully cached once more.

Most files are treated as a single unit for replacement purposes. It may be advantageous to divide large, partially cached files into a series of blocks, and to consider each block separately for replacement, but the UNIX file system primitives make this hard to do. There is currently no way to remove an arbitrary section from a file; truncation and deletion are the only operations that free space within a file.

Performance Improvements

The token server allows files to be cached in any size unit, from the file to the byte, but certain conventions are necessary if reasonable performance is to be achieved. The cache manager can round data requests to any block size, without fear of cache inconsistency, since each byte is individually protected. Block size should normally be chosen on the basis of the client cache size and the preferred file server transfer size. In the case of MFS, 8 kilobytes is preferred, though initial experiments were done on a byte-by-byte basis.

Two important techniques for increasing the performance of the system are whole file caching of small files, and read-ahead. In fact, whole file caching is often a side-effect of simple read-ahead, since small files are almost always read in their entirety. Read-ahead is implemented by initiating a read on the next block of data as well as the requested block, if the next block is not already in the cache. It has the effect of pipelining the processing and fetching of a file, and can improve the start up performance of many applications.

MFS currently has no explicit code to promote whole file caching. Any file that is read from beginning to end will be fully cached, but large, randomly accessed files will be partially cached. This would include large executable files if MFS supported demand paging from the file server. Additional heuristics are needed to decide when large files should be prefetched, and which parts should be removed first when they are to be thrown out of the cache.

Concurrency Issues

The MFS client is a complicated concurrent system. Its code runs partially in kernel-space, partially in user-space, and both kernel and cache manager have multiple threads of control. The locking strategies for the data structures in this system are complex, and in some cases deadlock is difficult to avoid. The most difficult problem is caused by the need for both kernel and user threads to hold locks on important data structures, with the constraint that neither can access the locks held in the other's address space.

When performing a file operation, a kernel thread must lock the i-node of the file to ensure that no other kernel thread modifies it, and that tokens held for the file will not be given up before the operation has completed. In the case of a partially cached file, the kernel must ask the cache manager for confirmation that particular tokens are held without releasing the lock, since the information could become out of date at any time if the lock is not held. Unfortunately, the cache manager might block on this same lock while trying to release tokens, which would

cause deadlock. This occurs only because UNIX does not allow independent user level threads to occupy the same address space. Fortunately, deadlock is rare, and a fairly crude technique is sufficient to resolve the problem. If a kernel thread has waited too long for a reply from the cache manager, it releases all its locks and waits for the cache manager to respond. It then restarts the pending kernel operation and reclaims the locks it previously held.

Mutual exclusion locks are used on file structures within the cache manager to guarantee that updates are serialized. As with the token server, only one lock is allocated per file, but little additional concurrency could be gained by locking parts of files independently. Internal cache manager operations are short, and the locks are always released during RPCs to the token server to avoid deadlocks with other clients.

Kazar describes a problem with callback synchronization in the Andrew file system [Kazar 88]. The problem arises when the client is in the middle of requesting a token but the token is reclaimed by the server before the client has received the reply to the original request. The problem is that the client cannot tell whether the reclaim request corresponds to an earlier token, or the token currently being requested. Kazar recommends solving this problem by attaching version numbers to tokens, so that the reclaim request can be matched with the correct token. An alternative solution, which is the one actually implemented in Andrew and in MFS, is to discard the token request and claim another token immediately. The version number approach is almost certainly the cleaner of the two schemes, but the problem is rare enough that the inefficiency of the latter solution is tolerable.

4.5.5 Crash Recovery

Server Crashes

The crash recovery procedures required by the token server itself were discussed in Section 4.4.4. However, the client is also affected by server crashes, and clients may behave in slightly different ways, depending on the needs of their users. When the token server crashes, or is uncontactable, clients may be able to continue for quite long periods without requiring any additional tokens. If a client were to continue during a network partition, it is possible that the token server may issue tokens to another client, assuming that the first client had crashed. But since write-sharing seems to be rare, we can infer that the probability of inconsistency is correspondingly low.

If the file system is required to have good consistency properties at all times, the possibility of conflicting tokens existing concurrently is unacceptable. A solution is for each client to attempt to contact the token server every T seconds, where T is less than the interval that the server waits while trying to contact a crashed client. Clients must discard all their tokens as soon as they detect loss of contact with the token server. In such situations, client applications must wait until contact is restored. At that time, the client will begin to reclaim tokens and applications will be permitted to continue. Even under this scheme, the file system consistency properties are not perfect if delayed writes are permitted. A

write-through caching policy gives better guarantees in the face of crashes, with some performance penalty.

An alternative approach is to permit clients to continue during server crashes, on the assumption that conflicts are unlikely. This has the advantage that existing tokens can be used, and new files created during extended periods of unreliability. The LOCUS system [Walker 83] allows partitioned updates to files, and even attempts to resolve conflicting updates for certain file types, such as mail files and directories. When automatic conflict resolution is impossible, the situation is brought to the attention of the interested user(s), who must then resolve the difficulty by hand. A simpler solution is to choose one of a conflicting set of updates, ignoring the others. This solution feels uncomfortable, but may be sufficient if users are informed of server crashes when they occur and required to take explicit action to allow applications to continue. Some applications may be tolerant of temporary inconsistency, and may be able to advise the client operating system if they will accept data under these conditions.

Each application will have slightly different consistency requirements, and will be subject to varying degrees of unreliability, particularly if sharing takes place between many people. It is therefore difficult to decide between the two approaches in a general way. One possibility is to allow the behaviour of the system to be selected independently for each individual file, by associating an extra bit with the file. A possible outcome is that most users' files would be used in the permissive style, but important shared files and databases could be marked as requiring consistency at all times, and so could not be used during server crashes. Experimentation is needed to compare the effects of these possibilities.

Client Crashes

Clients may try to minimize the impact of crashes by preserving cached files across reboots. If the local cache is in volatile storage, such as a RAM disc, neither tokens nor data contents will survive crashes and power failures. A client would simply start with an empty cache each time it rebooted. If client crashes are rare, the overhead of starting afresh may be small. Client file operations will be slower when files are first accessed, but not excessively so.

Unfortunately, client crashes are not independent events. Often, they are caused by power failures, network faults or replicated software faults, and it is not uncommon to find that large numbers of machines crash together on a single network. At such times, servers can become overloaded as a number of clients try to reboot simultaneously. These events are rare, perhaps happening no more than two or three times a year, but as the size and complexity of systems increase, the potential impact of a single power dip can be enormous. Experience with the Demand-Initialized Disc system (Chapter 3) has shown that a power dip can render a network unusable for nearly twenty minutes because of the excessive server load in the period after the failure.

If the local cache is on a disc, the cache contents can be preserved across reboots, but other important information may be lost. Although the MFS cache

is usually on the disc, much of the associated information is not. After a reboot, the cache manager has no knowledge of which files are cached, which parts of them are cached, and what tokens are held. In fact, some of the information is available in the i-nodes of the cache file system; any file which is fully cached has all the necessary information associated with it to rebuild the cache manager's file structure. Since most files and all directories are cached in their entirety, even this partial knowledge would enable a client to avoid re-reading a great many files. By paying particular attention to important system files required for the bootstrap procedure, clients could avoid overloading file servers after a power failure or similar catastrophe.

Due to time constraints, the current version of MFS does not attempt to reuse data in the cache after a crash. Instead it uses a simple incremental algorithm to overwrite existing data, without allowing applications to see old data in the cache. First, the root of the cached directory tree is marked as invalid. As applications reference parts of the tree, each directory level is rebuilt, and the next level in the tree is marked as invalid. An incremental strategy was chosen in preference to a lengthy initial search phase in order to minimize the time required to bootstrap the system. A background process removes old parts of the tree if they are not accessed shortly after booting.

An algorithm that could be used to restore a major part of the file system from the disc image is given below.

- The root of the cached directory tree is marked as invalid.
- A new token is obtained for the root directory. If the token server's sequence number has changed, the client's tokens were revoked during the crash.
- Applications start to examine parts of the tree, and a background cache manager thread runs over the entire tree to ensure that it is all processed eventually.
- For each partially cached file, the cached data is discarded.
- For each fully cached file touched by an application:
  - If the client lost no tokens during the crash, the cached copy is up to date.
  - If the client lost all of its tokens, the client obtains new tokens, and checks the last-modified time of the file at the server. If the cached copy appears to be more recent, it is kept and is written back to the server if it has been locally modified. Otherwise, the cached data is discarded.

This algorithm is certainly imperfect; delayed writes are lost on partially cached files, and when update conflicts have occurred during the crash. However, less data is lost than with a volatile cache, and most cached files need not be fetched

again. Better results could be obtained by keeping more information on non-volatile storage, but this seems unnecessary. If the incidence of client crashes is so high that this yields a noticeable improvement in performance, perhaps effort should be directed towards eliminating the causes of failure.

4.6 Free Space

In the current version of MFS, there is no easy way for a client to reserve free space on a file server disc. Without free space reservation, a client cannot safely cache its writes, since it cannot be sure of being able to write them later. Instead, it must write all new data on the file server immediately.

