1

Micro-D

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of production possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

Decentralized Management of Distributed File Systems

by

Amiyaranjan Nayak, B.Math.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfilment of the requirements
for the degree of
Master of Computer Science

Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University

© December, 1985

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-40625-9

The undersigned hereby recommend to the Faculty of Graduate Studies and Research acceptance of the thesis,

"Decentralized Management of Distributed File Systems"

submitted by Amiyaranjan Nayak, B.Math., in partial fulfillment of the requirements for the degree of Master of Computer Science.

Thesis Supervisor

Chairman,
Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

April, 1986

Abstract

Distributed computer systems have been the subject of a vast amount of research. In a distributed computer system, a file system is an integral part of a network operating system, distributed operating system, or a distributed database system. The main objective of this thesis is to identify desirable properties and features of file systems, and to survey existing file systems with respect to these properties and features, highlighting the differences in the different approaches. Noticing that none of them allows for truly decentralized management of the data, a file system which is known as the DFMS (for "distributed file management system") is proposed which preserves many of the characteristics of the others; unlike the others, it allows total "dispersion" of the data and exhibits fully and truly decentralized behaviour.

ACKNOWLEDGEMENTS

First and Foremost, I wish to express my sincere thanks and appreciation to my thesis supervisor, Professor Nicola Santoro, whose continued guidance, support and patience made this thesis a manageable effort.

I am grateful to my family members who showed their support of my efforts with continued interest and encouragement.

I wish to thank Dr. R. Pandya, Mr. D. Thomas and Dr. F. Hadziomerovic for their comments and suggestions on my work.

I am also thankful to Professor Morteza Niktash, Glenford Mapp and Dr. Anup Dewanji for discussions about the thesis in general.

Finally, I would like to thank Jayendra Kumar Doshi for proofreading this thesis.

Table of Contents

Abstract	iii
Acknowledgements	iv
Table of Contents	v
Chapter 1 Introduction	
1.1 Background and Motivation	1
1.2 Major Contributions of the Thesis	3
1.3 Organization of Thesis	4
Chapter 2 General Issues of a Distributed File System	
2.1 Introduction	5
2.2 Evolution	6
2.3 General Issues	7
2.3.1 Object Naming	7
2.3.2 Directory Management	8
2.3.3 Access Mechanism	9
2.3.4 File Sharing and Protection	10
2.3.5 Reliability	11
2.3.6 Internal Organization	12
2.4 Conclusion	12
Chapter 3 Existing Approaches	
3.1 Introduction	13

3.2	Backgrounds of Existing File Systems	14
3.2.1	Woodstock File Server	14
3.2.2	Felix File Server	14
3.2.3	Cambridge File Server	15
3.2.4	Xerox Distributed File Server	15
3.3	Primary Design Objectives	16
3.4	Approaches Towards General Issues	16
3.4.1	Object Naming	16
3.4.2	Directory Management	17
3.4.3	Access Control	18
3.4.4	File Sharing	19
3.4.5	Reliability and Protection	21
3.5	Comments on Implementation	28
3.6	Comments on Performance	25
3.7	Conclusion	27

Chapter 4 Proposed DFMS

4.1	Introduction	30
4.2	Motivation	31
4.3	Design Objectives	33
4.3.1	Access Mechanism	33
4.3.2	Virtual Memory	33
4.3.3	Data Sharing	33
4.3.4	Directory Management	34
4.3.5	Protection Mechanism	34
4.4	Functional View of the DFMS	37

4.5	File Server Description	40
4.5.1	Structural View	40
4.5.2	Primary and Secondary Sites	41
4.5.3	File Structure	42
4.5.4	File Server Interface	43
4.5.5	Internal Organization	47
4.6	Directory Server Description	51
4.7	Discussions on General Issues	52
4.8	Conclusion	54

Chapter 5 Complexity and Performance Issues

5.1	Introduction	55
5.2	Objectives	56
5.3	Strategies	58
5.3.1	Random Strategy	58
5.3.2	Best-Fit Strategy	58
5.3.3	Pass-Next Strategy	59
5.4	Comparison Between Strategies	60
5.5	Compaction	64
5.5.1	Decentralized Compaction Algorithm	67
5.6	Performance Prediction	69
5.6.1	The Model	69
5.6.2	Calculation of P_t	74
5.6.3	Accessing a Page	76
5.6.4	Opening a file	76
5.7	Conclusion	76

Chapter 6 Conclusion

Appendix A	83
Glossary	90
References	92

Chapter 1

Introduction

1.1 Background and Motivation

A distributed system can be minimally defined as a collection of processing elements (e.g., processors) connected through a communication medium (e.g., bus). Computations (e.g., processing user defined queries) are carried out through the cooperative effort of some (possibly all) of these elements giving the user the illusion of dealing with a monolithic system; thus, each computation might require both local processing and communication activities. Depending on how the coordination of a computation is organized, it is possible to distinguish between centralized and decentralized control within the system. In a centralized environment, a request for a computation is always referred to a special entity (the central controller) which will be responsible for the execution (although it might not be performed by it). A system where different controllers exist for distinct computations will still be considered centralized. In a decentralized system, where no central controller exists, all processing entities involved in a computation are considered equal. To make a decision, it requires some form of agreement between the involved entities.

Because an agreement is required, a system with decentralized control will be, in general, slower to reach a decision than a centralized one where decisions can be made immediately by the central controller. However, a centralized system is highly vulnerable: a failure of the central controller will cause a failure of the

entire decision system; in other terms, the availability of the system to perform a computation equals the reliability (both hardware and software) of the central controller. This situation must be contrasted with the high availability of a system with decentralized management: in case of a processor failure, the computation could still be performed by the remaining processors.

The use of distributed computer system has become increasingly widespread, due to several factors, including the price-performance revolution in microelectronics, the development of cost effective and efficient communication subnets, the development of resource sharing software and the increased user demand for communication. Such systems might provide significant advantages, including good performance, good reliability, good resource sharing and extensibility. This explains the ever increasing research interest in fields such as distributed operating system, distributed real-time systems, communication subnet, local area networks, concurrent and distributed programming languages and distributed databases.

An integral part of any database system or operating system is the file system; a fact still holds when the system under consideration is distributed. In most distributed systems, the file systems are generally viewed as servers which process requests originating from users as well as from other components of the system. The collection of these file servers create the illusion that there is a single logical file system when, in fact, there may be many different file systems. Depending on the level of sophistication required, the file server may support replication, movement of files across the system, and maintenance of the integrity of the data following updates to files. Although the file servers may allow for distribution of the data (e.g., a file) across several sites, only some of the existing systems exhibit this property; however, their management of these servers is purely centralized, and

thus highly vulnerable. It is therefore important to ask whether it is possible to design a truly decentralized management of distributed file servers, study which properties such a system should have, and investigate which problems must be faced at design and implementation level. Aim of this thesis is to address these questions and provide some reasonable answers.

1.2 Major Contributions of the Thesis

The first contribution of the thesis is the identification of a class of problems inherent to distributed file systems and the study of how these problems are addressed in some of the existing systems. The class of problems identified includes: possible inconsistencies due to file sharing, chaotic situations due to file naming and access mechanism, deadlock situations due to possible interleaved transaction processing, and unreliability due to failures of all kinds. The file systems considered are : Cambridge File Server (CFS), Woodstock File Server (WFS), Felix File Server (FFS) and Xerox Distributed File Server (XDFS).

Noticing that none of these existing file servers allows for a truly decentralized management of the data, a new file system is proposed. This system, called DFMS (for "distributed file management system") is shown to preserve many of the characteristics of the existing ones; unlike the others, it allows for total "dispersion" of the data and exhibits fully decentralized behaviour. The system is composed of a set of "file servers", each with a limited amount of page-based memory, and of a "directory server" which implements the usual directory functions. Requests for creating a file are sent directly to the directory server which will assign the request to a file server; from this point on, this file server becomes responsible for any operation on that file. Requests to perform an operation (e.g., open, close,

read, write) are communicated directly from the user to the file server. Since the local memory available at a file server is limited, it is possible that no space is available locally to store part of the file; in this case, the file server finds another server with available space from which it "borrows" the needed space. In this process of lending and borrowing, some of the files remain scattered in the system, but the process remains transparent to the users. Since the scattering of files induces an additional level of inter-node communication, thus, affecting the overall system performance, both allocation strategy and compaction become important. The final contribution of this thesis is the study of such issues in the proposed DFMS and the characterization of the performance. Several allocation strategies are considered, and the use of different approaches to compaction is studied.

The proposed approach does not obviously provide the definite answer to the questions which have motivated this thesis; it does, however, shed some light on the nature of the problem and provide some directions for further research.

1.3 Organization of Thesis

Chapter 2 discusses the general issues, such as object naming, access control, directory management, file sharing, reliability and protection, pertaining to the design of a distributed file system. Chapter 3 surveys four existing file servers, such as Felix file server, Cambridge file server, Woodstock file server, and Xerox distributed file server, to describe the approaches taken by each of them in addressing the general issues. In Chapter 4, a new distributed file server, known as the DFMS, is proposed, and its characteristics discussed. Chapter 5 gives a discussion on various management strategies, compaction algorithms, and performance issues.

Chapter 2

General Issues of a Distributed File System

2.1 Introduction

In the past decade, the growth of communication networks, particularly local area networks, has fostered research into the area of distributed processing. The widespread use of distributed system is due to the development of cost effective and high-bandwidth communication network, the development of resource sharing software, and the increased demand for communication, economical sharing of resources and productivity. A distributed system potentially provides significant advantages, including good performance, good reliability, good resource sharing and expandability. Traditional file systems which used to be an integral part of any operating system are being separated and set aside for common use by many computer systems in a communication network.

Section 2.2 discusses briefly the history of file servers and the evolution of distributed file systems from the simplest form of filing system which is called a backing store server.

Section 2.3 discusses the general issues such as object naming, directory management, file sharing, protection and reliability involved in the design of a distributed file system (distributed file server, as sometimes referred to).

2.2 Evolution

With the development of local communication networks, the need for having a "dedicated computer" which could provide mass storage to a number of other computers was realized. Since then, such an utility has been implemented with an increasing degree of complexity and sophistication.

In its simplest form, we have the backing store server [Birr 80] whose primary function is to control a physical storage medium (typically, disks). In the backing store server, the user had the responsibility of imposing any kind of structure desired on the data. The concept of file was not quite there. The user also had the freedom of choosing access control and providing desired form of symbolic name of a file and directory structure.

Unlike the backing store server, the file server is very much like a conventional filing system in a time-sharing computer and provides more sophisticated high-level interfaces. The concept of user and file is well established. Above all, such a device supports the facilities for access control and protection on files, for directory structure containing groups of files, for the lookup of textual names of files and for interlocks on simultaneous access to files [Birr 80].

Higher in the evolution scale, there is a solution that can provide flexibility, reliability and incremental expansion : distributed file system (or distributed file server). The basic idea is to distribute the file management function over a co-operating set of "server computers" which together create the illusion of a single, logical system [Stur 80]. Such a file system could support a number of different host operating systems, each having an added software to meet the network operating system standards. With the current technology, a single server with one disk

drive might hold up to 80 Mbytes of data. The use of rather small computers as file servers permits a wide range of file storage capacity because it is possible to expand from a single minicomputer with one disk drive to many servers, each with a number of disks. By exploring the economics of disk storage or by building a distributed file system, the solution has not been that simple as it would be in case of a centralized file system. The distributed file system, although flexible, gives rise to problems like possible inconsistencies due to file sharing, chaotic situations due to file naming and access mechanism, deadlock situations due to possible interleaved transaction processing, unreliability due to failures of all kinds, etc. These problems have been thoroughly examined in past, and several solutions have been suggested.

2.3 General Issues

In this section, a brief discussion about the most general and crucial issues arising in a distributed file system is given.

2.3.1 Object Naming

In a distributed environment where a high degree of cooperation among different users is required, the issue of object naming is fairly important. Files stored in a distributed file system are very often shared by many users. Therefore, each file (object) should be uniquely identified by an internal name. Unique identifiers (UIDs) are generally being used as internal names. Sometimes, they are referred as file identifiers (FIDs).

This identifier can be a system generated number or a structured name. It

has the benefit of being passed from one user to another easily. Above all, it can ensure location independence and avoid ambiguities. Problems like generating and guaranteeing uniqueness of UIDs, locating an object given its UID, naming different versions of objects, etc., exist. Possible solutions have been suggested in [Leac 82] for using UIDs effectively.