A primitive free space allocation mechanism could be built without modifying the file server, provided all clients can be trusted to behave fairly. Each client would initially create a file of a certain size to reserve that amount of free space. The client must keep write tokens for the file, so that no other client can truncate the file. To use the reserved space, a client would first truncate its reservation file, then write its data to the file server. All clients would have to be trusted not to steal free space in the interval between the truncation of the file and the writing of the data. If a client ran out of its reserved free space, it could request N bytes of free space from another client (the *victim*) by the following procedure:

- Find the current length of the victim's reservation file. Call this length L_{init}.
- Request write tokens for all bytes beyond $L_{init} - N$ of the victim's reservation file.
- When the tokens are granted, find the length of the file again. Call this value $L_{current}$.
- If $L_{init} - N > L_{current}$, truncate the file at byte position $L_{init} - N$. The victim has allowed us to use $L_{current} - L_{init} + N$ bytes.

When the victim is told that another client has requested write tokens, it can decide how much free space it wants to release. If it wishes to release only n bytes to the other client, it truncates the file by $N - n$ bytes, and writes $N - n$ bytes of dirty data onto the disc before releasing the write tokens for the file. The victim requests more write tokens for its reservation file as soon as it has released them, to return itself to its previous state. A client may unwittingly obtain more free space than it initially intended if the victim was in the process of reserving more space at the same time, but the victim always has the option of denying the request. Provided all clients are well behaved, keeping only the reservations that they need, and choosing suitable victims, this scheme could be made to work.

If the file server can be modified, a similar scheme of reservations can be implemented, but without the disadvantages of the scheme described above. The file server would choose which clients should give up reservations, perhaps logging antisocial behaviour on the part of the clients, and preventing clients from exceeding

their reservations. The mechanisms required at the file server are similar to those needed to enforce file system disc quotas in a time-sharing system. Both file server and clients require additional interfaces:

- to allow a client to bid for free space;
- to request that clients give up free space, when it is required by other clients.

Unlike the token service, the reservation service must be combined with the file server for performance reasons. Free space reservations can potentially change on each write to the file server. Only the file server has easy access to the data involved and only the file server can prevent a client exceeding a reservation.

Whatever technique is employed for free space reservation, the performance of the system will degrade if there is insufficient free space. Each client will be constantly requesting space from the others, resulting in many messages. Eventually, the system will be using a write-through policy, but will be doing a great deal more token passing than is necessary. The file server could use heuristics to detect this situation and disable caching, if this were a common case. In most environments, peer pressure or user quota systems can be used to control this problem before it becomes too acute.

An alternative to space reservation is to build file servers with storage capacities so great that they will never be exhausted. A possible implementation would be a file server that automatically moves files between disc and tape with the help of a human operator, or tape library robot. The file server could delay requests until the necessary space or data was available. The system performance would degrade gracefully until the set of commonly accessed files exceeded the disc storage capacity. This situation calls for the purchase of more discs, rather than an improved algorithm.

4.7 Summary

A design for a caching file system has been presented, and a prototype implementation described. MFS has large client caches stored on local disc to minimize contact with the file server. A *token server* co-ordinates access to files to preserve cache consistency and informs clients when they should flush modified data from their cache, or discard old data. Fine grain sharing is supported at the granularity of individual read and write system calls of a single byte.

Algorithms for crash recovery and cache management have been presented, which are both efficient and practical. The implementation has shown that:

- It is practical to maintain cache consistency by keeping records of the contents of client caches in a centralized server.
- Efficient data structures can be designed for storing fine-grain caching information.

- It is possible to build a caching system using an existing local file system and existing file servers.
- It is possible to separate the functions of the token server and the file server, at the cost of some performance.

The next chapter examines the behaviour and performance of the prototype.

Chapter 5

Behaviour and Performance

This chapter describes the behaviour and performance of the MFS design presented in Chapter 4. Section 5.1 describes the semantics of MFS and compares it with those of a normal UNIX file system. Section 5.2 presents performance figures for MFS, and compares them with existing file systems.

5.1 Semantics

MFS conforms well to most of the more esoteric file system semantics normally associated with UNIX. It allows fine-grain concurrent write-sharing of files, and allows them to be removed from the name space while still open, as timesharing UNIX systems do. Nevertheless, a few incompatibilities exist:

- If the permissions on a partially cached file are changed, an application with the file open may not be able to continue to read or write the file. This problem is intrinsic to the NFS file server interface, and is difficult to solve without modifying the server to allow the use of capabilities, or some other handle that grants file access.

- The *link count* of a file should be equal to the number of file system entries for the file. In MFS, it is not correct unless all the directories containing the file have been cached. This is a side-effect of caching data in the local file system structure. This problem can be corrected by maintaining an additional link count in the i-node, and returning this to application programmes in place of the local link count.

- All file time information is slightly suspect if the clocks of the servers and the clients are not synchronized. Each machine uses its own clock to timestamp files, and these timestamps are not converted when they are passed between machines. This effect could be minimized by synchronizing client and server clocks using a clock synchronization protocol [Lamport 87], or by including clock values in messages sent from file servers to clients.

- A file's *last-modified time* should indicate when any application last wrote to the file. Delaying write-backs can cause an error of up to 30 seconds in the last-modified time of a file, since the time is recorded at the server only during write-back. This error could be avoided by setting the file modification time on each write-back at the cost of an extra RPC, or by altering the file server interface slightly.

  Moreover, a client machine *does not necessarily communicate* with any other machine when it writes part of a file. So, when a file is partially cached on more than one machine, the local copy of the last-modified time may not be up to date. Because the file size and file time information are requested together, the modification time is locally correct whenever one of the following is true:

  - no writes have taken place on other machines since the file was cached;
  - the file size has changed since the modification time was last read;
  - the file is fully cached.

  Since most files are not write-shared, are fully cached, and they change size when they are updated, the last-modified time is rarely affected by this problem.

- The file *last-accessed time* is very difficult to maintain efficiently. MFS usually supplies the last time the file was read on the local machine, but if there have been no reads locally, it gives the last time the file was cached by any machine.

- The *changed time* of a file is not maintained at all. This value normally gives the last time the file was modified or any property of the file was altered. It is modified as a side effect of the cache manager's activities and a kernel change would be needed to prevent this.

Many of the problems described above can be overcome without gross inefficiency, but this is not always the case. Some of the problems would have an easy solution if the system call interface presented to applications were redesigned. For example, the file access time is rarely needed by a UNIX process, but it is returned with many other values which are frequently requested by the `stat/fstat` system calls. As a result, the file system must obtain the access time far more often than one would normally expect, and the performance of the solution must be correspondingly better. It is too late to change standard interfaces but better standards can be created from this experience.

5.2 Performance

The performance of MFS was measured under an artificial load provided by machines running the Andrew benchmark, which was described in Section 3.3.1. The

measurements were made with MicroVAX-II clients, each with a 40 megabyte disc cache and about 5 megabytes of main memory. The file server was a Sun-3/160 with a 70 megabyte low-performance disc. The MFS token server ran on an unloaded MicroVAX-II. The benchmark was run with numbers of clients ranging from 1 to 15; there were not enough suitable machines available to attempt a test with 20 machines. Each test was run at least three times unless otherwise noted.

Each MFS client was configured with a single cache manager thread responsible for writing modified data to the file server after 30 seconds if it had not already been deleted. The 30 second period was chosen to correspond to the normal delayed write interval in UNIX.

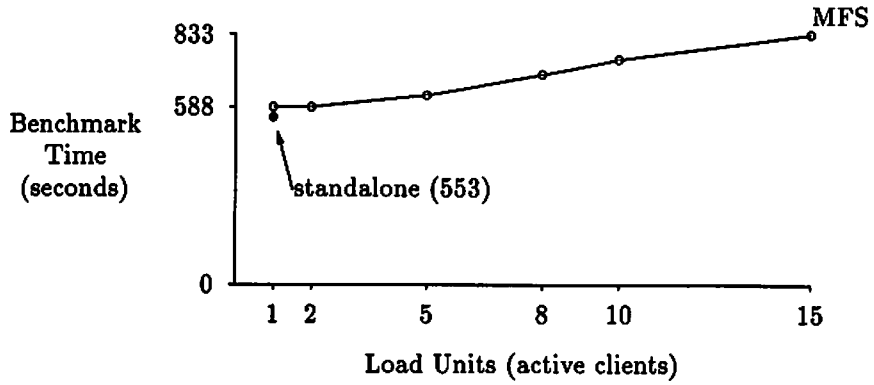
Table 5.1 shows the time taken to complete the phases of the Andrew benchmark for various numbers of MFS clients. Figure 5.1 shows the variation in the overall benchmark times for MFS. The table also shows the timings for the Andrew file system for comparison, though the Andrew clients had a significant advantage in CPU power. The relative performance of MFS and the Andrew file system are compared in Figure 5.2.

File System	Load	Overall Time	Time for Each Phase				
			MakeDir	Copy	ScanDir	ReadAll	Make
Standalone	1	553 (2)	6 (0)	28 (1)	26 (1)	47 (1)	445 (2)
MFS	1	588 (6)	7 (2)	57 (1)	28 (2)	44 (1)	451 (4)
	2	588 (7)	7 (1)	57 (2)	29 (0)	45 (1)	450 (5)
	5	628 (6)	8 (1)	92 (4)	26 (1)	50 (1)	457 (6)
	8	697 (4)	9 (1)	160 (5)	27 (0)	49 (1)	461 (3)
	10	748 (12)	8 (0)	207 (1)	27 (1)	51 (0)	462 (5)
	15	833 (6)	9 (1)	294 (5)	28 (1)	52 (2)	460 (1)
Andrew	1	588 (2)	5 (1)	71 (4)	100 (2)	50 (3)	363 (3)
	2	582 (4)	5 (1)	72 (3)	98 (0)	50 (2)	356 (2)
	5	605 (2)	7 (1)	85 (1)	97 (0)	47 (0)	368 (2)
	7	636 (4)	9 (1)	104 (2)	97 (1)	48 (0)	377 (2)
	10	688 (4)	12 (1)	137 (5)	94 (0)	48 (0)	395 (2)
	15	801 (2)	18 (1)	200 (3)	91 (0)	48 (1)	442 (2)

This table shows the elapsed time of the benchmark as a function of load for MFS. Figures are also given for the Andrew file system, taken from the paper by Howard *et al* [Howard 88]. All timings are in seconds; the figures in parentheses are standard deviations. All tests were run at least three times. Part of this data is reproduced in Figure 5.1.
All MFS clients were MicroVAX-IIs, as was the token server. The file server was a Sun-3/160 with a low-performance 70 megabyte disc. These measurements were taken without any active network servers, hence the standalone timings are less than the corresponding timings in Table 3.1 by about 10%.
The Andrew clients were Sun-3/50's. The Andrew file server was a Sun-3/160 with a fast, 450 megabyte disc. Only seven Andrew clients were used in one of the tests, whereas eight were used in the corresponding MFS test.

Table 5.1: Running Time of Benchmark for MFS

Although the overall time for the benchmark increases with the number of MFS clients, this is almost entirely due to the increased time for the Copy phase; all other phases complete almost as quickly at all loads. With the exception of the Copy phase, none of the phases reads a significant number of files that have not



The running time of the benchmark is shown for MFS. The data is taken from Table 5.1, which also describes the test conditions.

Figure 5.1: Running Time of Benchmark for MFS

yet been cached. The Make phase writes at least three large files, but its speed is almost independent of the number of clients. This seems to indicate that clients running under MFS are not significantly affected by the writing of modified data, but that reading from the file server is slow.

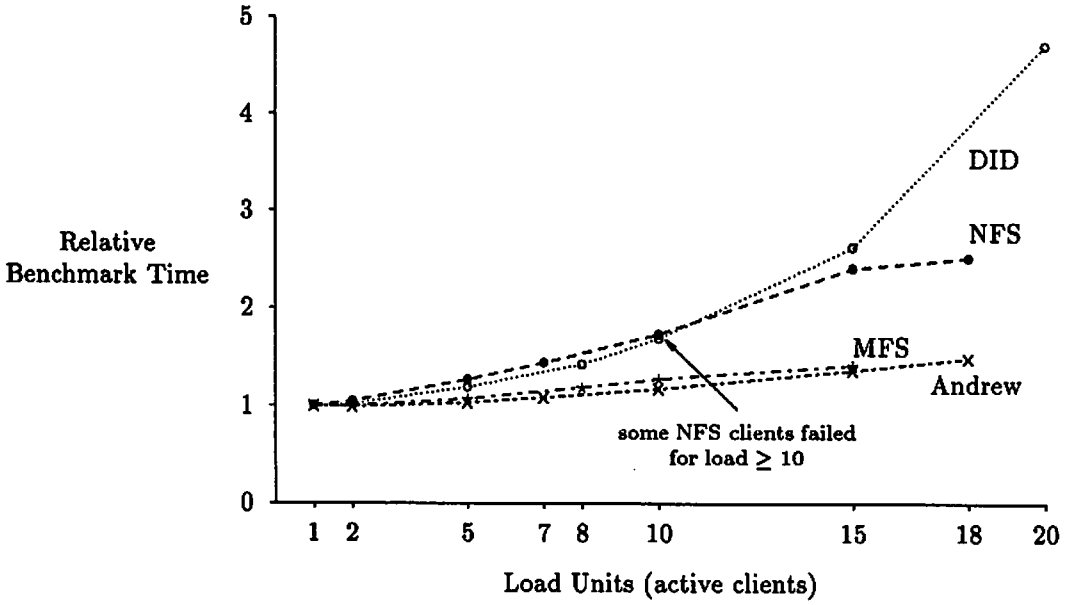
Andrew scales better for the Copy phase, but worse for the other phases which write data (MakeDir and Make). This indicates that Andrew is more efficient at reading files before they have been cached, but less efficient at writing. Andrew writes files back to the file server as soon as they are closed. The writing is initiated synchronously, causing one round trip delay on each file close. In addition, the Make phase of the benchmark generates a few temporary files in the file system under test when creating library files. Under Andrew, these will always be written to the file server; under MFS they will only be written back if they survive more than 30 seconds, which is unlikely.

The remote file access was further investigated by measuring the file access latency of MFS—the time taken to open a file, read the first byte and close it. These figures are given in Table 5.2 and Table 5.3, which compare the performance of MFS with NFS and the Andrew system for varying file sizes.

The reason for Andrew's superior performance when reading small files from the file server is believed to be due to the design of the Andrew file server interface. In Andrew, one RPC (Fetch) accesses a file given its directory entry. This one call returns the file status, the contents of the file and claims a token for the file (registers a callback). The MFS file server is an unmodified NFS file server, so at least three separate RPCs are needed to accomplish the same task:

- lookup the file name to obtain a *file handle*;
- obtain a token from the token server;
- read the file status and contents.

In fact, an oversight in the MFS implementation has caused the file status and contents to be fetched by separate RPCs, giving a total of 4 RPCs per file read by



The relative running time of the benchmark is shown as the number of clients running the benchmark is increased. All times are relative to the running time at a load of 1. Figures for NFS and Andrew are taken from the measurements of Howard *et al.* [Howard 88]. Note that some NFS machines failed to complete the benchmark at loads of ten and higher. The MFS test conditions are described under Table 5.1. Andrew and NFS used Sun-3/50 clients. The Andrew and NFS servers were Sun-3/160s, and were equipped with 450 megabyte, fast discs.

Figure 5.2: Relative Running Time of Benchmark with MFS

System	File cached	Data location	File Access Latency (ms)
Standalone (MicroVAX)	-	local disc	27 (0.2)
	-	local memory	3.2 (0.1)
MFS	no	file server	328 (7)
	partially	file server	146 (9)
	partially	local disc	35 (1)
	partially	local memory	11.6 (0.1)
	yes	local disc	27 (0.3)
	yes	local memory	3.3 (0.1)
Standalone (Sun-3/160)	-	local disc	23 (0.5)
	-	local memory	1.7 (0.1)
NFS	no	file server	54 (1)
	yes	local memory	10.3 (0.1)
Andrew	no	file server	160 (35)
	yes	local memory	16 (0.5)

This table shows the file access latency in milliseconds of a standalone system, MFS, NFS and Andrew. The files were all 3 bytes long. The latency is the time to open the file, read one byte and close the file. The Andrew times are taken from the paper by Howard *et al* [Howard 88]. The MFS times were measured with a single MicroVAX-II client. The Andrew and NFS times were measured with a Sun-3/50 client, which is approximately 1.5 times faster than a MicroVAX-II. Standalone times from both clients are given for comparison. All network tests used a Sun-3/160 as file server, which is about twice the speed of a MicroVAX-II. The figures in parentheses are standard deviations. All tests were run at least three times.

Table 5.2: File Access Latency of MFS

MFS. This could be reduced to a single call by redesigning the file server interface and combining the file server and token server. When a file has been partially cached by MFS, only one call is needed to fetch each new block of data (assuming that it has not already been prefetched). The time in this case is faster than the time to read a byte from an Andrew file (Table 5.2). The timings suggest that this modification to the MFS file server would improve the performance of MFS beyond that of Andrew, even when using slower client processors.

Andrew's cache seems to be quite slow; the ScanDir and ReadAll phases access only cached data. MFS performs better than Andrew on these phases, even though the Andrew clients were between 2 and 3 times faster. Both systems have user-space processes which trap application system calls and fetch data from the file server when required. The main design difference in this area is that MFS stores cached files in the local file system of the client, using the normal directory structure, while Andrew stores cached files in a separately managed area of the file system. The MFS approach involves additional system calls to place data in the cache, but requires no context switches when a fully cached file is read or written. This is reinforced by the data of Table 5.2, which shows that once a file has been cached, it can be accessed as quickly as if it were on a standalone file system. Both systems suffer from the delays of context switching when fetching data from the file server; this delay could be eliminated in kernel-space implementations.

Although the Andrew file system reads small files quite quickly, large files carry a performance penalty, as is shown in Table 5.3. The access latency increases in

File Size (bytes)	File Access Latency (ms)	
	MFS	Andrew
3	328 (7)	160 (35)
1113	331 (12)	148 (18)
4334	347 (3)	203 (29)
10278	419 (4)	310 (54)
24576	418 (6)	515 (142)

These tables show the latency of file access in milliseconds as a function of file size for MFS and Andrew. The test conditions are the same as in Table 5.2. The MFS transfer block size was 8 kilobytes; no more than 8 kilobytes will be transferred regardless of file size. Andrew always transfers the whole file. Figures in parentheses are standard deviations.

Table 5.3: File Access Latency by File Size for MFS

both systems up to the MFS transfer block size (8 kilobytes in this test). For larger files, the access latency is independent of file size for MFS, but continues to grow under Andrew. Andrew transfers the entire file when it is opened, with a large delay before the application is allowed to continue. In MFS, only the requested blocks are transferred, with a one block read-ahead. The effect is to transfer the entire file on the first access for all files less than 16 kilobytes, but to permit the application to continue as soon as the first block is available.