2.3.2 Directory Management

A file system should have some way of keeping track of all information through various tables, used in locating where a file or a page is, in identifying the file owner or the file properties, and in mapping the file's symbolic name to a system given name. The major table is considered to be the directory. The mapping between the symbolic file name and the corresponding file is accomplished through a file directory. The mapping function becomes complex when dealing with thousands of files grouped in a hierarchical fashion like path name in UNIX type of files. Therefore, the role of directory server is vital in a distributed file system.

Various options for providing directory services have been investigated. Directory services could be centralized or distributed. In the centralized case, one independent unit like any other server can simply provide directory services to the users. However, a single directory reduces accessibility and makes recovery difficult. With a distributed directory system, it is possible to enhance accessibility through multiple copies of the directory. By allowing redundancy, we run into consistency problem. Lunn and Bennett [Lunn 81] have given a scheme to overcome such inconsistencies.

The primary function of a directory server is to maintain a mapping from alphanumeric name (possibly, structured) to unique identifiers and to provide lookup

services to the user community.

For the case where just one file server provides storage facilities for the entire network, it is useful to have the directory facilities embedded with the file server. With a set of cooperating file servers which form a file system, a global directory server like in [Stur 80] is a simple solution.

2.3.3 Access Mechanism

Access to a resource must be suitable to its requirement and at the same time must be restricted to a set of allowable users. This is usually done by access control list or by capabilities [Andr 79]. Capability based access control is very flexible and has received a great deal of attention. It can be passed from one user to another freely. Access to a resource depends on the ability of a user to present a valid capability for it. However, a capability may suffer from accidental or purposeful modification in course of transmission through the communication medium and can create chaotic situations. Proper care must be taken for full protection of the capabilities.

A distributed file system should be able to provide friendly interfaces to the user community. The file system and its users communicate by sending messages to each other. In case of an error like unauthorized access or unrecognized request, the user should be promptly notified with adequate explanation.

Fundamental file system access mechanisms must be straight forward and follow "result for every request" pattern. In other words, when a user requests for some service, his request should always be prompted by a reply, with no regard to success or failure.

To follow the guidelines of virtual memory construct, it is necessary to have block (page) oriented files. The file system should make no assumption about the structure of any application file except that it is a sequence of blocks (pages). Files can be accessed sequentially or randomly on block (page) basis, word or byte basis if required.

2.3.4 File Sharing and Protection

One of the primary objectives of any distributed file system is to allow file sharing effectively. If the files are not shared in a controlled manner, major problems will arise. Therefore, steps towards synchronized operation and data integrity must be taken to maintain internal consistency.

File sharing is permitted through specification of proper access modes. The file system should provide facilities for locking files, either entirely or on page basis. Read and write locks are used to control access to files. Proper care must be taken to prevent antagonistic users from holding locks indefinitely. Provision for allowing users to request for locks should be made available. Cyclic lock dependency (dead-lock), if at all exists, should be resolved. To cope with concurrency problem on shared data, various concurrency control algorithms exist. The three major ones are: locking, timestamp ordering [Bern 81] and optimistic approach [Kung 81].

Moreover, the file system must have mechanisms for protecting files from unauthorized accessess. These mechanisms which constitute the protection subsystem must be flexible enough to allow controlled migration of access rights to files. Protection on files can be implemented by capabilities at the cost of the protecting the capabilities.

2.3.5 Reliability

Reliability is a fundamental issue in any distributed system. It can be viewed as the degree of tolerance against any hardware or software failure. Faults that cause a system failure can be classified as file server failure, disk controller failure or media failure [Stan 84].

In a distributed file system, involving a number of file servers, a user can spread copies of files throughout in an effort to ensure availability. If it is the responsibility of the file system to maintain consistency of these copies during subsequent updates, then it is a burden even when the files are not shared. A site storing a copy of the file can fail just before or during an update and recover independently which may be hazardous. Ways to deal with such problems have been discussed in [Svob 84]. Therefore, consistency should be given a high priority even though it applies somewhat lower availability and performance.

File update procedures must be atomic to preserve a consistent view of the data. An atomic action is indivisible with respect to failures and concurrent accesses. To ensure atomicity, the problem of concurrency control, commitment and object recovery must be solved [Svob 84].

Reliability is often judged by the stability of the system and the robustness of the recovery procedures in face of a system crash. To guarantee that objects like directory map or file map will survive a crash, they must also be stored on nonvolatile secondary storage and be reconstructable from the information in the secondary storage.

2.3.6 Internal Organization

From performance point of view, internal organization is fairly important although it is not a major issue. It is upto the designer to decide on a policy for determining where and how information should be stored and who gets access to the information. Factors influencing this policy are efficient utilization of space, efficient access, flexibility to users, and protection of access rights to the information required. The overall success lies on these factors.

As discussed earlier, the file system should make no assumption about the structure of a file except that it is a sequence of pages (blocks). Organizing the occupied or available space in an orderly manner has the advantage of faster access and easy manipulation. Therefore, a tree-structured organization such as B-tree is widely used.

2.4 Conclusion

This chapter reviewed the evolution of the file management starting from backing store server to more sophisticated file servers and distributed file servers in a communication network. These file servers are not database systems by themselves, but they provide interfaces to database systems. The general issues governing the design of a distributed file system have been highlighted in this chapter.

Chapter 3

Existing Approaches

3.1 Introduction

This chapter gives a summary of some of the existing file servers, their design goals and approaches in addressing the general issues described in the previous chapter. The discussion will primarily focus on the Cambridge File Server, the Woodstock File Server, the Felix File Server and the Xerox Distributed File Server.

In Section 3.2, the salient facts about these existing systems are given in terms of the hardware and software environment they are based on; Section 3.3 describes the overall design objectives. In Section 3.4, a description on how each file server addresses the general issues described in Chapter 2 (i.e. object naming, access control, directory management, file sharing, reliability and protection) is given. Finally, in Section 3.5, implementation details are given, followed by a concise table highlighting the main characteristics of these file servers.

3.2 Backgrounds of Existing File Systems

3.2.1 Woodstock File Server

The Woodstock File Server (WFS) [Swin 79] was developed at the Xerox Palo Research Center, California, and was primarily intended for the Xerox community.

Woodstock's hardware environment is a network of minicomputers, each providing specialized functions such as terminal control, editing, filing, message services, etc. in support of the overall application. WFS is implemented on a dedicated Xerox Alto research minicomputer, augmented by one or more disk drives, each with a transfer rate of around 7 megabits per second and a capacity of 80 to 300 megabytes. The software for WFS is written in BCPL [Rich 69] language, supported by a custom-tailored operating system and communications package.

3.2.2 Felix File Server

The Felix File Server (FFS) [Frid 81] was developed at Bell-Northern Research for an experimental distributed microcomputer system.

FFS's technology encompasses a high-speed local communication network, a user workstation, the file server, a network operating system and a concurrent programming language. Felix software is implemented in extended Pascal. The language is a superset of UCSD Pascal with extensions for parallel programming. The concurrency constructs include tasks and an intertask communication mechanism similar to the rendezvous in ADA.

3.2.3 Cambridge File Server

The Cambridge File Server (CFS) [Dion 80] was built at the Cambridge University computer laboratory and was originally conceived as a replacement for the backing store management of the CAP computer [Wilk 79a].

The CFS runs on a Computer Automation LSI4-30 minicomputer with 64K 16-bits word of memory and 150 megabytes of disk storage with access time of 35 msec and transfer rate of about 1 Mbyte per second. The Cambridge Ring [Wilk 79b] serves as the communication medium between CFS and its users. The software for CFS is written using BCPL language.

3.2.4 Xerox Distributed File Server

The Xerox Distributed File Server (XDFS) [Stur 80] was built at Xerox Palo Alto Research Center, California, after the development of Woodstock File Server.

Ethernet [Metc 76] with raw bandwidth of 2.94 Mbits per second serves as the communication media between XDFS and its users. XDFS runs on an Alto minicomputer of 164K 16-bits words, supporting 300 Mbyte disk units with just about the same access time and transfer rate as the Cambridge File Server. The software for XDFS is written using Mesa language [Metc 79].

3.3 Primary Design Objectives

The fundamental goals on which the design of all the existing file servers are based, fall into the following three categories:

- Support for virtual memory environment
- Extensive sharing of data
- Secure storage for data

Minor variations in the above categories exist with all the file servers to cope with intended applications. However, all of them follow capability-based access control in some form or other, allow a high degree of file sharing and provide adequate level of data integrity and security.

3.4 Approaches Towards General Issues

This section describes various approaches taken by the Felix File Server, the Woodstock File Server, the Cambridge File Server, and the Xerox Distributed File Server in addressing the general issues such as object naming, directory management, access control, file sharing, protection and reliability. Internal organization of each of these file servers will also be discussed briefly.

3.4.1 Object Naming

Each file server uses a unique identifier called internal name for each file. The WFS has a unique 32 bits identifier (FID) to name a file. It is the responsibility of the user to remember the FID's. The mapping from a FID to the physical disc address is done via a hash table. The CFS uses a 64 bits long unique identifier

(UID) to name a file out of which 32 bits are random numbers, and the rest contains the mapping information. CFS preserves these UID's before returning them to the users. The FFS uses a system generated file identifier (FID) to uniquely identify a file. Once a file is deleted, the corresponding FID is guaranteed not to be used for a given period of time. The XDFS names its files by a unique identifier (UID), which is essentially a long integer. This UID differs from that used in CFS.

3.4.2 Directory Management

In addition to maintaining directories for the stored files, most of the file servers provide name lookup services to the user community in some form or the other.

The WFS maintains a simple form of directory for all the files from the time they are created. Each directory entry includes a FID, the location of the file (disk address) and the file properties. There is no additional naming or directory structure within the system. The FFS does not provide a general purpose directory service. It simply allows its users to make an entry in the directory. Each entry in the directory is a pair (filename,FID). An optional password protection facility is provided in which case the directory entry looks like (filename,password,FID). Similar things can be said about the CFS. It provides basically the same facilities as the other two. In case of the XDFS, a special server called the directory server provides the overall directory facilities. By isolating the directory services means that different implementations of directories may use different file naming schemes and still share the file server facilities. The directory server in XDFS acts as a special user and participates in all transactions.

3.4.3 Access Control

When the file servers allow for file sharing extensively, it is fairly important to have a suitable access control mechanism to regulate access and ensure protection.

The CFS provides access control based on capabilities, basically the UIDs. Validity of these capabilities are checked upon every request without any concern about the identity of the users. The CFS associates access information in every UID. In XDFS, users access files using UIDs in a manner similar to the CFS, but most access is preceded by looking up a file's name in a directory to obtain its UID. Directories contain identity-based access control. The directory server participates with the file servers in every transaction what is called default directory service. The FFS also uses capability-based access control on its files. The FID serves as a universal access capability to the file. There is no concept of file ownership. The capability (FID) can be passed from one user to another without modifications, thus providing full access privileges. The access privileges can be restricted through system defined primitives to any specific access mode. In WFS, a FID serves as a capability to a file. Upon creation of a file, a FID is handed over to the prospective user, and the file properties such as access mechanism corresponding to that FID are recorded in the directory entry.

The CFS permits word-level random access to files. The WFS and FFS allow page-level addressing scheme although the page size differs in each case. The XDFS allows random access to a sequence of bytes and pages.

3.4.4 File Sharing

File sharing is one of the primary objectives in any file server. All the existing

file servers allow shared access to the stored data in a coherent fashion and deal with inconsistencies arising due to concurrent access.

The FFS gives the option for manipulating a number of files as a single entity which is called a file set. A user may open several files, update them and commit all the changes in one indivisible operation. File commit operation whether involving a single file or a set of files as in a file set is atomic to achieve data integrity and consistency. No changes to files are visible externally until they are committed. File commit operation gives rise to the notion of file version. The version number is incremented after each commit operation. File sharing in FFS is possible through the specification of various access modes at the time of file creation. These access modes are: read copy (RC), write copy (WC), read exclusive (RX), write exclusive (WX), read original (RO) and write original (WO). Each access mode has a different interpretation. In read copy mode, the requestor is given a copy of the most recent version of the file for read-only access. Concurrent reading or writing by other users is allowed. In write copy mode, concurrent reading or writing is permitted but the requestor is not allowed to commit his copy of the file. In read or write exclusive mode, no concurrent users are permitted. Also no concurrent writer (unless it is a WC writer) is permitted in a read original mode while concurrent RC reads and WC writes are allowed in write original mode. The overall sharing and protection are provided through these access mechanisms. FFS also provides the equivalent of conventional two-phase locking in RO and WO modes. In case of deadlocked transactions, FFS breaks the deadlock by aborting judiciously chosen transactions and notifies the prospective users.