Server	CPU Utilization (%) by Load (active clients)					
	1	2	5	8	10	15
file server	4.0 (0.6)	6.6 (1.6)	7.1 (0.2)	10.4 (0.5)	11.2 (0.9)	16.1 (0.4)
token server	0.6 (0.0)	1.2 (0.0)	3.1 (0.1)	4.9 (0.1)	6.0 (0.0)	9.3 (0.2)

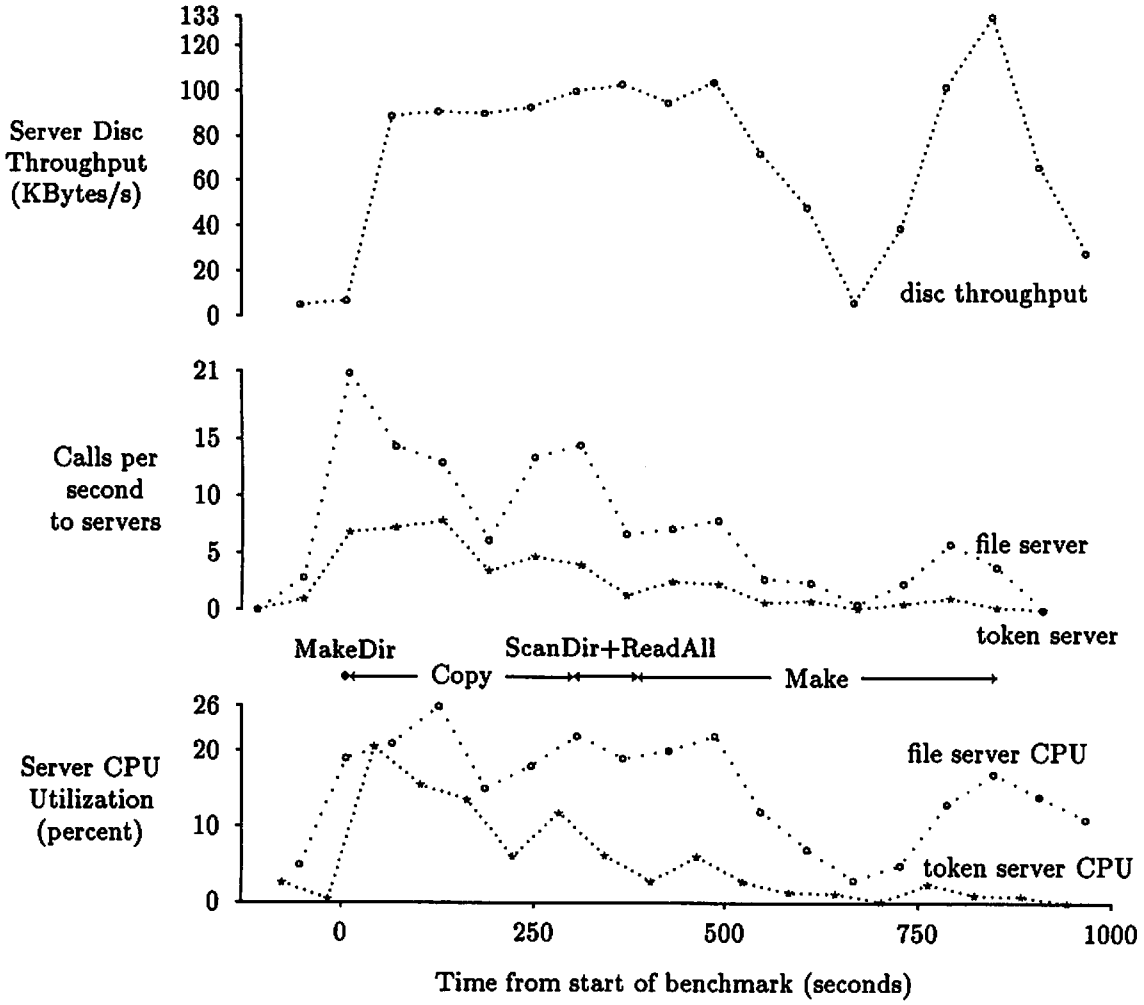
This table shows the server CPU utilization during the benchmark for various MFS loads. Two figures are given; one is the CPU load on the file server, the other is the CPU load on the token server. The figures in parentheses are standard deviations. The test conditions are given in Table 5.1.

Table 5.4: MFS Server Load during Benchmark

Table 5.4 shows the mean percentage of the CPU taken by the MFS file server and token server during the benchmark for various loads. The table shows that the average CPU load on the servers is low, even with 15 active clients. However, the experience with DID in Chapter 3 shows that mean load values are sometimes misleading, so more detailed loads were measured during a single run of the benchmark with 15 clients. The results are shown in Figure 5.3.

The graphs show the disc throughput in kilobytes per second, the number of RPCs to the file server and token server per second and the server CPU utilization averaged over each minute of the benchmark. The times for the various phases to run are also shown.

Unfortunately, disc utilization figures could not be obtained with standard utilities, and since source code for the file server operating system was not available, it was not possible to add the required instrumentation. Rough tests showed that the disc is able to transfer about 150 kilobytes a second when copying files between



The uppermost graph shows the variation of server disc throughput during one run of the benchmark with 15 clients running MFS. The middle graph shows the number of calls per second made to the token server and file server. The lowest graph shows the variation of file server and token server CPU utilization for the same run. All values are averaged over 60 second intervals.
The clients were all MicroVAX-IIs, as was the token server. The file server was a Sun-3/160 with a low-performance 70 megabyte winchester disc.

Figure 5.3: Server throughput and CPU utilization during benchmark for MFS

two local directories. There are likely to be more seeks when 15 machines are running the benchmark, so it is probable that the disc is saturated during the Copy phase. The higher peak during the Make phase corresponds to the writing of a few large files, which will involve fewer seeks per block transferred.

The disc throughput rises to a plateau during the Copy phase of the benchmark, and remains high for about three minutes after the Copy phase, even though the next two phases do not access the file server at all. This indicates that the delayed write system has spread the write traffic over a period of a few minutes, even though each client has a nominal write-back delay of only 30 seconds. Applications are able to force data to be written synchronously, but this facility is rarely invoked. Excessive delay in normal use could affect file system robustness in the face of client crashes if the cache is volatile. It is likely that the delay is due to the saturation of the file server disc. The clients seem able to overload the server even though only one cache manager thread is responsible for delayed writes. The application times are never seriously affected by the actions of the write-back thread. The CPU utilization of the client cache manager was measured to be 30 seconds for each run of the benchmark, which corresponds to about 4% averaged over the run. Lock conflicts between the application and the write-back thread seem to be rare.

If the backlog of write requests is due to disc saturation, this could be alleviated by faster discs and controllers, such as those used by the Andrew server. As with DID, the throughput of the file server could be improved by writing new data to a log, rather than over existing blocks. The reimplementation of the Cedar File System [Hagmann 87] has shown that even logging only file status and naming information can significantly improve local file system performance. Distributed file systems with effective client caches will naturally generate more write traffic than read traffic, so logging may become an increasingly important technique in file server design.

The middle graph shows the number of calls made to the file server and the token server during the benchmark. The number of calls to the file server is fairly high during the Copy phase, probably due to the large number of RPCs needed to read each file. Additionally, at least three calls are needed to write each new file, and this number could certainly be reduced by redesigning the file server interface. Even so, the number of calls is quite low, amounting to less than 30 per second to both servers during the busiest minute.

The CPU utilization figures for the file server and the token server are shown in the lowest graph. The CPU utilization of the token server is high during the early part of the benchmark, but drops as the benchmark progresses, even though the number of calls peaks after about 150 seconds. The high CPU utilization at the start of the run is surprising, but is probably due to memory allocation, and forking new threads. The token server is careful to reuse old data structures and threads for later RPCs to avoid the overhead of thread creation and memory allocation. The file server CPU utilization reaches a peak of 26% and is never close to saturation. The combined utilization on both servers is still well below 50% at all times, and even lower if the CPU power of the MicroVAX token server is scaled down to that of the Sun-3 file server.

5.3 Summary

MFS conforms well to the semantics of the UNIX file system. The prototype has difficulties with file timestamps, but there are easy solutions to most of the problems. The UNIX system call interface imposes various unexpected constraints on the file system. In particular, the need to return all file status information in one call, even though only one or two of the values are required causes some difficulty, since not all the information can be readily cached.

The granularity of sharing in MFS is the same as the local UNIX file system, even though the system has been tuned to work with large block sizes and sequential, non-shared file accesses.

The performance of MFS is comparable to that of Andrew when reading files from the file server, in spite of inefficiencies imposed by the NFS file server interface. The delay in accessing a file for the first time can be reduced by a factor between 2 and 3 by redesigning the file server interface, and combining it with the token server. Partial file transfer is important in limiting the latency imposed on each file transfer from the server, particularly when the file size grows beyond several tens of kilobytes.

When accessing cached files, MFS performs as well as a local file system, and benefits from the UNIX buffer cache. NFS and Andrew, which context switch to a user process whenever they access a new file, are significantly slower in this case.

The delayed write policy in MFS is effective in spreading bursts of file system updates over a longer period, and in reducing the amount of data written. During extended periods of heavy file server load it may be necessary to delay application programmes in order to avoid a backlog of write requests. File servers based on logging techniques may be needed to increase the throughput required by large numbers of clients with large file caches, since the majority of file server requests are writes.

The prototype has shown that the MFS design is practical, efficient, scales well as the number of clients increases, and preserves the semantics of the underlying operating system. Its performance in normal use is comparable with and often exceeds that of existing file systems which do not achieve all of these goals. The implementation can be improved in a number of ways which will increase the performance still further.

Chapter 6

Comparison with Related Work

This chapter compares MFS with existing distributed file systems.

6.1 Introduction

Distributed file systems are an important aspect of distributed systems as a whole. Consequently, there are many existing file system designs and implementations. I present a cross section of these, concentrating on those that allow a large amount of client caching.

The file systems compared in this chapter are:

- The Demand-Initialized Disc system (DID).
- The Cedar file system (CFS).
- Sun's Network File System (NFS).
- Apollo's Domain system.
- The Andrew file system.
- The Rochester file system (Roe).
- The Sprite network file system.
- The LOCUS file system.
- DEC's VAXcluster.
- AT&T's Remote File Sharing (RFS).

Table 6.1 shows some of the properties of the caching mechanisms of these file systems. The remaining sections of this chapter describe each of these systems in turn, comparing them with MFS, which was described in Chapters 4 and 5.

System	Consistency Algorithm	Write-sharing	Cache Unit	Write-back	Non-volatile Cache	Access in Partition
DID	no write-sharing	none	disc block	delayed writes	no	yes
Andrew	server callback on modification	sequential	whole file	on close	yes	yes
RFS	validate on open, disable caching when necessary	sequential, concurrent	file block	write-through	no	no
Sprite	validate on open, disable caching	sequential, concurrent	file block	delayed, callback	no	no
MFS	token passing, callbacks	sequential concurrent	file block	delayed, callback	potentially	yes
NFS	check every few seconds	none	file block	delayed, on close	no	no
CFS	immutable files	none	whole file	user control	yes	yes
Domain	validate on read, file locks	sequential	file block	delayed, unlock	no	no
VAX cluster	distributed lock manager	sequential, concurrent	file block	write-through	no	no
LOCUS	validate on open, token passing	sequential, concurrent	file block	delayed, callback	no	yes
Roe	full replicas, weighted voting	sequential	whole file	write-through	yes	yes

Table 6.1: Comparison of File Systems

The table characterizes caching file systems in several ways:

- the means by which the file system achieves consistency across multiple machines;
- how well the system deals with mutable shared files; (The terms sequential and concurrent write-sharing are defined in Section 1.2.)
- the unit of caching; some systems cache whole files, some partial files, while in others the caching is organized by disc block, rather than by file;
- when dirty data is written back to the file server;
- whether the cache data is lost during a client crash;
- whether the cache data is useful during network partition or server failure.

6.2 Demand-Initialized Discs

The Demand-Initialized Disc (DID) system [Burrows 85] is unlike any of the other file systems mentioned in that it deals with disc blocks and pays no attention to files. DID concurrency control operates at the level of the volume—write-sharing on individual files is simply prohibited. This is the case because the client operating

systems treat each volume as if it were a local disc. Each machine requires exclusive access to the disc when performing operations such as disc block allocation. The file servers ensure that each volume is available to only one client at any time, or is read-only to all clients.

Typically, each user has his own volume that is mounted on the machine he is currently using. Though simple and quite efficient, DID can be annoying to use because of the lack of write-sharing. Experience with a system of this type quickly shows that, although it can be used effectively, it is no substitute for a caching system that works at the file level rather than the disc block level. MFS allows sharing of mutable files down to the granularity of a single byte, and a single read or write system call, and so avoids the problems caused by the lack of write-sharing.

DID attempts to do a great deal of caching—a 40 megabyte cache is typical. The use of delayed writes and a large cache means that a DID client can often tolerate server reboots without users noticing any delays. Caches can safely be used during network partitions because write-sharing of files is impossible. With a cache of this size, data is usually read only once from a file server, so most file server traffic consists of writes of dirty data. Once a machine has cached most of the data needed for a particular session, it operates at speeds close to those observed on machines accessing only their local disc. Many users are only aware of the remote nature of the file system because of its restrictive sharing properties. The main effect on performance is the reduced speed observed just after a machine has rebooted, and to a lesser extent, after a user has just logged on. It seems reasonable that such effects will be observed by users of systems with large caches. MFS also makes use of large caches and delayed writes to protect the users from the delays associated with file server reboots. Cache sizes are similar, and so once again, files are rarely read more than once. Caching of partial files permits prefetching in the same way.

6.3 The Cedar File System

In the Cedar File System (CFS) [Schroeder 85], the cache consistency problem is eliminated for data by forcing all files to be immutable. Files are cached in their entirety and are typically read from the file server and written back under the explicit control of the user with a powerful version control system [Schmidt 82] intended to co-ordinate shared access to packages and groups of files. The use of immutable files means that the cache is useful even during network partitions. CFS is unusual amongst the distributed file systems described here in that its stylized use does not require mutable files. It would not be suitable for many existing applications and operating systems, which expect the more traditional type of file system found on most timesharing systems. It is difficult to compare CFS with MFS, because they address quite different problems. CFS is not able to support the write-sharing semantics offered by MFS, or support any kind of mutable file. MFS can offer the services provided by CFS, but at greater cost, since it would continue

to record consistency information about all files. CFS uses immutability to provide well understood semantics at low cost in terms of complexity, computation and communication. The implementors of CFS felt that it was better to optimize their file system for one particular purpose, and build other systems (e.g Alpine [Brown 85]), to handle the different problems of databases. MFS is an attempt to provide a more general purpose file system at reasonable cost.

6.4 Sun's Network File System

Sun's Network File System (NFS) [Sandberg 85] is a simple commercial file system. Though relatively unsophisticated, it has become popular with vendors of UNIX machines. Unfortunately, the protocol specification document [SUN 86a] discusses only the server RPC interfaces, and does not mention the conditions under which these routines are called. It is therefore unclear what guarantees the system makes about the consistency of the client caches.

In most NFS implementations, data is used directly from the cache if it has been validated within some time period, usually a few seconds. Cached data that is older than this can be refreshed by checking the file modification time at the server. This could be done periodically for all data in the cache, but typically it is done on demand, when the data is required by a client application. There are two reasons why NFS implementations do not attempt to revalidate the entire cache periodically:

- Client caches are typically small (less than a megabyte), which makes hits on old data less likely.
- The protocol allows only one cached file to be validated per RPC, so validating an entire cache would require many network accesses.

Dirty data may be flushed back to the file server at any time, and clients often write data back at different times depending on context. Typically, directory modifications are written back immediately, while modifications to files are written back when the file is closed, or after a few tens of seconds, whichever is the sooner. The resulting consistency guarantees are not particularly good, but inconsistencies are rarely observed in normal use. Concurrent write-sharing has no well defined effect, but sequential write-sharing usually works, failing only in cases where data is read on another machine less than a few seconds after it was written. This is usually not a problem, but it can cause difficulties with applications such as distributed compilers. These effects can be lessened by shortening timeouts, but this has the side effect of increasing network utilization and delaying file accesses that would otherwise have been satisfied from the cache immediately. As an increasing number of users have more than one machine at their disposal, this problem will become more and more significant. At the present time only a handful of users of NFS machines in this department have complained about the inconsistencies, though I expect this number to increase as distributed compilation and other applications become more commonplace.

NFS was designed with stateless servers on the grounds that this simplifies crash recovery. While this is true, it has crippled NFS's ability to maintain a consistent view of the file system with reasonable efficiency. There is no possibility of using callbacks to inform clients when file data changes, nor even to place locks on files to warn clients that a file is being updated. Additionally, it has made it impossible to permit concurrent write-sharing, which is needed for the efficient implementation of log files and databases. Also, separate ('*stateful*') services must be built to implement the file locking required by many clients. In contrast, MFS servers maintain a considerable amount of state to keep track of client cache status and thus minimize network utilization. The problems of crash recovery are real, but can be solved. MFS has shown that keeping track of server state is not a tremendous burden on the file server, particularly given that the overall effect is to dramatically lessen server load. The complexity of efficient recovery procedures must be weighed against lower performance, lack of scalability and weak consistency guarantees.

6.5 LOCUS

The file system of LOCUS [Walker 83] makes only moderate use of client file caching. LOCUS concentrates far more on the use of replication to increase file availability and performance. Remote files are accessed via a current synchronization site (CSS) that co-ordinates updates amongst the replicated storage sites and the clients. When a file is opened, its CSS is always contacted, and it is the CSS that controls a token passing mechanism that enforces single writer, multiple reader synchronization for file updates. Client machines hold tokens only while the file is open, and do not cache remote data when the file is closed. Dirty data is written back to storage sites immediately, and the token passing scheme allows other clients to determine when they can read new data. The consistency properties of this approach are good, but the CSS is an obvious bottleneck in a large scale implementation, since it is contacted each time the file is opened, rather than on the first open. The issues involved in replicating files are largely independent of caching mechanisms, so it seems likely that LOCUS could achieve higher performance and better scalability using a distributed file cache of the kind used in MFS.