The WFS's approach to file sharing is primitive. A user may lock a file, preventing access by anyone without a proper key. The lock operation returns a key

that must be supplied with all subsequent operations on the file until it is unlocked. WFS associates timeout mechanism with the lock operation. It will intervene and break a file's lock if no operation has been performed for a certain period of time. These simple locking mechanisms at file level provide primitives that are adequate for file sharing and protection against unauthorized access.

The CFS supports file sharing through interlock mechanisms. It provides atomic transactions to its users. By supplying the UID of an object, the user receives in return a temporary unique identifier (TUID) which defines a capability for an interlocked object. The user can perform appropriate operation upon possession of a TUID. A TUID can be passed from one user to another. While there is a TUID outstanding, further requests to establish conflicting interlocks on the same object are likely to be rejected. CFS sets timeout values to detect possible crashes in user machine. It also locks the entire file in case of a multiple-reader-single-writer transaction for the duration of the transaction and allows one file to be updated atomically in a single transaction.

The XDFS provides a number of mechanisms to assist its users to update shared files while keeping them consistent. The most important of these mechanisms, the atomic property for transaction, allows the user to encapsulate a sequence of reads and writes as an indivisible, atomic action. XDFS uses time limit locks as part of its approach to file sharing. Unlike CFS locks, XDFS locks are set implicitly as side effects of read and write requests. These locks are set at byte level and have associated timeout periods. The lock may be broken after its timeout period has elapsed if another transaction is waiting to acquire that lock. To enable users to maintain local caches of file data in the presence of concurrent access, XDFS informs the user when any of its read locks is broken. In such a situation, the user

releases the locks and attempts to reread the data.

3.4.5 Reliability and Protection

In the presence of extensive file sharing and possibility of various failures, the issue of reliability draws more attention than any other issue in any file system. A serious service failure may destroy the contents of a processor's volatile memory, corrupt or destroy a disk block. More information may be lost than can be reconstructed during automatic recovery. All the file servers, as described in the previous subsection, guarantee atomic updates on files and simultaneously provide some mechanisms to deal with various failures.

In case of the FFS, a file is securely maintained by the system until a "delete file" request is received. The user must explicitly delete files when they are no longer needed. As discussed earlier, FFS uses multiple file commit mechanism to secure a file, thus guaranteeing mutual consistency. The user always receives an acknowledgement to its commit request. In case of a system crash, unopened files have no effect but the opened files are closed without commitment. All instances are wiped out, and the file is restored to its last committed version before crash. Optional password protection mechanism on files are also provided. A file or a file set is always committed in one indivisible operation.

In case of the WFS, the kind of protection required has to be specified by the user. A set of file properties are assigned to a file, and they are used by the low-level crash recovery routines to reconstruct the damaged file structure. WFS has implemented a simple, low-level abstraction with relatively few operations and with high reliability. Each operation runs to its completion before it is acknowledged. A write operation is not complete until the data is written to the disk successfully.

Update is atomic, and if there is any discrepancy then the request is rejected, and the user is notified. Between operations, WFS retains no state information that can not be regenerated from the contents of the disk.

In the case of CFS, the user need not take any precaution against a file directory update being interrupted by a system crash. CFS distinguishes between data blocks, which hold the contents of the object, the map blocks which define their structure. In case of a crash, all map blocks are restored to their last consistent state so that the storage is consistent in terms of block allocation, but the data in it are restored only as directed by the user. The system takes the responsibility of deciding when to delete an object. A CFS object is either a file or an index, each identified by a unique identifier. An index is a list of unique identifiers. After creating an object, its UID is saved in a selected index. This ensures the reachability of the object from some index and prevents its premature deletion. The CFS also keeps a block allocation table. The allocation and deallocation procedure is atomic. Files are treated either as normal or special kind. The user must declare at the time of creating a file according to whether or not the file will be used to hold critical data. A special file will always revert to its initial state if a transaction fails, but a normal file may not. For normal files, the open and close requests provide only concurrency control without atomic update guarantee.

In case of the XDfs, lost files can exist, because the directory server is not used by the users directly. The directory server is like an ordinary user to the XDfs. The possibility of a file being lost is small. If a transaction fails, then any files created during that transaction are deleted automatically. The XDfs provides the structural redundancy using an abstraction called stable storage. Pages of essential information such as page maps are recorded twice on disk. Like CFS, it restricts its

recovery algorithms to failures that corrupt single disk page during a disk transfer. It does not rely on the contents of the processor's volatile memory after a failure, and the construction of lost information depends solely on the server's disk. With the recovery algorithm, writes to pages of stable storage can be considered as atomic actions. When XDFS involves in multiserver transactions, it employs two-phase commit protocol.

3.5 Comments on Implementation

In the FFS, a file is physically organized as an unbalanced tree of disk blocks with leaves carrying data and internal nodes providing access mechanisms. Each node in the tree keeps a one-bit indicator in addition to its address to indicate if a block is temporary or permanent. A file commit resets the indicator for all blocks in the tree to permanent. The physical implementation is hidden from the users. The concurrency constructs include tasks and intertask communication, each task being responsible for a specific activity. An "instance" task facilitates access to a file while a "file" task takes care of various aspects of file sharing such as queuing of new open requests, creation and control of instances, etc. The "directory" task maps a FID into the current disk address of the file root. The "system" task is the final authority in performing commit operation and managing free storage.

Like in FFS, files in the WFS are also organized in tree forms, the leaves being data pages and the internal nodes being root or leaf page maps. A page map is either one or two levels deep depending on the length of a file. The FID directory is a hash table, implemented as a contiguous, fix-sized file at a known disk address. A single process, unlike in FFS, interprets the user operations in WFS. This process sequentially extracts request packets from the input queue, checks for validity and

dispatches for appropriate operation. When the operation is completed, the process returns a respond packet to the requestor.

Recall that the objects stored in the CFS are of two types— files and indices. The storage controlled by CFS are divided into data blocks and map blocks. The data blocks hold contents of the objects while the map blocks define their structures. The storage maintained by the CFS appears as a directed graph to its users. The information needed to map a (file id,page number) pair into the disk address of the page is recorded in a tree of disk pages per file. Files are terminal nodes in the graph, and indices hold links to other nodes. To economize on space, the CFS creates each object as a single empty disk page, and adds indexing levels to the tree only as the object expands. In this way, small objects occupy a single disk block, while larger objects may require additional one or two levels of indexing. There is one root index. To allow rapid access to the tree describing a file, each UID contains the disk address of the root page of the tree. A file or index is kept as long as it can be reached by a series of retrieve operations starting from the root index. Because of the directed graph maintained by the CFS, cycles may develop. So, periodic garbage collection is needed to detect objects which are no longer reachable from the root index. The CFS uses shadow pages to record accumulating states of transactions. These pages are inserted into the appropriate positions in the tree after the transaction is completed, otherwise discarded. Cylinder map entries record these shadow pages.

The XDFS has more efficient way of structuring its spaces to ensure minimal access. It uses a B-tree to translate a (file id,page number) pair into a disk address. The B-tree is implemented using stable pages. There is a single B-tree per disk pack, and it records the allocations of pages to files for all files on the

pack. Near the center of each pack a bit map is stored which is used to record the allocation state of each page on the disk. The XDFS, like CFS, uses shadow pages to record accumulating states of transactions. To commit a transaction, the B-tree is updated in a bottom-up fashion with shadow and stable pages of its own.

3.6 Comments on Performance

Performance measures are fairly important to both the users and the designers. In addition to the reliability, a user is always concerned about the responsiveness of the system. The delay incurred in accessing a file is influenced by several factors, the major ones being the communication overhead, the number of users in the system and the locality of reference to a file. Not all the existing file servers provide performance models and analysis explicitly. It is also difficult to compare performance characteristics of two file servers with different structural configurations.

WFS has never been used in an environment subject to a high volume of concurrent accesses by a large number of users. However, it gives performance figures obtained from the analysis under a heavy load with upto three users. Each page in WFS is 256 16-bit words long. With a single user, it takes about 48 msec on an average to read a page and 73 msec on average to write a page. With two users, the figures are 76 msec and 109 msec respectively. With three users, the figures are 100 msec and 150 msec respectively. Note that these figures include communication overhead. It clearly shows that the response time increases with the increase in the number of users in the system.

Performance analysis of any sort for the FFS is not available. Thus, there is

no way of determining efficiency of this file server. The page size used in FFS is 1 KB.

Performance analysis of CFS and XDFS has been looked at in great detail in [Mitc 82]. The page size for CFS is 2048 bytes while it is 1024 bytes in case of XDFS. Therefore, it is difficult to compare both systems precisely. Several experiments have been conducted on an equal amount of data (256 bytes) for read and write requests in both the file servers. Basic communication overhead for CFS is considerably less (12 msec) than that of XDFS (38 msec). It takes about 41 msec in case of the CFS to read 256 bytes on an average (taken from 100 random reads), while it takes 76 msec in case of the XDFS to read the same amount of data. The reading time includes seek and rotational latency of the disks plus the time to find the page under a given file structure. On the average, CFS takes 73 msec to write 256 bytes while XDFS takes 142 msec for the same amount of data. CFS uses two kinds of files — normal and special. If a normal page is written, no allocation cost is incurred because a shadow page is never created. Special files require more overhead due to the cost of allocation.

3.7 Conclusion

This chapter briefly summarized the backgrounds of some of the well-known file servers. The designs of all of them, although application oriented, follow some general principles to a great extent. The XDFS, as the name suggests, supports a number of cooperating servers which together give the illusion of a single logical system. The CFS, FFS and WFS are not based on that principle. All these file servers have some common design objectives such as capability-based access control, support of virtual memory, high degree of data integrity, adequate level of security and reliability. These file servers are not databases by themselves, but rather provide high level interface to distributed databases. They are used as permanent data storage for several users in communication networks. The highlights of the important features of these file servers is provided in Table 1.

Table 1: Highlights of Important Factors of Four File Servers

Object Naming:

CFS : 64 bits identifier (UID)
WFS : 32 bits identifier (FID)
FFS : Unique identifier (FID)
XDFS: Large unique integer (UID)

Access Mechanism:

CFS : Capability based access control
WFS : Capability based access control
FFS : Capability based access control
XDFS: Identity based. Names and passwords are used.
to authenticate users

Data Unit:

CFS : A sequential subset of a file (Word-level)
WFS : Page
FFS : Page
XDFS: Sequence of bytes and pages

Directory Management:

CFS : Basic facilities
WFS : Simple directories
FFS : No general purpose directory service
XDFS: Separate Directory Server and default service

File Sharing:

- CFS : Through interlock mechanisms. User receives a temporary id (TUID) on interlocked objects. CFS locks the entire file for concurrency control
- WFS : Through basic locking primitives. Timeouts are associated with locks
- FFS : Through specification of access modes. FFS uses conventional two-phase locking. Locks the entire file for concurrency control
- XDFS: Through timeout locks and twophase locking. Also locks the entire file for concurrency control

Deadlock Control:

- CFS : No specific mechanism. Mostly it is the user's responsibility
- WFS : Through time locks
- FFS : Detects, resolves and also prevents deadlocks
- XDFS: Through time-limited locks

Reliability and Protection:

- CFS : Saves UIDs to prevent premature deletion. User must specify protection by specifying the type for the file. Supports atomic update on single files
- WFS : Supports atomic updates. Kind of protection required must be specified by the user
- FFS : Supports multiple-file commit mechanism. Also provides optional password protection mechanism
- XDFS: Provides two-phase commit mechanism for multi-server transactions. Essential information are recorded twice on disk. Supports atomic updates on multiple files

Chapter 4

Proposed DFMS

4.1 Introduction

In this chapter, a new file management system will be proposed, its main features described, its behaviours outlined, and its performance analyzed.

The proposed system, which we shall call DFMS (for "distributed file management system") preserves many of the characteristics of the others. Unlike the others, it allows total "dispersion" of the data, and it exhibits fully (and truly) decentralized behaviour. In particular, it manages storage by scattering files as required and performing compaction periodically. It is formed as a cooperating set of file servers and a directory server, which together give the illusion of a single and logical file management system.