LOCUS already allows and detects conflicting updates in a partitioned network, and a client may read a file if at least one storage site of that file can be contacted. MFS has not tackled the problem of replicated file servers, but it potentially allows each machine to read and write its cache while disconnected from the server. The present implementation of MFS has no well defined conflict resolution algorithm, and makes no guarantee that particular files will be available, as LOCUS does. This loose approach is useful for some applications during temporary server failure, but would be unacceptable if a server was unavailable for an extended period. In such cases, clients could either refuse to operate with possibly invalid data, or they could use application specific conflict resolution algorithms.

6.6 Apollo Domain

The Apollo Domain system [Leach 83] [Leach 84] uses a file system that is strongly coupled to the virtual memory system. Each object (file) in the Domain system consists of a series of pages on the object's home machine, which may be mapped into a process' address space on any machine in the network. Pages may then be read and written by accessing those areas of memory.

In contrast to Li's network virtual memory system [Li 86], Domain does not guarantee consistency across multiple machines. Instead, sufficient infrastructure is provided for clients to detect inconsistency when it occurs. Each object has a timestamp containing the time of its last modification at the home machine. Whenever a page is read from the home machine, the current object timestamp is also given to the client, which can then check that the timestamp has not changed since the previous read from the same object. In this way, a machine that wishes to read a single consistent copy of a file can detect writes that have been interleaved with its read requests. A similar scheme is used when pages are written back to an object; each dirty page is accompanied by the timestamp of the object to be updated. If the timestamp matches the object's current timestamp, the write proceeds and the new timestamp is returned. Otherwise, the write fails, indicating that the object was updated since the client last read a page from it. This scheme requires that each client read at least one page of an object before commencing to write the object. Also, it is tuned for a style of access that is typical of programme development using small files. Database access is fairly inefficient; there is only one timestamp per object, rather than one per page, which effectively prohibits the partial caching of shared, mutable files.

In addition to the concurrency control primitives described above, the Domain system provides locking primitives on objects, which may be used to ensure consistency. The designers have taken the view that most applications do not require locking, so these primitives are directly invoked by individual applications such as editors. Once again, the emphasis is on programme development and similar applications, since the locking primitives lock entire objects at a time, rather than individual pages.

The Domain system assumes that write-sharing of files is rare and does not provide consistency by default on the grounds of efficiency. It provides primitives that can be used to guarantee consistency for those few applications that require it, and to detect inconsistency for those applications that wish to generate an error message when it occurs. MFS takes the view that consistency can be guaranteed without great expense, and provides it by default. This allows application writers great freedom in choosing programme synchronization primitives. Any communications channel can be used for synchronization—it does not need to have side-effects on the file system. Unlike Domain, MFS allows partial file caching of files even in the presence of concurrent write-sharing. MFS is likely to be more efficient for implementing log files and low-performance databases, provided conflicts for blocks within the databases are rare.

6.7 The Roe File System

Roe [Ellis 83] uses the weighted voting algorithms described by Gifford [Gifford 79] to maintain many replicas of a file. Operations on files are enclosed in implicit transactions that update the number of copies of a file necessary for a quorum. The emphasis of Roe is on consistency of the file system, and this is traded against performance and availability in several ways.

Client caching is implemented by storing a full replica of a file on the client's local disc. From this point onwards, the client participates in the voting taking place between the replicas in the normal way. Since the local copy is guaranteed to be accessible to the client, the client will always read from the local copy immediately if the read quorum is one. Unfortunately, a read quorum of one implies that all copies must be available for update, which often will not be the case. In any case, we can be sure that at least one of the read and write quorums for a file will be greater than one. Therefore, we can be sure that clients will require synchronous contact with remote servers either when opening for read or when opening for write. Since all updates to multiple machines pass through the current *transaction manager site*, this machine is likely to become heavily loaded as the system size is increased. Even for system files, large scale client caching is not attractive because the number of replicas of the cached files would increase dramatically, and (assuming the read quorum was kept at one) this could lower the availability of the file for writing to nearly zero.

The Roe design is interesting, but it has features which prevent efficient, large scale caching of files. It can support high availability by replication, but it appears that the caching system of MFS can be used alongside such a system to provide the benefits of both. Roe provides good consistency guarantees at the expense of heavyweight transactions across replicas. MFS provides clients with the option of good consistency guarantees in the presence of crashes, or high availability in the presence of partition at the cost of consistency.

6.8 The Andrew File System

Of the systems described in this chapter, the Andrew file system [Howard 88] is the one most like MFS in its techniques for caching. The following list identifies the similarities between the two systems:

- File servers keep track of the files cached by clients and inform clients of file modifications.
- Clients can open and use cached information without contacting the server.
- The cache is held on disc and survives crashes.
- The cache is controlled by a user-space process that intercepts UNIX system calls.

Nevertheless, there are also several differences between the two:

- Andrew caches only whole files; MFS allows partial file caching.
- Andrew writes changes back to file servers on file close; MFS uses delayed writes and callbacks to control write-back.
- Andrew does not support concurrent write-sharing.
- Andrew client caches cannot hold dirty data during network partitions, since file close is synchronous at the file server.
- Andrew intercepts only open and close calls; MFS intercepts all reads and writes.
- Andrew intercepts system calls even when accessing cached files, causing extra context switches.

Andrew is tailored to a teaching environment, and works well within that environment. It demonstrates good scaling properties and fair performance. Comparison with the prototype Andrew system [Howard 88] [Satyanarayanan 85] demonstrates that server callbacks can provide scalability and performance that is superior to systems that contact file servers on each open.

The disadvantages of caching whole files are not noticeable while files remain small, but become apparent when file sizes increase beyond a few tens of kilobytes. The first observed effect is the latency in reading remote files, since the whole file is fetched before the application can see any of the data. Partial file caching with prefetching has some small overhead in software complexity, but can increase effective performance dramatically for large files. MFS is currently unable to demand page executables into its cache as they are executed, but this would be possible with changes to the virtual memory system of the UNIX kernel. The importance of such measures should not be underestimated, since they can have great effects on the startup times for large interactive programmes. Many programmes now have executable images measured in megabytes, and even popular editors can be as large as 0.7 megabytes. Systems such as NFS read files at about 0.2 megabytes per second (measured between two unloaded Sun-3 machines on a single 10 megabit/s Ethernet). Even the most efficient file transport mechanisms will not achieve speeds exceeding this by more than a small factor, assuming the current generation of popular 10 megabit/s networks. These figures indicate that large programmes will necessarily experience a noticeable startup delay in a system that cannot partially cache executable files. Many large programmes use comparatively few pages during their startup sequences, and this property can be used to reduce startup delay to a more reasonable amount. Although we can expect faster networks, faster network interfaces and faster CPUs, it seems likely that these problems will continue to be noticeable for some years to come.

In addition to the delay of fetching large files, whole file caching has the disadvantage of limiting the total size of every file in the system to the size of the smallest cache of any machine in the system. (An alternative, but equally distasteful view of this problem is that some machines are unable to read large files). Of course, users can split their files into smaller units to avoid the problem, but this is a rather unattractive solution.

Andrew writes all dirty data back to file servers on close. This permits sequential write-sharing to take place and improves the robustness of the file system in the case of client failure. It has the side effect of generating unnecessary file server traffic by writing back temporary files that are about to be deleted. This effect has been reduced substantially in Andrew by using temporary file directories that are local to each client. This has the disadvantage that different parts of the file system behave in different ways as far as the user is concerned. Moreover, routine sharing of temporary files is made more complicated. Delaying writes by 30 seconds can reduce writes to the file server by a quarter [Ousterhout 85]. The corresponding loss in robustness is no worse than that already tolerated by UNIX users, whose writes are already delayed by 30 seconds in the UNIX buffer cache. Of course, applications that require additional guarantees can make explicit system calls to ensure that data is written to non-volatile storage.

By not supporting concurrent write-sharing, Andrew prevents the use of log files and databases. These services must be provided by other, independent services which will be separate from the file system and, unless work is done to integrate them, accessed in a different way. Moreover, this may reduce the ease with which certain applications can be ported from a non-distributed environment. Even in an environment where concurrent write-sharing is used infrequently by applications, its omission can complicate process migration in operating systems that support it, since the migration procedure must ensure the consistency of the file system between the source and target machines at an arbitrary point in the running of the process.

Since Andrew contacts the file server each time a modified file is closed, it is harder to shield the user from brief periods of file server down time, even though data can be read from the cache during such periods. MFS permits the user to continue writing to its local cache without contacting the file server, if the user is prepared to tolerate the uncertainty associated with this option. In cases where client crashes are relatively rare and conflict over updates is unlikely, this will provide a noticeably better service when contact has been lost with the file server. A better way to solve this problem is by means of file replication, but this cannot help the situation where the client is connected to the rest of the network by an unreliable gateway prone to short periods of down-time or congestion.

Andrew, then, is specifically aimed at the teaching environment, and, while it is well suited to that environment, its design prevents its use in a more general setting, where databases, log files, fine-grain file sharing or process migration are commonplace. MFS overcomes these problems at the expense of additional file server state, but the amount of additional state is minimal when these facilities are not being used.

6.9 The Sprite File System

The Sprite file system [Nelson 88] is similar to MFS in that it provides strong consistency guarantees, even in the presence of concurrent write-sharing. Its strategy for achieving this is quite different however. The most important difference is that Sprite clients always contact the file server on open, even if the file already resides in the client cache.

Sprite's strategy is as follows. On each open, the client contacts the server and informs the server of its intent to read, write or both. The server keeps track of which clients have the file open at any time, so it is able to detect the case when concurrent write-sharing is possible. If the server finds that the file will be open by multiple clients and written by at least one of them, it disables all caching for that file, forcing the clients to perform all reads and writes synchronously at the file server. If necessary, the server instructs each client to flush dirty data and discard data cached from the file. Once the file has entered this mode, it remains uncached until the file is no longer open anywhere. This last condition is used simply because it is easier for the server to detect than the condition that concurrent write-sharing is no longer taking place.

In addition to disabling caching on detecting concurrent write-sharing, the server allows sequential write-sharing by keeping a version number for each file, in a similar way to the Domain system. On each open, the client obtains the current version number of the file and compares it with its cached version number. If there is a mismatch, the client discards all cached information associated with the file. In order to enable delayed writing of data to files, the server remembers the last client to open the file for writing, and instructs that client to flush dirty data before further opens are permitted.

The performance measurements for Sprite are impressive, but this is partly due to the way it has been implemented, rather than the caching algorithms used:

- Sprite's file system is implemented entirely in kernel-space, which reduces the number of context switches necessary.
- The operating system and file system were developed together, and can be expected to be well integrated with one another.
- Sprite uses a fast communications mechanism as one of its basic primitives, which reduces the overhead of multiple calls to a single server.

A similar implementation of MFS, or the other systems being considered, would result in performance improvements that would make comparison easier.

The protocols used in Sprite rely on the file server being contacted at each open of a file. This indicates that Sprite is unlikely to scale as well as MFS or Andrew, and the figures given by Nelson support this. The cache validation strategies allow a reasonable amount of caching, but there are a number of features which limit the extent to which caching can be used. In particular, caching is entirely disabled during concurrent write-sharing. Although this is a relatively rare occurrence in many environments, this feature of the file system may discourage

the use of concurrent write-sharing because of the performance penalty. Even during sequential file sharing, Sprite fails to retain cached information about parts of files that were not updated, because its version mechanism treats the whole file as a single entity. This would tend to penalize operations such as examining large log files, since one update would cause any cached information to be lost. MFS allows partial file caching in both of these situations, and so can provide a useful performance improvement for applications involving large databases or append-only files, at the cost of some extra complexity in the file server. If individual blocks of a file were heavily write-shared by a database application, the performance of Sprite would exceed that of MFS, since Sprite avoids token passing.

6.10 Remote File Sharing

AT&T's Remote File Sharing (RFS) [Bach 87] is remarkably similar to the Sprite file system. Both systems contact the file server on open, maintain file version numbers to determine when cached data is invalid, and disable caching whenever concurrent write-sharing is taking place. The most significant difference is that RFS never caches dirty data; instead, all modifications are written through to the file server. This strategy improves the robustness of the system, at the expense of writing some fraction of data that would otherwise have been deleted without ever going to disc. The write-through strategy is hard to justify on the grounds of robustness, since RFS is used with systems that typically delay local disc writes by 30 seconds.

A less important difference is the algorithm used for disabling and re-enabling caching. Sprite disables caching as soon as conflicting opens are detected, but RFS disables caching only when the first write operation is performed. RFS re-enables caching when the writing process closes the file, or when there have been no writes for some period. Sprite takes the simple approach of disabling writes until the file is no longer open by any process.

6.11 VAXclusters

Digital's VAXcluster[1] uses a distributed lock manager to maintain consistency across its file caches. [Kronenberg 87] [Snaman 87] [Goldstein 87] VAXclusters consist of a small number (typically two or three) of similar machines, connected by a fast, purpose built network.[2] The machines in the cluster co-operate to provide a single distributed operating system, each allowing all the members of the cluster to access its discs. Every machine in a cluster trusts every other machine to perform access checks in the same way, which makes some aspects of the design slightly simpler than they would have been in a more general distributed system.

[1]The description given here applies to major release 4 of the VMS operating system.

[2]In fact, some clusters are connected via Ethernet.

The distributed lock manager is a very powerful, general-purpose locking system. Locks may have a number of states, ranging from unlocked to an exclusive lock. They have textual names, which may be structured hierarchically to provide trees of locks, also known as *resource trees*. As well as non-blocking error returns, the lock manager can provide callbacks to clients, either when a lock is released, or when a conflicting lock is requested. Each lock also has an associated value block, which can be used to store version numbers. The system is distributed and fault tolerant—when a machine enters or leaves the cluster, a reconfiguration takes place in which each machine reclaims the locks it had before the event. Each machine is trusted by the others, so there is no question of fraud when locks are reclaimed.

The first machine to lock any part of a resource tree after a reconfiguration becomes the manager of the resource. This machine maintains a fixed size data structure for the each sublock in the tree until no more locks are held, or another reconfiguration takes place. There is a special lock state, *no lock*, which does not conflict with any other state, but maintains an interest in the lock and ensures that its data structure is not deleted. When a machine first wants to lock a resource, it first hashes the name of the resource to find the *directory node* for that resource. This machine holds the name of the current manager of the resource. Once a machine has locked some part of a resource tree, it can cache the name of the resource manager and so is able to lock other parts of the tree without contacting the directory node.

The file system locking structure is very complicated; most operations require several different file system structures to be cached and locked. Directories, quota files, free space bitmaps and file headers are all treated specially, but data blocks are handled by a simple mechanism. Disc blocks are organized into groups, each of which is protected by a single lock, with a version number in its value block. Whenever an exclusive lock is obtained, indicating that the blocks are to be updated, the version number is incremented. When a lock is obtained for reading, the version number is compared with the cached version number to see if the blocks have been updated since they were last read. If so, cached blocks are deleted. When data is not being accessed, but is still in a machine's cache, the lock is set back to *no lock* state to ensure that the version number in the value block is maintained. The lock can be deleted entirely when all the file's blocks have been removed from all caches. No callbacks are used for the file data cache in the current version of the VAXcluster software, even though the lock manager supports them. This forces the caches to be used in a write-through mode, which lowers performance slightly.

Normally, a machine that accesses a file will become the resource manager for the file. However, if the file is shared, there is some chance that some of its blocks may still be cached on another machine. If cache sizes are large, and one access occurs soon after another, the management of a resource is not transferred to the machine accessing the resource. This forces machines to use the remote lock manager, even if there are no further shared accesses to the file.

A more severe problem is that of concurrent sharing. In this case, each access requires contact with the resource manager, even when no writing is taking place.

It may be possible to avoid this problem by accessing files under *arbitration locks* that preclude write-sharing; unfortunately it is not clear that these arbitration locks interact with the cache consistency mechanism in a way that achieves this. In any case, it is undesirable to apply over-restrictive locking policies in order to achieve consistency.

VAXclusters are more tightly coupled than a distributed system using MFS. VAXcluster algorithms work well for a small number of machines and styles of use encouraged by the VMS operating system, but are likely to perform less well if the cache sizes, cluster sizes or number of distributed applications increase dramatically. MFS reduces communication in such situations by greater use of callbacks than the current version of the VAXcluster file system.

6.12 Other Work

Techniques similar to those used in MFS are the subject of distributed file system research at the Digital Equipment Corporation Systems Research Center [Mann 88]. The aims of this project are to provide a high degree of caching, high file availability and reasonable consistency semantics.

Other distributed file systems exist, but pay little attention to client caching, or are similar to those already discussed. Among them are the Newcastle Connection [Brownbridge 82], WFS [Swinehart 79] and the Cambridge File Server [Birrell 80], which can be used via a separate *cache server*, known as the Tripos Filing Machine [Needham 82], but has no client caching. The caching mechanism in Hewlett-Packard's DUX system [Hwang 87] is similar to that of Sprite and RFS.

6.13 Summary

The literature contains a wide range of file system designs, with many policies for allowing client caching. There are many schemes for maintaining cache consistency across machines:

- preventing write-sharing (DID);
- immutable files (CFS);
- version number checking (Sprite, RFS, Domain, VAXclusters);
- locking (Domain, LOCUS, VAXclusters);
- disabling caching during sharing (Sprite, RFS);
- callbacks on modifications (Andrew, MFS);
- callbacks to flush dirty data (Sprite, RFS, MFS);
- approximate solutions (NFS);

- weighted voting and transactions (Roe).

The technique employed in MFS requires more server state than the other systems described, but is the only system to exhibit all of the following properties:

- low server utilization and good scaling (CFS, Andrew, MFS);
- sequential and concurrent write-sharing (Sprite, RFS, VAXclusters, LOCUS, MFS);
- partial file caching (DID, NFS, Sprite, RFS, VAXclusters, LOCUS, Domain, MFS);
- partial file cache invalidation (VAXclusters, MFS);
- cache potentially useful during server failure (DID, CFS, Andrew, MFS).

In situations where these properties are useful, the additional complexity of storing and manipulating additional server state seems to be worthwhile. MFS therefore seems to have good properties for use as a general purpose file system, and to differ from other existing file system designs in several fundamental ways.

Chapter 7

Conclusion

This dissertation has presented a design for a distributed file system with client caching. This chapter summarizes the main conclusions, and suggests possible further work.

7.1 Summary

Most existing distributed file systems fail to provide efficient solutions to the problems of shared, writable data, under the assumption that it is too costly. Although rare in the programme development and teaching environments, there are examples of shared, writable data, and they are likely to become more common, unless write-sharing is discouraged by file system implementations.