In Section 4.2, the motivations for such a system are outlined from the point of view of the current technology; in Section 4.3, the design objectives of DFMS are described, especially its functionalities and the file processing environment it can be suitable for; in Section 4.4, the functional view of DFMS is given in terms of its components; in Section 4.5, more detailed information about the file servers in the system is given, while in Section 4.6, the focus is on the directory server structure and its functionalities. Finally, in Section 4.7, it is described how some

of the general issues of distributed file systems mentioned in Chapter 2 are met in the proposed DFMS.

4.2 Motivation

The widespread use of a distributed system is due to the development of cost effective and high-bandwidth communication networks, the development of resource sharing software, and the increased demand for communication, economical sharing of resources and productivity. A distributed system potentially provides significant advantages, including good performance, good reliability, good resource sharing and expandability.

Advances in VLSI technology have driven the cost of hardware down. Similarly, advances in data communications technology have produced high-bandwidth, local area communication networks. These networks permit interactive access to their facilities with reasonable response times. The availability of high-bandwidth communication subnets like the Ethernet is an underlying motivation for the design of a distributed computer system. The communication subnets are characterized by high data rates (0.1 – 100 megabits per second) and low error rates (10^{-8} – 10^{-11} bits per second). Hence, the communication time and cost are fairly low.

Integration of current technologies has made possible the sharing of expensive resources, coherency and efficient communication among the users. Most benefits of communication systems come from their distributed nature. It is reasonable to separate the actual processing of data from the system (file system) required to store that data reliably. Such a file system could support a number of different host operating systems, each having an added software to meet the network operating

system standards. In most distributed systems the file system is considered as a server which processes requests from users as well as from the rest of the operating system. CFS, WFS and FFS represent such types of file systems.

With current technology, a single server with one disk drive might hold 80 Mbytes of data. The nature of resource sharing can be enhanced further by exploring the economics of disk storage. Using rather small computers as file servers permits a wide range of file storage capacity because it is possible to expand from a single minicomputer with one disk drive to many servers, each with a number of disks. XDFS is based on this principle. The set of cooperating file servers under XDFS not only provides a wide range of file storage but also gives the illusion of a single logical file system. The user simply has to communicate with simply one of the file servers for any transaction. The bottleneck problem could possibly exist in case of CFS, WFS and FFS. As the number of users grow considerably large, the overall performance, in terms of response time, will be affected significantly.

4.3 Design Objectives

The proposed Distributed File Management System is intended to provide facilities from single file access to complex arrangements which provide file sharing, security and many more. It is not a Database Management System (DBMS) nor a Distributed Database Management System (DDBMS) by itself, but it can provide services to be used in higher functionalities such as query processing. It will have the ability to provide high-level filing facilities to different kinds of operating systems.

4.3.1 Access Mechanism

Access to a resource must be restricted to a set of allowable users. Capability based access control is very flexible and is being widely used. The DFMS is intended to provide capability based access control mechanisms, straight forward file access mechanism and friendly interfaces to the user community.

4.3.2 Virtual Memory

Virtual memory concept in memory management is getting a lot of attention in many computer systems of the present age. To facilitate in virtual memory constructs, the proposed DFMS is intended to support page-level addressing.

4.3.3 Data Sharing

While the sharing of files among different clients is widely recognized as one of the primary services vital in a network, it has not always been convenient or easy to do. The DFMS allows users to share information in a controlled manner. By allowing extensive sharing of information, the problem of data inconsistencies may

develop. Consistency is given high priority even though it implies somewhat lower availability and performance. The DFMS will guarantee atomic updates on stored data.

4.3.4 Directory Management

All the existing file servers provide some form of directory services to their clients. This facility, for example, is available as part of the services provided by a file server like FFS, CFS and WFS; in other words, the directory server in each of these file servers is combined with the file server. XDFS has a separate directory server, leaving less responsibility on its file servers. In principle, a file server should be able to provide well-defined and general purpose directory services. The DFMS's directory server is centralized, containing non-replicated directory files; it supports hierarchical file structures and engages itself in certain decision making process. In a sense, it monitors the activities of the file servers under DFMS.

4.3.5 Protection Mechanism

Protection of files from unauthorized access is done through the specification of proper access modes in DFMS. It uses shadow pages for update purposes. Important information such as directory files are recorded twice in stable storage. The recovery procedure is robust, and it uses standard techniques.

The DFMS is well-suited for various kinds of file processing environments which involve sequential or random access files. The key features distinguishing this DFMS from other file servers in this aspect are the following:

- Ability of each file server to share space with others

- Ability of each file server to perform compaction
- Ability of incremental upward scaling of storage capacity upon demand by introducing additional file server(s)

The type of the file system considered here may consist of several small computers with no facility for memory expansion to satisfy personal need. The DFMS does not impose a size limit on any file although there is memory restriction at each file server under it. To satisfy a need, a file server is permitted to "borrow" unused space from other file servers and store part of its data at different locations. The entire process remains transparent to the users. The consequence of this "borrowing" is that a file may remain scattered among two or more file servers, each holding a portion of the file. The user is in illusion that its file is located at a specific file server. To access a page, if it does not happen to be at the host server at the time of request, will involve extra cost and time. Therefore, compaction is necessary to minimize access time and communication cost.

The schema of the proposed DFMS is outlined in the Figure 1. Its principal characteristics can be summarized as follows:

- Support for virtual memory
- Capability based access control
- Ability to provide well-defined directory service
- Ability to support extensive file sharing
- Ability to perform atomic updates on files
- Ability to provide adequate level of security
- Ability to perform compaction of used space

Figure 1 : Schema of a Distributed System

4.4 Functional View of the DFMS

Figure 1 described the style of the proposed Distributed File Management System. The DFMS consists of two types of components:

- File servers
- Directory Server

The cooperating set of file servers and the directory server collectively form a single logical file system known as DFMS. They are connected to a high-bandwidth communication bus along with other servers and potential users. The communication medium is considered to be highly reliable with a negligible error rate and so is the underlying protocol. Message passing is the only mode of communication between any two entities (e.g., user or server) in this distributed system.

Each file server can be thought of as a small computer with a fixed disk storage, and it is identical to other file servers in DFMS in terms of page size. In other words, each file server provides storage (thought of as a set of pages) in which the page size is same for all the servers although the capacity of each of them may differ from that of the other. A detailed description of the file servers will be given in the next section.

The directory server simply provides general directory services to the users in the system and plays a vital role in coordinating and monitoring the activities of the file servers. The function and organization of the directory server will be given in Section 4.6 of this chapter.

Each file server and the directory server has a unique network address which is the file server identifier (FSID) in the case of a file server and directory server

identifier (DSID) in the case of the directory server. The DSID is known to all users in the distributed system. The file servers know each other's FSID. The FSID is like a token that is assigned by the directory server to a client to carry out a transaction with the designated file server. The reason for such an arrangement will be more clear later.

The DFMS protocol is based on direct transmission of network packets which does not require any connection to be established. It has the following structure:

1. The user sends out a request packet
2. The DFMS processes the request and sends out one or more response packet
3. Upon receipt of the response, the user sends an acknowledgement packet.

To perform an operation on a file, the user constructs a request packet containing the appropriate command(s) with associated parameters, if any, and sends it to the directory server or a file server as required.

Figure 2 : Structure of a File Server

4.5 File Server Description

The file servers constitute the main component of the DFMS. They are cooperating in the sense that in the event of a shortage of space in any of the servers, other servers can lend their unused space temporarily. This section describes the intended structure, user interface and internal organization of a typical file server.

4.5.1 Structural View

A typical file server can simply be a small computer (possibly of 16 - bit word size) with a disk unit and sufficient processor memory to hold file server software. The limit on disk space is not of primary concern at this stage. However, it has been assumed that each file server can provide a fixed amount of storage. Any particular size of the memory and disk unit has to be decided at the time of implementation depending on the need. Also for the simplicity of discussion, it will be assumed that the available space in each of the file server is structured into a finite number of fixed size pages, possibly 256 bytes each. A file server can be viewed as consisting of the following components:

1. Medium controller module
2. File server and system software
3. Disk controller module

The function of medium control module is to interface with the communication bus. It receives packets addressed to its file server from the bus and transmits outgoing packets into the bus, possibly with multiple attempts until the packet reaches its destination successfully. Several users and servers access the bus in communicating with each other, so there are possibilities of packet collisions. Damaged

packets received are discarded by the medium controller module. Fundamental message-passing primitives such *send*, *receive*, and *forward* are known to this module in interfacing the communication bus. The medium controller module is like a front-end processor connected to the rest of the file server.

The system software is the sole authority in monitoring the entire file server activities. Its overall function is to verify and process user requests, to manipulate complex internal data structures, to ensure consistencies following concurrent activities, to maintain disk storage, to make lending and borrowing decisions, and to perform compaction periodically as necessary. The file server software can be written in a language like Mesa or DC (Distributed C).

The function of the disk controller unit is to read information from and write information into the disk upon request from the file server software. The three modules described above constitute a typical file server which is shown in Figure 2.

4.5.2 Primary and Secondary Sites

It has been discussed earlier that when a file server needs extra space, it can always look for unused space from other file servers and can store data there until it can accommodate within its own premises. Each file in the system is considered as a finite linked list of pages stored at one or more servers, creating an illusion as if it has been located at just one server. The file distribution follows from the assumption that each file server has a limited storage capacity. The server where a file is initially created is called the *Primary* site while the other file servers are called *Secondary* sites with respect to that particular file. In other words, a file server can be *primary* and *secondary* site at the same time for different files.

A file may be scattered over several sites with its beginning pages at one *primary* site. The strategy for selecting a *secondary* site will be discussed in the next chapter. If the file server is a *primary* site for a file, it keeps track of all the sites which have contributed pages (i.e., are *secondary* for that file). From this information and its local knowledge, the *primary* site can reconstruct the entire file-chain. A file-chain for a file has information such as which page is located at which site. It is essentially a link list with respect to the pages in a file. Since many sites may contribute pages to construct a file, it is important for the *primary* site for that file to retain the global view. A *secondary* site just needs to know the last site (ancestor site) which has contributed some pages to the file. This *secondary* site will be the successor of its ancestor site. If the file server is a *secondary* site, it remembers the immediate ancestor and the immediate successor sites for every file it serves as a *secondary* site.

4.5.3 File Structure

A file follows the UNIX [Ritc 78] style and simply is a sequence of fixed size pages. This follows from the convention of structuring each file server's storage. There is no absolute limit on the size of any file imposed by the DFMS. Each file is identified by a system (directory server) generated unique identifier called the FID. The FID serves as universal access capability to a file. From the time of its creation, a file can be accessed through this identifier. The access mechanism and permission have to be specified for a file at the time of its creation. Facilities for file sharing is permitted on proper mode as specified by the owner. Locking facilities on files are also available. There are two types of files: directory files, containing file map and directory information, and ordinary files carrying user data.

4.5.4 File Server Interface

To open a file the client first sends an open request, giving all properties of the file to the directory server, and in return it receives a FID and a file server identifier (FSID). Before starting a new transaction on a file, the user must locate the file server where the file is stored by sending a lookup request to the directory server. The access to a file depends on the ability of the user to provide a valid capability which is nothing but the FID. With a valid FID and a FSID, a user can perform basic operations on a file. DFMS provides random access to files on page basis. The operations on file fall into the following categories:

1. Opening and closing a file
2. Reading and writing pages of a file
3. Allocating and deallocating pages of a file
4. Locking and unlocking a file
5. Obtaining and modifying file properties

A summary of the DFMS operation is shown in Table 2. The most commonly used commands are *open*, *close*, *readpage*, *writepage*, *allocatepage*, *deallocatepage*, *commit*, *deletefile*, *lock*, *unlock*, and *lookup*. Commands of first and fifth categories are directed to the directory server. File properties can be modified only by the owner of the file whose identification number (possibly a password) is recorded at the directory server entry at the time of its creation. Owner identifier may represent a single user or a group of users.

Commitment is the major tool for achieving data integrity and consistency. A file must be committed after an arbitrary number of write operations. No change to a file is externally visible until it is committed. Commit operation is atomic and

Table 2 : Interface Table

Readpage (<i>FSID, FID, PageNum</i>)	————→	Data
Writepage (<i>FSID, FID, PageNum</i>)	————→	Status
Allocatepage (<i>FSID, FID, PageNum</i>)	————→	Status
Deallocatepage (<i>FSID, FID, PageNum</i>)	————→	Status
Commit (<i>FSID, FID</i>)	————→	Status
Open (<i>DSID, Filename, File Properties, Ownerid</i>)	————→	(FSID, FID)
Close (<i>DSID, FID</i>)	————→	Status
Delete (<i>DSID, FID</i>)	————→	Status
Lookup (<i>DSID, FID</i>)	————→	(FSID, FID)
GetFidstatus (<i>DSID, FID</i>)	————→	(FID, FSID, File Properties)
SetFidstatus (<i>DSID, FID, Ownerid, File Properties</i>)	————→	Status
Lock (<i>FSID, FID, Frompage, Topage</i>)	————→	Lock Status
Unlock (<i>FSID, FID, Frompage, Topage</i>)	————→	Lock Status

it gives rise to the notion of a file instance or version. Whenever a file is committed, its version number is incremented by one. The file server keeps few versions of a file.