Large client caches can improve the performance of a distributed file system until it approaches that of a local file system. They can eliminate file server accesses for most read requests and a significant number of write requests. The result of effective client caching is that most requests received by file servers are write requests. Clients can reduce the amount of write traffic by delaying writes of new data in the hope that it will be deleted soon before it is written to the file server. File server interfaces should be designed to minimize the number of calls needed for each file access, and should be optimized for writing, possibly by using logging or fast non-volatile storage to reduce the number of disc seeks and increase overall throughput.

The file system presented in preceding chapters exhibits reasonable performance and good scaling characteristics, while preserving cache consistency for individual reads and writes of a single byte. The MFS prototype has shown that neither performance nor functionality need be sacrificed in building a distributed file system.

7.2 Further Work

The system described in Chapter 4 and Chapter 5 is a prototype. Although the measurements taken under artificial loads are encouraging, experience with

actual use is needed to identify where the system could be improved. A second implementation could benefit from:

- integration of the token server and file server;
- an improved file server interface;
- improved file server performance for writing files;
- closer integration of the client cache manager and the local file system;
- modifications to the programming interface to avoid the problems of fetching file status information unnecessarily;
- heuristics for prefetching data before it is referenced, such as prefetching all the files in a directory that has been listed.

Current work in the Computer Laboratory includes an investigation of better file server interfaces, and of techniques for improving the performance of file servers such as non-volatile memory and logging.

Although MFS supports the notion of accessing cached files when the file server cannot be contacted, further experimentation is needed to develop algorithms for prefetching essential files. In addition, techniques developed in other systems for resolving update conflicts must be integrated with the caching system if this mechanism is to be made reliable.

References

The pages on which each reference is cited are listed in parentheses after the reference.

[Archibald 84] J. Archibald and J.-L. Baer. An Economical Solution to the Cache Coherence Problem. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 355–362, IEEE, 1984. (38)

[Archibald 86] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986. (38)

[Babaoglu 81] O. Babaoglu and W. Joy. Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 78–86, December 1981. (61)

[Bach 87] M.J. Bach, M.W. Luppi, A.S. Melamed, and K. Yueh. A Remote File Cache for RFS. In *Proceedings of the USENIX Summer 1987 Conference*, pages 275–280, June 1987. (xiv, 5, 32, 89)

[Barbara 86] D. Barbara, H. Garcia-Molina, and A. Spauster. *Increasing Availability under Mutual Exclusion Constraints with Dynamic Vote Reassignment.* Technical Report CS-TR-056-86, Princeton University, November 1986. (51)

[Birrell 80] A.D. Birrell and R.M. Needham. A Universal File Server. *IEEE Transactions on Software Engineering*, SE-6(5):450–453, September 1980. (91)

[Braunstein 88] A. Braunstein. *File System Design in Computers with Very Large Physical Memories.* Master's thesis, MIT, Dept of Electrical Engineering and Computer Science, 1988. (32)

[Brown 85] M. Brown, K. Kolling, and E. Taft. The Alpine File System. *ACM Transactions on Computer Systems*, 3(4):261–293, November 1985. (3, 82)

[Brownbridge 82] D.R. Brownbridge, L.F. Marshall, and B.Randell. The Newcastle Connection, or, UNIXes of the World Unite! *Software Practice and Experience*, 12:1147–1162, 1982. (91)

[Burrows 85] M. Burrows. Using a Local Disc as a Block Cache. November 1985. Cambridge Computer Laboratory Internal Note. (80)

[CCITT 87] CCITT. Directory—Authentication Framework. CCITT Draft Recommendation X.509 (version 7), November 1987. (49)

[Censier 78] L.M. Censier and P. Feautrier. A New Solution to the Coherence Problems in Multicache Systems. *IEEE Trans. Computers*, C-27(12):1112–1118, December 1978. (38)

[Ellis 83] C.A. Ellis and R.A. Floyd. The ROE File System. In *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems*, October 1983. (85)

[Gifford 79] D. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, December 1979. (51, 85)

[Goldstein 87] A.C. Goldstein. The Design and Implementation of a Distributed File System. *Digital Technical Journal*, (5):45–55, September 1987. (89)

[Greenwald 85] M. Greenwald et al. Remote Virtual Disc Source Code. Supplied by the MIT Athena Project, 1985. (17)

[Hagmann 87] R. Hagmann. Reimplementing the Cedar File System using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 155–162, November 1987. (77)

[Howard 88] J. Howard et al. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), 1988. (xiii, 5, 22, 25, 29, 71, 73, 74, 85, 86)

[Hwang 87] C. Hwang, J. Tesler, and C. Lin. Achieving the One-System view for Distributed UNIX Operating Systems. In *Uniforum 1987 Conference Proceedings*, 1987. (91)

[Johnson 85] M.A. Johnson. Private Communication, 1985. (17)

[Kazar 88] M.L. Kazar. Synchronization and Caching Issues in the Andrew File System. In *Proceedings of the USENIX Winter 1988 Conference*, pages 27–36, February 1988. (63)

[Kowalski 78] T.J. Kowalski. *FSCK - The UNIX System Check Program.* Technical Report, Bell Laboratories, Murray Hill, NJ 07974, March 1978. (34)

[Kronenberg 87] 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. (89)

[Lamport 87] L. Lamport. *Synchronizing Time Servers.* Technical Report 18, DEC SRC, June 1987. (69)

[Leach 83] P.J. Leach et al. The Architecture of an Integrated Local Network. *IEEE Journal on Selected Areas in Communications*, SAC-1(5):842–857, November 1983. (84)

[Leach 84] P.J. Leach et al. The Architecture and Applications of the Apollo Domain. *IEEE Computer Graphics and Applications*, April 1984. (84)

[Li 86] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, 1986. (84)

[Mann 88] T. Mann. Private Communication, July 1988. (91)

[NBS 77] National Bureau of Standards. *Data Encryption Standard.* Technical Report 46, U.S. Dept of Commerce, Washington DC, January 1977. Federal Information Processing Standard publication. (49)

[Needham 82] R.M. Needham and A.J. Herbert. *The Cambridge Distributed Computing System. International Computer Science Series*, Addison-Wesley Publishing Company, 1982. (17, 91)

[Needham 88] R.M. Needham and M. Burrows. Locks in Distributed Systems – Observations. *Operating Systems Review*, 22(3):44, July 1988. (42, 43)

[Nelson 88] M.N. Nelson, B.B. Welch, and J.K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1), 1988. (xiv, 3, 5, 32, 88)

[Ousterhout 85] J.K. Ousterhout et al. A Trace Driven Analysis of the 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 15–24, December 1985. (12, 14, 87)

[Paxton 79] W.H. Paxton. A Client Based Transaction System to Maintain Data Integrity. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 18–23, December 1979. (51)

[Ritchie 78] D.M. Ritchie and K. Thompson. The UNIX Time-sharing System. *Bell System Technical Journal*, 57(6), July 1978. (3)

[Sandberg 85] R. Sandberg et al. Design and Implementation of the Sun Network File System. In *Proceedings of the USENIX Summer 1985 Conference*, pages 119–130, June 1985. (xiv, 5, 28, 82)

[Satyanarayanan 81] M. Satyanarayanan. A Study of File Sizes and Functional Lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 96–108, December 1981. (13)

[Satyanarayanan 85] M. Satyanarayanan et al. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, December 1985. (86)

[Schmidt 82] E.E. Schmidt. *Controlling Large Software Development in a Distributed Environment.* Technical Report CSL-82-7, Xerox PARC, December 1982. (4, 81)

[Schroeder 85] M.D. Schroeder, D.K. Gifford, and R.M. Needham. A Caching File System for a Programmers Workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 25–34, December 1985. (xiii, 1, 3, 81)

[Sidebotham 86] R.N. Sidebotham. Volumes: The Andrew File System Data Structuring Primitive. In *The European UNIX User Group Conference Proceedings*, August 1986. Also available as Technical Report CMU-ITC-053, ITC, CMU. (35)

[Smith 81] A.J. Smith. Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering*, SE-7(4):403–417, July 1981. (13)

[Smith 85] A.J. Smith. Disk Cache—Miss Ratio Analysis and Design Considerations. *ACM Transactions on Computer Systems*, 161–203, August 1985. (13)

[Snaman 87] W.E. Snaman Jr. and D.W. Thiel. The Vax VMS Distributed Lock Manager. *Digital Technical Journal*, (5):29–44, September 1987. (89)

[SUN 86a] *Network File System Protocol Specification.* b edition, February 1986. (40, 82)

[SUN 86b] *Remote Procedure Call Protocol Specification.* b edition, February 1986. (40)

[Swinehart 79] D. Swinehart, G. McDaniel, and D. Boggs. WFS: A Simple Shared File System for a Distributed Environment. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 9–17, December 1979. (91)

[Tang 76] C.K. Tang. Cache System Design in a Tightly Coupled Multiprocessor System. In *Proceedings of the 1976 AFIPS National Computer Conference*, pages 749–753, AFIPS, 1976. (38)

[Thompson 78] K. Thompson. UNIX Implementation. *Bell System Technical Journal*, 57(6), July 1978. (19)

[Treese 88] G.W. Treese. Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3bsd. In *Proceedings of the USENIX Winter 1988 Conference*, pages 175–182, February 1988. (xiv, 17, 32, 38)

[Walker 83] B. Walker et al. The LOCUS Distributed Computing System. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–70, October 1983. (64, 83)

[Wheeler 87] D.J. Wheeler. *Block Encryption.* Technical Report 120, University of Cambridge Computer Laboratory, November 1987. (49)