The DFMS is intended to allow file sharing whenever possible. Sharing of files is possible through different access modes which are very much similar to the ones used in FFS. The access modes are specified at the time the file is opened. Read copy (rc) and write copy (wc) access modes allow concurrent reading and writing. The requester is given a copy of the most recent version of the file for read/write access but not allowed to commit. Read only (ro) and write only (wo) modes also exist. Read exclusive (rx) and write exclusive (wx) modes do not allow users to read or write concurrently. Possibility of locking a file entirely or locking specific pages of a file are also provided. Associated with each lock is a timeout mechanism to prevent users from holding a lock indefinitely.

Text complete; leaf 46 omitted in numbering.

4.5.5 Internal Organization

The internal organization of a file server is fairly important from overall performance point of view. How well it maintains the stored information (tables and data) internally determines how fast an unit of information can be obtained and how reliable the system is in face of any failure. It is, therefore, necessary to pay enough attention in organizing the structure of a file server.

Since files are allowed to be scattered when some of the file servers run out of space and they remain like that until compaction is done, each file server (whether *primary* or *secondary*) should keep the following information:

1. File map for all locally stored files
2. Lending and borrowing table
3. Local file directory

Figure 3 shows the type of file map required by all the file servers. It has eight entries for each FID known to the file server. It is necessary for a site to know the previous site contributing page(s) to the file, the primary site for the file, and the next site, if any. Besides that the number of pages given by the present site, location of the first page, version number, and the file properties are also recorded in this file map. In addition to that, each primary site must keep a complete list of the pages and their locations for each file it is responsible for. This way, the primary sites only knows where the pages of an entire file are located in order to access them as required.

Lending and borrowing maps simply indicate the interaction of a file server with all other file servers. They show how many pages a site has borrowed from

FID	.	.	.	FID
Primary Site	.	.	.	Primary Site
Previous Site	.	.	.	Previous Site
Next Site	.	.	.	Next Site
Root	.	.	.	Root
#Pages	.	.	.	#Pages
Version #	.	.	.	Version #
File Properties	.	.	.	File Properties

Figure 3 : File Map

5

Page Identification (FID, Page#)
Date & Time
System Private
Page Data

Figure 4a : Page Format of a Typical Page

FID
File Location
File Properties

Figure 4b : FID Directory Entry in a File Server

others, and similarly how many pages have the other sites borrowed from it. These information are useful in making swapping decision during compaction.

Each file server must keep a local directory for all the files stored internally and for which the server acts as a primary site. The information contained in each file directory entry is shown in Figure 4.

As discussed earlier, the storage offered by a file server consists of a set of fixed size pages. A file may occupy several of such pages. It is inefficient to organize the pages of a file in shape of a link list. Typically, the pages in a file in the proposed file system should be arranged in shape of a B-tree to have fast access to each of them. Maintenance of the B-tree, although complex, can still be done with small overhead.

4.6 Directory Server Description

In the proposed distributed file management system, the directory server is a primary component like any file server. However, it does not normally provide storage facilities to the users or any file server. It is just a processor with some internal memory for storing directory files. This file system can support Unix types of files (hierarchical), therefore, the directory server must be able to handle such files.

The directory server has an id, DSID, known to every user in the system. The primary function of the directory server is to provide directory facilities and lookup services to its users. It also selects file servers to assign them to the users and may engage in compaction activity if centralized compaction algorithm is chosen. The open command for a file always comes directly to the directory server. Upon receiving an open request, it does the following:

- It looks if the file already exists. If so, then sends the existing FID and the FSID of the primary site to the user
- If it is a new file, then it records the file properties and the owner id (if any)
- It records the file name, generates a unique 32 bit number, and assigns as the FID of the file
- It looks for a suitable file server, by broadcasting a request to all file servers to send their available pages. When all the replies come in, it simply picks one site based on a given strategy. The FID and the file properties are sent to the selected file server for it to allocate a page and verify user requests from this point. The user is then sent the FSID and the FID to carry on with its file processing.

The directory server also provides lookup services, like where a file is located, which directory the file belongs, what are the access rights assigned to the file, etc. The directory server maintains appropriate tables for storing such information.

When it is required to close a file, the user broadcasts a close request with the FID for that file. Whoever is the primary site or the secondary site for that FID take necessary action when they receive this request. The directory server marks the file denoted by that FID as closed upon receiving the close request. During centralized compaction, if at all used, the directory server serves as a host until the completion of the algorithm. The section on compaction in Chapter 5 has more on it.

4.7 Discussion On General Issues

This section discusses about how and to what extent the general issues of distributed file systems are met in the proposed DFMS. Table 3 gives a summary of the main points.

The DFMS supports capability based access control. FID, a 32 bit unique system generated number, serves as an universal capability for a file. No identity of the user is needed in accessing a file.

The basic data unit that can be accessed is simply a page. The typical page size is 256 bytes. Files can be accessed randomly or sequentially on page basis. The file system, therefore, aids to virtual memory management. It also provides well-defined basic directory facilities like other file servers.

The DFMS supports extensive file sharing among the users. Access to a file is

Table 3 : Summary of Characteristics of the DFMS

Object Naming:

Through unique identifier (e.g. 32 bits FID)

Access Mechanism:

Capability based. Through the use of FIDs

Data Unit:

A Page (typically 256 bytes)

Directory Management:

Simple and directory lookup services

File Sharing:

Through specification of access modes at the time of file creation. User may also request for locks. Conventional Two-Phase locking protocol is used

Deadlock Control:

Detection through timeout locks

Reliability and Protection:

Update is atomic. The DFMS uses shadow pages for intermediate updates. Robust recovery procedure is available

restricted to users by the specification of access rights in the file properties when the file is created. Locking facilities are provided on request. The smallest locking unit is a page. However, an entire file can be locked. Timeouts on locks are setup to prevent users from holding locks for indefinite periods.

For interleaving transactions, concurrency control facility is available. Normally, conventional two-phase locking algorithm is used. Deadlock detection algorithm is also available through time limited locks. In face of a deadlock, the involved transactions are aborted, and the users are notified.

Essential information are recorded twice on disk. Updates on files are atomic. The DFMS uses shadow pages for storing intermediate record before updating. Robust recovery procedure is available. When there is a failure, the files remain in consistent state.

4.8 Conclusion

The proposed distributed file management system is an evolution of the existing file servers. It provides all the functions that others do, and it does something more than that. There is no imposition of memory limit at any file server or size limit for any file. When one server runs out of space temporarily, it finds from other file servers. Compaction is done periodically to swap spaces so as to keep as much data as possible locally in order to provide fast access. The main characteristics that distinguish this DFMS from other file systems are file distribution and compaction.

Chapter 5

Complexity and Performance Issues

5.1 Introduction

In Chapter 4, the proposed distributed file management system was introduced. In this chapter, we concentrate on performance analysis and complexity issues in the proposed system.

Section 5.2 discusses the primary performance objectives arising in the decentralized file management. Section 5.3 discusses different possible strategies, such as best-fit, random-fit and pass-next, suitable for site selection and allocation. Section 5.4 describes the simulation procedures and results obtained to validate the characteristics of different strategies.

Section 5.5 focuses on the compaction characteristic of the proposed file system. The two types of compaction methods considered are: centralized and decentralized compaction.

Section 5.6 looks at the performance characteristics of the system from an analytic point of view in terms of average delay or message complexity as applicable. In some cases realistic bounds on these delays or message complexities are given.

5.2 Objectives

The last chapter described motivations for designing the proposed distributed file management system. A main characteristic of DFMS is its ability to allow a file to be "scattered" if any server is unable to accommodate an entire file at a given time while still giving the user an illusion of single-site transactions. The entire performance becomes strictly dependent on how this "scattering" is performed (i.e., depending on which strategy is actually employed to allocate, relocate and manage a file). This section provides some facts and important objectives.

In a distributed system, communication time and costs are very much significant and vary with the type of activity performed. In order to provide the user with an acceptable range of response time, it is desirable to maintain communication time at very low level at all time. Since in our application the files might be scattered, the servers may need to communicate with each other, degrading the overall performance. For this reason, the number of secondary sites involved with any single file should be minimized. Furthermore, interactions between file servers for overall activities should also be minimized.

The directory server plays a significant role in the decentralized management of this file system. The strategy according to which the directory server selects an initial site determines the overall future interaction of that site with others with respect to that given file. In particular, if the file is assigned (for an open request) to a site with maximum number of available pages, the probability of overflow for that site, and, thus, the intersite communication is smaller than if a site with fewer pages is chosen. Another important time to look at performance is when a file overflows and a decision must be made as to which the other sites to "borrow"

space from. The information necessary to choose a secondary server is essentially the number of free pages at other servers at that point in time. There are two possible ways to acquire this information. One way is to broadcast a request to all other servers and wait for a response from them before making any decision. The second way is to request page status information from a centralized location, such as the directory server (in our case). The directory server, in this case, should have up-to-date information at all time. In our application, the first choice is very suitable. In certain situations, a server may have to choose a site which does not necessarily have the maximum available space in the system. Such a situation is the case when we minimize the number of secondary sites with respect to all the scattered files of a server. If a server has all its interactions due to its "own" files, with a set of secondary sites then choosing one of these secondary sites as one to "borrow" a page from, provided it can serve the request can be a step towards reducing the number of secondary sites.

In any resource sharing environment, the presence of deadlock is an inherent problem. Most people believe that it is easier, better and less expensive to prevent deadlock situation than to allow, detect and resolve the deadlock. In our application the deadlock problem is not a major threat; however, it may appear at two levels. First of all, there may be deadlock in transaction processing (described in the last chapter); secondly, a deadlock situation involving several file servers may be encountered when pages are tied up and the servers undergo compaction. In order to avoid possible deadlock situations, each site should choose appropriate sites for interaction.

In summary, the main goal for the decentralized file management is the minimization of communication cost which follows from the minimization of inter-site

communication. Proper choice in site selection and compaction are two key factors for the minimization of inter-site communications.

5.3 Strategies

This section describes some of the strategies under consideration for selecting a site for interaction. The advantages, disadvantages and the feasibilities of such strategies will be explored.

5.3.1 Random Strategy

This strategy can be adopted by the file servers (to select a site where space can be borrowed) as well as the directory server (to select a site where an open request should be assigned). Choosing a site for a page is done in a random fashion with no regard to any prior information, if at all available. When a site overflows, the overflow request is directed to another site, chosen at random. This approach is very simple and easy to implement. However, there are some disadvantages. First, a page request may have to hop a few times before finding an appropriate place. Secondly, the chosen site may be just about to overflow as soon it processes the page request.

5.3.2 Best-Fit Strategy

The essence of this strategy is to select a candidate which has the maximum number of free pages. This ensures that the probability of overflow at the selected site will be low. A page request does not have to hop a few times before finding an appropriate place. Also, this strategy can be adopted by the file servers as well as the directory server. It requires prior information about the status of

each file server. Gathering and maintaining this information requires additional communication time and cost.

5.3.3 Pass-Next Strategy

In case of an overflow, the server redirects the request to the server located either in left or in right. In such a case, each server should know the identity of its two immediate neighbours. This strategy is only appropriate for the file servers. Like random strategy, it does not require any prior information. This strategy may not look reasonable because of the left/right concept. This concept is applicable in a ring structure. However, it is possible to associate an implicit ring structure in a bus by numbering the file servers. This pass-next strategy is fairly simple and easy to implement, but suffers from similar disadvantages as in the case of random strategy.

Each strategy described above has some advantages and some disadvantages. It will be difficult to compare one strategy with another without making a tradeoff between communication costs and simplicity. When communication overhead is not a major criteria, the best-fit strategy is superior. Intuitively, the mean interoverflow time for each server, using best-fit strategy, will be considerably longer than that for the servers using other strategies under light or moderate load in the system. In a very heavy load, the gap is not very significant. Furthermore, the average waiting time for a page request to be processed should be lower in case of best-fit than any other strategy under light or moderate load.

5.4 Comparison Between Strategies

To derive a relationship among the various strategies discussed in the previous section, and to study the characteristics of the strategies under different loads in the system, a simulation study was performed. The model used for the simulation can be described as follows:

- Each file server is modelled very similar to an $M/M/n$ queue, where n is the total number of pages (can be thought of as n servers) in that file server
- The arrival (request to acquire a page) to the entire file system is at random and follows a Poisson statistical distribution with mean λ per unit time. The request goes to one of the file servers with equal probability. Therefore, the arrival rate for each file server is λ/m , where m is the total number of file servers in the system. There is another type of arrival to each file server what is known as the overflow from other file servers. The overflow request is not necessarily random, and it depends on the state of the server it comes from. This is the reason why the model is similar to $M/M/n$ with the exception of overflow
- How long a user holds a page is considered as the service time for that page. The service time for each file server is identical and follows an exponential distribution with a mean of $1/\mu$

When a file server overflows (i.e., no page available when a page request comes in), the request is directed to another server based on an acceptable strategy. Only the two strategies, Random and Best-Fit, are studied for the analysis. For the random strategy, the overflow request goes to one of the rest of the servers chosen at random, while in best-fit case, the overflow request goes to a server who has

Figure 5 : Mean Waiting Time vs Sys. Utilization Factor

Figure 6 : Mean Interoverflow Time vs Sys. Utilization Factor

the maximum number of free pages. If no server is found, having any free page, then the request is sent to the one who has least number of requests in its arrival queue. Note that each server has an arrival queue. The simulation is done for different arrival rates and service times, thus keeping the system utilization factor, ρ , constant. The following parameters are observed and plotted against the system utilization factor:

1. *Mean Waiting Time*: It is defined as the mean time a request must wait in the queue before it gets service. It is computed by taking the total delay for all the arrivals and then dividing it by the total number of observations
2. *Mean Interoverflow Time*: It is defined as the mean time a file server takes to overflow again just after it overflows. In fact, it is the mean time difference between two overflow epochs. The mean interoverflow time is computed by summing all such time differences and then dividing it by the total number of observations

The simulation program is given in the Appendix A. The result shows that the best-fit strategy produces longer mean interoverflow time than the random strategy. The best-fit strategy chooses a server with maximum number of free pages; therefore, the probability that the selected site will overflow soon is less compared to other servers with fewer pages. Another result shows that the mean delay produced by random strategy is larger than that produced by best-fit strategy. The mean delay in random strategy is more due to the uncertainty factors. A request may be required to hop a few times before finding an appropriate destination. The simulation analysis does not take into account the communication delay. Every time a request to acquire a page bounces from a site, it accumulates additional delay in order to reach another server. The total communication delay is a factor

of the total number of hops required and the total number of bus users. Since the expected number of hops in case of random strategy is larger than that in best-fit, we can expect the random strategy to give rise to higher communication delay than the other strategy. The two simulation results (mean waiting time and mean interoverflow time) are plotted and shown in Figure 5 and 6.

5.5 Compaction

Compaction is a very common technique, widely used in memory management. The need for compaction is vital to keep memory ordered and organized when allocation and deallocation are frequent. Compaction technique not only keeps the memory organized but also upgrades the performance of the overall system. This section focuses primarily on the compaction issue governing the design of the distributed file management system.

If we look at a snap shot of the system at any given time, we will observe the following:

- Each file server FS_i has S_i number of empty pages, where $S_i \geq 0, i = 1, 2, \dots, m$
- Each file server FS_i serves as a primary site for FX_i number of files and as secondary site for FY_i number of files
- Each file server FS_i has borrowed P_{ij} number of pages from the server FS_j , where $P_{ij} \geq 0$

In addition to that each server not necessarily has an organized memory because of constant lending, borrowing, allocating and deallocating activities in the system. Snap shots for file servers will be difficult to predict from time to time. Further

more, a file stored in part or in full at a server does not always occupy continuous pages of memory. Therefore, compaction is advisable. The objectives of compaction are the following:

1. To ensure that continuous pages of memory are allocated to any file
2. To ensure that the number of secondary sites for a scattered file is minimized
3. To ensure that sites interact with as fewer sites as possible to minimize communication time and cost

Observe that the objectives (2) and (3) are literally the same, but they convey two different meanings. Objective (2) corresponds to the files where as objective (3) corresponds to the overall activities of a file server.

The compaction issue concerns with internal compaction which involves rearrangement of pages locally at any file server and global compaction which looks after the redistribution of used and unused spaces subject to the objectives just described above. Internal compaction is straight forward, and it simply involves local knowledge. However, global compaction is something non-trivial and must be contemplated. Most compaction problems belong to the class of NP-Complete problems [Sant 84]. Hence, we have focussed on suboptimal or heuristic algorithms for compaction.

Compaction can be done periodically, especially when the load in the system is very low in order not to interfere with the regular activities. It can be thought of as a background activity. There are two ways to carry out compaction in the DFMS; namely,

- Centralized compaction

- Decentralized compaction

Centralized compaction requires the intervention of the directory server. Before initiating compaction, the directory server must know the memory status of each file server. It broadcasts a compaction message as required. Each file server, upon receiving a compaction request will send a status vector along with a quantity representing the number of available pages. The status vector for each server corresponds to the number of pages this server has borrowed from other servers. When the directory server receives all m status vectors, it tries to resolve the mutual dependencies, if any. Mutual dependency tells that if server A has borrowed X pages from server B, and the server B has borrowed Y pages from server A, then the two servers should exchange data stored in $\min(X, Y)$ number of pages in each in order to be free from one another. At the later stage, the directory server, with the help of the information received on available pages in each server, directs the servers in a specific manner to relinquish occupied pages at remote locations.

Distributed compaction is another alternative, probably the most suitable approach of carrying out compaction in a distributed system. It does not require a great deal of effort by the directory server as it is the case with centralized compaction. Each file server, based on its local information such as borrowing and lending table, can perform compaction in a coherent manner.

The decentralized algorithm has two phases. In the first phase, each file server executes the compaction algorithm if it has some free pages. The algorithm is common to all servers. After a finite number of steps, the first phase comes to a halt. At the end of the first phase, the following two things will happen:

1. Some file servers are free from others, i.e. they have free pages but do

not owe others

2. Some file servers have no free page but still owe others

Case two indicates some type of deadlock situation. When this happens, i.e. a deadlock is detected, the second phase of compaction starts. It requires a third party like the directory server to break the deadlock.

5.5.1 Decentralized Compaction Algorithm

The algorithm for the first phase is presented here. The second phase is fairly simple. Assume that all the file servers are numbered from FS_1 to FS_m in addition to their respective FSIDs. These numbers $FS_1, \dots, FS_m$ are used only by the file servers and the directory server. There is one-to-one correspondence between a FSID and a FS number.

5.5.1.1 Notation:

Let $MSG(CR; FS_i, FID, X)$ denote the message received from the file server, FS_i , to release (during compaction) X pages of the file denoted by FID . Also, let $MSG(CD; FS_i, FID, data)$ denote the message from FS_i with data(pages) required for FID .

5.5.1.2 Algorithm for Phase 1

At each file server, FS_i :

When receive $MSG(CR; FS_j, FID, Pages)$

DO

IF (FS_i is a secondary site for FID) THEN

Locate the corresponding pages

Send $MSG(CD; FS_i, FID, Page\ data)$ to FS_j

Mark the corresponding pages free

Update free page counter

Update the lending map

ELSE

```

        Disregard the message
    END-IF
END-DO

```

```

When receive MSG(CD; $FS_j$ ,FID,Page data)
    DO
        Copy data into free pages
        Update borrowing map
    END-DO

```

```

When ready for compaction
    DO
        Step1 : Count number of free pages
        Step2 : Sum each row of the borrowing table
        Step3 : Pick a row, R, with smallest sum
                (Note that entry,  $R_k$ , in R corresponds to  $FS_k$ )
        Step4 : IF (row sum of R  $\leq$  free pages) THEN
                FOR each  $R_k \neq 0$  DO
                    Send MSG(CR; $FS_i$ ,FID, $R_k$ ) to  $FS_k$ 
                    Decrement free pages by  $R_k$ 
                END-FOR
            ELSE
                Choose a largest subset, S, of R such that
                    the sum of entries in S  $\leq$  free pages
                FOR each  $S_k$  DO
                    Send MSG(CR; $FS_i$ ,FID, $S_k$ ) to  $FS_k$ 
                    Decrement free pages by  $S_k$ 
                END-FOR
            END-IF
    END-DO

```

It has been assumed that a server becomes ready to perform compaction when it has sufficient number of free pages. The algorithm starts with at least few servers which have some files scattered and have sufficient amount of empty pages to bring a part or full of scattered file(s) to local memory. The algorithm stops in a finite number of steps in two circumstances as described previously.

5.6 Performance Prediction

This section gives a performance model for the proposed Distributed File Management System. There are many performance indicators which are normally used to characterize any system. In our application, we will concentrate our attention in determining the responsiveness of the system in terms of delay (average waiting time as well as communication delay) for different operations on a file.

5.6.1 The Model

The analysis of the system is based on a simple queuing model. The pictorial representation of the model is shown in Figure 7. We will base our analysis on two types of requests — “acquisition” request for a page and “read/write” request on a file. Our objective is to find the expected waiting time for a successful request of these kind. In particular, we are interested in knowing how long (expected time) it would take to allocate a page in presence of site overflows, or how long it would take to read or write a page in a file.

- Let λ_a be the arrival rate of “acquisition” request to the file system
- Let λ_{rw} be the arrival rate of “read/write” request to the system
- Let μ_{rw} be the mean service time required at any file server to process a “read/write” request. We assume that “read/write” request is handled on page basis for simplicity of analysis. Batch arrivals are permitted but they are treated as series of requests
- Let μ_a be the service time required at any file server to determine if any “acquisition” request (for a page) can be honoured. Essentially, this is the time required to search for a free page and come up with an answer. Recall that a list of free pages is maintained at each file

Figure 7 : The Model

server. Note that μ_a is not the time that a page would stay occupied in the file server if the request is honoured

- Because of random nature of requests, we would assume that λ_a and λ_{rw} follow Poisson distribution. The service times μ_a and μ_{rw} are assumed to follow the exponential distribution

Recall that a file server gets two types of requests, such as “acquisition” and “read/write”. The probability (proportion) of “acquisition” request in presence of these two interleaved requests is $\lambda_a/(\lambda_a + \lambda_{rw})$. Similarly, the proportion of “read/write” request is $\lambda_{rw}/(\lambda_a + \lambda_{rw})$. “Acquisition” requests get mean service time μ_a , whereas “read/write” requests get mean service time μ_{rw} . Therefore, the mean service time at any file server in presence of these two types of requests is

$$\mu = \frac{\mu_a \lambda_a}{(\lambda_a + \lambda_{rw})} + \frac{\mu_{rw} \lambda_{rw}}{(\lambda_a + \lambda_{rw})}$$

- Let ρ_a be the traffic intensity for “acquisition” requests. $\rho_a = \lambda_a \mu$
- Let ρ_{rw} be the traffic intensity for “read/write” requests. $\rho_{rw} = \lambda_{rw} \mu$
- Let $P_b = \text{Prob} [\text{all pages are busy at any file server}]$. P_b is same for all file servers because of same arrival rate, same number of pages and same number of service rate at each of them
- Each “acquisition” request to a file server stays at that server with probability $(1 - P_b)$ or routed to one of the other servers. Each file server gets such overflow request with a certain probability which depends on the strategy of the site selection. For example, in random site selection strategy, each file server would get an overflow request with a probability $1/n$, where n is the number of servers
- Because of possible overflow at each file server, the overflow traffic is a significant amount when the system is loaded. This overflow traffic is added to the overall traffic

Text complete; leaf 72 omitted in numbering

A certain number of visits is necessary for a successful "acquisition" request. This depends on the strategy we choose. For example, it would be one if we choose "best-fit" strategy to reroute the request. It is not the case with "random" strategy. Therefore, for this analysis let us consider the strategy to be "random".

Expected number of visits is, therefore,

$$V = 1(1 - P_b) + 2(1 - P_b)P_b + 3(1 - P_b)P_b^2 + \dots$$

$$(1 - P_b) \sum_{n=1}^{\infty} n P_b^{(n-1)} = \frac{1}{(1 - P_b)}$$

Notice that each "acquisition" request creates $V - 1$ additional requests. Thus,

- Net "acquisition" arrival rate to the system is $\lambda_a V$
- Net "acquisition" arrival rate per server is $\lambda_a V/n$
- Net "acquisition" traffic intensity per server is $\rho_a V/n$
- Net arrival rate of all requests to the file system is, therefore, $\lambda_a V + \lambda_{rw}$
- For each server, the net arrival rate is $(\lambda_a + \lambda_{rw})/n$. Also, the net "read/write" traffic intensity per file server is ρ_{rw}/n

We now have an "open queue network" problem with each file server as an exponential server. Jackson has done analysis on "open queue network" with a set of Poisson queues [Jack 57]. Jackson's theorem states that in a network of queues, each node is an independent server and can be modelled as $M/M/1$ system. In our application, we can model each file server as an $M/M/1$ system with arrival rate $(\lambda_a V + \lambda_{rw})/n$ and mean service time μ , where n is the number of file servers. μ is defined earlier.

Using standard results of $M/M/1$ queues [Klei 76], we can derive expected time to allocate a page and the expected time to process a "read/write" request. Therefore,

Expected time for a "read/write" request is

$$T_{rw} = \frac{\mu}{1 - (\rho_a V + \rho_{rw})/n}$$

Similarly, Expected time to allocate a page is

$$T_a = \frac{\mu V}{1 - (\rho_a V + \rho_{rw})/n}$$

5.6.2 Calculation of P_b

Recall that P_b is used in the calculation of V , the expected number of visits needed per "acquisition" request. By definition, $P_b = \text{Prob} [\text{all pages are busy at any file server}]$. P_b will depend on the arrival rate of net requests at the file server and the mean time a page is occupied at that server.

- Let τ be the mean time a typical page is held in a file at a file server. τ can be assumed to be the mean of an exponential distribution
- Let Λ_i be the net arrival rate at the file server. Let us assume that Λ_i follows a Poisson distribution

$$\Lambda_i = \lambda_a/n + \sum_{j=1}^n \theta_{ji} \Lambda_j \quad (1)$$

The transition probability θ_{ij} depends on the strategy we choose. It is the probability that the overflow request from server j is sent to server i . For example, the transition matrix for random strategy looks like the following:

$$\Theta = \begin{pmatrix} 0 & 1/(n-1) & 1/(n-1) & \dots & 1/(n-1) \\ 1/(n-1) & 0 & 1/(n-1) & \dots & 1/(n-1) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1/(n-1) & 1/(n-1) & 1/(n-1) & \dots & 0 \end{pmatrix}$$

where n is the number of file servers. The rows and columns of the $n \times n$ matrix, Θ add up to 1. For pass-next strategy, where the overflow can be sent to either direction, the transition matrix looks like the following:

$$\Theta = \begin{pmatrix} 0 & 1/2 & 0 & 0 & \dots & 1/2 \\ 1/2 & 0 & 1/2 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1/2 & 0 & 0 & 0 & \dots & 0 \end{pmatrix}$$

Equation (1) can be solved for $\Lambda_1, \Lambda_2, \dots, \Lambda_n$ by using Fixed-Point Iterative Method [Gera 78] with some initial estimates of $\Lambda_1, \Lambda_2, \dots, \Lambda_n$. The iterative process will converge in a finite number of steps with a solution for a given accuracy.

This is again an "open network of queues" problem. By Jackson's theorem, in such a network of queues, each node (file server) can be modelled as an independent server. Therefore, we can analyze each server independently, treating it as $M/M/k$ system with parameters Λ_i and τ . Note that, k is the number of total pages at each server. It has been assumed that each server consists of equal number of pages.

Using $M/M/k$ result from [Klei 76], we have,

$$P_b = \sum_{i=k}^{\infty} P_i$$

Let $\rho_f = \Lambda/k\tau$, where Λ is the steady state solution of equation (1). Then,

$$P_b = \frac{\left(\frac{(k\rho_f)^k}{k!}\right) \left(\frac{1}{1-\rho_f}\right)}{\left[\sum_{m=0}^{k-1} \frac{(k\rho_f)^m}{m!} + \left(\frac{(k\rho_f)^k}{k!}\right) \left(\frac{1}{1-\rho_f}\right)\right]}$$

5.6.3 Accessing a Page

When a request to read a page is received at the primary server, the required page may or may not exist at the primary site. In order to access the page at a secondary site, one extra communication is needed. Let the communication delay between any two entities, X and Y , be denoted by $d(X, Y)$. Note that $d(X, Y)$ and $d(Y, X)$ are the same. To read a page, the bus is accessed at least twice, once by user and once by the file server to send data. Therefore, the minimum delay is at least $2d(\text{User}, \text{FS})$. The total time to read the entire page is the sum of the time to access the page in the disk, the time to transmit the page across the bus, and the delay incurred in accessing the bus. It is assumed that the time to transmit a message is negligible in comparison to the time to transmit a page.

5.6.4 Opening a File

Assume the processing delay like recognizing a message and responding to it when message does not require disk access is negligible. To open a file, the major overhead is the communication delay. The total delay is: $d(\text{user}, \text{DS})$ (to send open request) plus $2 d(\text{DS}, \text{FS})$ (once to broadcast for free pages and once to confirm) plus $m(\text{FS}, \text{DS})$ (one access by each of the m servers to send its free pages) plus $d(\text{DS}, \text{user})$ (to assign the file server).

5.7 Conclusion

This chapter discussed about the primary performance objectives arising in the decentralized file management. Various strategies for site selection and allocation were discussed. We noticed that each strategy had some advantages and some

disadvantages. It would be difficult to compare one strategy with another without making a tradeoff between communication cost and simplicity. Compaction issues were looked-at in this chapter. Compaction could be done independently by each server in a distributed fashion or centrally with the help of the directory server. This chapter also addressed the performance characteristics of the DFMS from an analytical point of view in terms of average time for allocating a free page and reading or writing a page in a file.

Chapter 6

Conclusion.

To conclude the discussion on the decentralized management of the proposed distributed file management system, it is desirable to highlight the main points that we have seen throughout the thesis and at the same time to point out the things that have been overlooked. Furthermore, the types of applications the DFMS is most compatible should also be mentioned. This chapter reviews the contents of each chapter briefly, looks at possible areas of application and mentions about further extensions to the DFMS in future.

The design of a distributed file system is governed by some fundamental principles or the global issues. It is the designer's responsibility to meet most of the essential requirements mentioned in terms of the global issues. File servers or distributed file servers are not always intended for general applications, but they satisfy specific needs as required. Therefore, some of the concepts of file management may not be noticeable in all file servers. From the general issues mentioned in chapter 2, we have looked at object naming, access control, directory management, file sharing, reliability and protection. Files must be uniquely identified by the system generated identifiers, called FIDs or UIDs. An access mechanism can be built into the file identifier to control access to a file. Users can specify access structures at the time a file is created. The directory management is an important issue that can not be overlooked. For a single file server, serving a community of users, the directory function is simple and can be embedded in the file server itself. Situation

of a cooperating set of file servers exists, in which case, a central directory server can be useful. In such cases, bottleneck in directory server may develop. The distributed directory can resolve this bottleneck, but the management of distributed directories can be a difficult task. Directory servers can be designed to provide a wide range of utilities not just maintaining file directories and providing lookup services. A trend for designing universal directory servers is on. One other major purpose of designing a file server is to allow a greater degree of file sharing. By doing so, it is the responsibility of the file server designer to ensure consistent state of information stored, to regulate the access in an orderly fashion and to protect from unauthorized access. This gives rise to the notion of reliability. In face of any hardware or communication failure, the file server should be able to act promptly and handle failures effectively.

In the thesis, chapter 3 surveyed four of the existing file servers, namely the Felix file server, the Woodstock file server, the Cambridge file server and the Xerox distributed file server to describe the approaches taken by each of them in addressing the general issues. The XDFS has a structure completely different from the others in the sense that a set of cooperating file servers along with a directory server gives the illusion of a single logical file system. All these file systems identify their files by unique identifiers, provide flexible access control and suitable directory management. The WFS is not intended for a high degree of file sharing, but FFS, CFS and XDFS allow extensive file sharing. Each of these servers provide a fair level of security and protection and have robust recovery algorithm to cope with failures of various nature. The way in which each server has been implemented and its internal organization has been chosen differs from one another.

The motivation for this thesis comes from the survey of the existing file servers.

It has been observed that none of the file servers assume any limitation on memory space which is obvious, especially with the current technology. There are situations where it is reasonable to assume limited memory. One such situation is the case of the proposed distributed file management system. We have a small distributed system with a set of nodes served by a reasonably high speed communication bus. Some of the nodes serve as what is called "file servers", each with a limited amount of memory. We also have a "directory server" node in the network which implements the usual directory functions. Users in the network create files by sending open request to the directory server. The directory server assigns the user a file server to perform file operations like read, write, allocate, deallocate, etc., on page basis by communicating directly with the file server. Some of these file servers, by assumption of limited memory, may likely run out of space in which case they look for some space in neighbouring or other file servers. In the process of lending and borrowing some of the files remain scattered in the system, but the process remains transparent to the users. To operate under such condition is not very favourable when the communication time and cost are significant factors. Periodic compaction is done to exchange borrowed pages with locally available pages as much as possible. The role of the directory server is very much significant. How it assigns a file server to a user is fairly important to the overall control and performance of the distributed file system.³

Chapter 5 in this thesis involved discussion on various management strategies, compaction algorithms, performance issues and simulation. The three strategies discussed are: best-fit, random-fit, and pass-next. Each strategy has some advantages as well as some disadvantages from practical point of view. Theoretically, the best-fit strategy is superior to the other two strategies, but in practice it may

not be feasible to implement. The interoverflow time for any file server is typically larger in case of the best-fit than the other two strategies. Moreover, the average delay in the system (including the communication delay) has a definite effect of the strategy chosen. The best-fit strategy produces lower delay than the other two. All these strategies concern the allocation of a file server or some resources. A simulation was performed to study certain characteristics like interoverflow time and mean delay in the system, using three file servers, each with a given number of pages. The simulation study enabled us to compare any two strategies and draw a conclusion on the basis of the above two characteristics. Compaction is required periodically to minimize the number of intersite communication and to reorganize internal contents locally for fast access. It can be done as a background activity. There are two ways of doing compaction. They are: centralized compaction which involves the guidance and direction of the directory server, and the decentralized compaction which require each file server to carry out the process independently based on a common set of rules. The proposed DFMS has been modelled using a combination of analytic and queuing model for performance analysis. Approximate bounds on certain performance indices are given wherever applicable.

The type of distributed file management system described in this thesis has several applications. The DFMS fits very well in any word-processing environment where we have a set of small computers connected to a communication media. Files are created every now and then disappear all of a sudden to a remote network database. Another application that can be mapped to this DFMS is the telephone network. Consider the file servers as the telephone exchanges in a city. Each telephone exchange has a capacity of a fixed number of trunks leading to a specific destination. When an exchange receives a request and finds no trunk available to

process the call, it simply routes the call through another exchange. The difference is that there is no concept of compaction in the telephone networks. Moreover, the proposed DFMS is also suitable for academic purposes.

There are some possible extensions to the outlined DFMS. One such extension is the feasibility study of more number of directory servers than just one as described. Another extension is to have the overall directory functions distributed over the file servers by getting rid of the directory server. In any case, the problems like synchronization and consistencies will have to be looked at more carefully.

The performance of the proposed distributed file management system is centered around two main criteria - time and cost due to communication and compaction. The development and mass production of sophisticated VLSI technology used in network communication may reduce communication cost and time. Reduction in communication cost and time would result in further compaction in distributed file systems.

Appendix A

This program simulates the situation where we have three file servers, each having five pages. A specific arrival rate and service time has been chosen for the simulation. The statistics is collected for a given number of events. The major characteristics to observe in this simulation are the behavior of average waiting time in the queue and the behavior of interoverflow time. This simulation program is based on the random strategy for routing overflow traffic. When overflow occurs in any given server, it is redirected to a server at random.

However, in case of a different strategy, the routing part in the simulation is simply modified. In particular, for the best-fit strategy, the overflow is sent to the server with maximum number of available pages or with smallest queue size when no server has any free page.

{ ————— Simulation Program ————— }

```

program queue(input,output);
const  maxservers = 3;
       maxpages   = 5;
type   qentry = qrecord;
       qrecord = record
               atime : real;
               next  : qentry
           end;
       qtype = array[1..maxservers] of qentry;
       qlength = array[1..maxservers] of integer;
       ltype = array[1..maxpages] of boolean;
       ptype = array[1..maxservers,1..maxpages] of real;
var    qfs, pqfs : qtype;
       qsize, psize : qlength;
       x, y : qentry;
       avail : ltype;
       tpage : ptype;
       fpage : array[1..maxservers] of integer;
       totque, overflow : array[1..maxservers] of integer;
       past, toverflow : array[1..maxservers] of real;
       over : array[1..maxservers] of boolean;
       arrive, dest, pindex, totevent, count : integer;
       total, i, j : integer;
       pmean, stime, emean, tstep, time, u, r, rho : real;
       wtime, totwait, aveque, avetime : real;
       totwinque, avewinque : real;

```

{This procedure initializes the simulation parameters }

```

procedure initialize;
var i, j : integer;
    x : qentry;

begin
    totevent := 0;
    totwait := 0.0;
    totwinque := 0.0;
    for i := 1 to maxservers do
        begin
            for j := 1 to maxpages do
                tpage[i,j] := 0.0;
            totque[i] := 0;
            over[i] := false;
            toverflow[i] := 0.0;
            qsize[i] := 0; psize[i] := 0;
            overflow[i] := 0;
            past[i] := 0.0;
            new(x), x.next := nil;
            x.atime := x
        end;
    end;
end;

```

{This procedure generates arrival using the poisson distribution for a given arrival rate }

```

procedure poisson(mean : real; var p : integer);
var prod, u, b, r : real;

begin
    p := 1; prod := 1.0;
    b := exp(-mean);
    u := random(r);
    prod := prod * u;
    while (prod ≥ b) do
        begin
            u := random(r);
            p := p + 1;
            prod := prod * u
        end;
    p := p - 1;
end;

```

{This procedure generates service time using exponential distribution for a given mean }

```

procedure expon(mean : real; var e : real);
var u, r : real;
begin
  u := random(r);
  e := -mean * ln(u);
end;

```

{ This procedure examines a list of pages, some free and some occupied and returns, and returns the index of the first available page }

```

procedure findindex(var avail : ltype; var index : integer);
var i : integer;
    found : boolean;
begin
  found := false;
  i := 1;
  while ((found = false) and (i ≤ maxpages)) do
    begin
      if (avail[i] = true) then
        begin
          found := true;
          avail[i] := false;
          index := i;
        end
      else
        i := i + 1
      end;
    end;
end;

```

{ This procedure inserts an event at the end of a given event queue }

```

procedure insert(x : qentry; var inqueue : qtype;
                i, num : integer);
var index : qentry;
begin
  if (num = 0) then
    inqueue[i]^next := x
  else
    begin
      index := inqueue[i];

```



```

        while (index^.next ≠ nil) do
            index := index^.next;
            index^.next := x;
        end;
    end;
end;

```

{ This procedure assigns the events into the respective queues. When an arrival comes in, it is assigned to one of the file server's queue at random. }

```

procedure assign_queue(var queue : qtype; var qsize, totque
    : qlength; time : real; arrive : integer);

```

```

var x : qentry;
    dest, i : integer;
    u, r : real;

```

```

begin
    for i := 1 to arrive do
        begin
            u := random(r);
            dest := trunc(u * maxservers) + 1;
            new(x); x^.atime := time;
            x^.next := nil;
            insert(x, queue, dest, qsize[dest]);
            qsize[dest] := qsize[dest] + 1;
            totque[dest] := totque[dest] + 1;
        end;
    end;
end;

```

{ This procedure counts the number of free pages available in a given file server }

```

procedure count_freepage(var tpage : ptype; var avail :
    ltype; var i, count : integer);

```

```

var j : integer;

```

```

begin
    count := 0;
    for j := 1 to maxpages do
        begin
            if (tpage[i,j] > 0.0) then begin
                avail[j] := false;
                tpage[i,j] := tpage[i,j] - tstep;
                if (tpage[i,j] < 0.0) then begin
                    avail[j] := true;
                    tpage[i,j] := 0.0;
                end;
            end;
        end;
    end;
end;

```

```

        count := count + 1;
    end;
end;
else begin
    avail[j] := true;
    count := count + 1;
end;
end;
end;

```

{ This procedure looks at the overflow queue of a given file server to see if there is any overflow from other servers. If so, it serves one if a page is free. }

```

procedure serve_overflow(var queue : qtype; var avail : ltype;
                        i : integer);

var index^ : integer;
    stime, wtime : real;

begin
    findindex(avail, index);
    expon(emean, stime);
    tpage[i, index] := stime;
    wtime := time - queue[i]^next^.atime + stime;
    totwait := totwait + wtime;
    totwinque := totwinque + time - queue[i]^next^.atime;
    queue[i] := queue[i]^next;
end;

```

{ This procedure looks at the arrival queue of a given file server. For a nonempty queue, it searches for an empty page. If it finds one then the request gets service; otherwise, it is an overflow situation. If there is an overflow, then the request is sent to another file server based on a given strategy (random in this case). For the server, overflow time is recorded for use in mean interoverflow time calculation. }

```

procedure serve_queue(var qfs: qtype; var qsize : qlength;
                    var avail : ltype; i : integer);

var y : qentry;
    dest, index, k : integer;
    u, stime, wtime : real;

begin
    new(y); y^.atime := q[i]^next^.atime;

```

2 of/de 2

```

y^.next := nil;
q[i] := q[i]^.next;
qsize[i] := qsize[i] - 1;
if (fpage[i] > 0) then
  begin
    findindex(avail,index);
    expon(emean,stime);
    fpage[i] := fpage[i] - 1;
    tpage[i,index] := stime;
    wtime := time - y^.atime + stime;
    totwait := totwait + wtime;
    totwinque := totwinque + time - y^.atime;
    dispose(y);
  end
else
  begin
    over[i] := true;
    overflow[i] := overflow[i] + 1;
    toverflow[i] := toverflow[i] + time - past[i];
    past[i] := time;
    u := random(r);
    k := trunc(u * (maxservers - 1)) + 1;
    dest := (k + i) mod maxservers;
    if (dest = 0) then dest := maxservers;
    insert(y,qfs,dest,qsize[dest]);
    qsize[dest] := qsize[dest] + 1;
  end;
end;

```

———— Main Program ————

{ This program simulates for a set of arrival rate and mean service time. From the collected statistics, it computes the necessary characteristics like mean time spent in the system, mean waiting time in the queue, and the mean interoverflow time for each server. }

```

begin
  readln(pmean,emean,tstep,total);
  while (pmean ≠ 0) do
    begin
      rho := (pmean * emean) / (maxservers * maxpages);
      writeln('mean arrival rate is ',pmean);
      writeln('mean service time is ',emean);
      writeln('utilization factor is ',rho);
    end
  end

```

```

time := 0.0;
initialize;

while (totevent ≤ total) do begin
    poisson(pmean * tstep, arrive);
    totevent := totevent + arrive;

    if (arrive > 0) then
        assign_queue(qfs, qsize, totque, time, arrive);

    for i := 1 to maxservers do begin
        count_freepage(tpage, avail, i, count);
        fpage[i] := count;
        if (over[i] = false) then past[i] := time;
        if (qsize[i] > 0) then
            serve_queue(qfs, qsize, avail, i);
        end;
        time := time + tstep;
    end;

    avetime := totwait / totevent;
    writeln('average time in the system is', avetime);
    avewinque := totwinque / totevent;
    writeln('average waiting in queue is', avewinque);

    for i := 1 to maxservers do
        if (overflow[i] > 0) then begin
            writeln('overflow in server', i:3, 'is', overflow[i]);
            avetime := toverflow[i] / overflow[i];
            — writeln('average interoverflow time for
                server', i:3, 'is', avetime);
        end;
    readln(pmean, emean, tstep, total);
end.

```

Glossary

Atomic Action

A transaction is an atomic action : if it completes successfully, it will move the system to a new consistent state; if it fails, the result will be the same as if the transaction had never been started.

Commitment

The completion of a transaction is called commitment. It involves coordination of permanent changes to stored objects by an atomic action when that atomic action completes successfully.

Concurrency Control

Preventing interference among users who concurrently attempt to update some common data. Three well-known approaches to concurrency control are : Timestamp Ordering, Two-Phase Locking and Optimistic method.

Integrity

It refers to accuracy, correctness or validity of the data under the system's control and protection.

Internal Consistency

Requires that each copy of the data (file) remain consistent within itself.

Mutual Consistency

Requires that all copies of a data (file) be identical. Though it is not possible for the copies to be identical at all times, they must converge to the same final state if all user activities were to cease.

Object Recovery

Restoration of objects used by an atomic action to their previous state when

that atomic action aborts.

Recovery

It is the mechanism by which the system (or data) is returned to some known consistent state after some failure has corrupted the present state.

Reliability

Avoiding the loss or corruption of data due to device malfunction or software failure. It is also defined as a measure of the success with which the system conforms to some authoritative specification of its behavior.

Security

Protection of data and its contents from unauthorized access, change or destruction.

Two-Phase Commit Protocol

The basic idea of this protocol is to determine a unique decision for all participants with respect to committing or aborting all local subtransactions. The protocol has two phases. The goal of the first phase is to reach a common decision; the goal of the second phase is to implement this decision.

References

- [Andr 79] Andrews, G.R., McGraw, J.R., "Access Control in Parallel Programs," *IEEE Trans. Software Engineering*, Vol. SE-5, Jan 1979.
- [Birr 80] Birrell, A.D., Needham, R.M., "A Universal File Server," *IEEE Trans. Software Engineering*, SE-6, 5, Sept 1980, 450-453.
- [Bern 81] Bernstein, P., Goodman, N., "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981.
- [Bosw 85] Boswell, F.D., Cowan, D.D., Grove, T.R., "A Distributed File-Server for a personal computer system," University of Waterloo Technical Report, Jan 1985.
- [Dion 80] Dion, J., "The Cambridge File Server," *ACM Operating Systems Review*, Oct. 1980.
- [Elli 83] Ellis, C.S., Floyd, R.A., "The RØE file system," *Proc. 3rd Symp. Reliability Distrib. Software Database Systems*, Oct. 1983.
- [Frid 81] Fridrich, M., Older, W., "The FELIX file server," *Proc. 8th Symp. Operating System Principles (SIGOPS)*, Dec. 1981, pp. 37-44.
- [Gera 78] Gerald, C.F., "Applied Numerical Analysis," Addison-Wesley Publishing Company, 1978.
- [Glig 80] Gligor, V.D., Shattuck, S.H., "On deadlock detection in distributed system," *IEEE Trans. Software Engineering*, Vol. SE-6, No. 5, pp. 435-440, Sept. 1980.
- [Jack 57] Jackson, J.R., "Networks of Waiting Lines," *Operations Research*, Vol. 5, 1957.

- [Klei 75] Kleinrock, L., "Queuing Systems," Vol. 1, John Wiley & Sons, 1975.
- [Kohl 81] Kohler, W., "A survey of techniques for synchronization and recovery in decentralized computer system," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981.
- [Kung 81] Kung, H.T., Robinson, J.T., "On optimistic methods for concurrency control," *ACM Trans. Database Systems*, Vol. 16, No. 2, June 1981.
- [Leac 82] Leach, P.J., Stumft, B.L., Hamilton, J.A., Levine, P.H., "UIDs as Internal Names in a Distributed File System," *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1982.
- [Lude 81] Luderer, G.W.R., Che, H., Haggerty, J.P., Kirslis, P.A., Marshall, W.T., "A distributed UNIX system based on a virtual circuit switch," *Proc. 8th Symp. Operating Systems Principles*, Dec. 1981.
- [Lunn 81] Lunn, K., Bennett, K.H., "A Highly Reliable Distributed File Store Directory System,"
- [Mena 79] Menasce, D.A., Muntz, R.R., "Locking and deadlock-detection in distributed databases," *IEEE Trans. Software Engineering*, Vol. SE-5, No. 3, May 1979.
- [Metc 76] Metcalfe, R.M., Boggs, D.R., "Ethernet : Distributed packet switching for local computer networks," *Comm. ACM*, 19, 7, July 1976.
- [Mitc 79] Mitchell, J.G., Maybury, W., Sweet, R.E., "Mesa language manual," Report CSL-79-3, Xerox PARC, Palo Alto, Aug. 1979.
- [Mitc 82] Mitchell, J.G., Dion, J., "A comparison of two network-based file servers," *Comm. ACM*, Vol. 25, pp 233-245, April 1982.

- [Need 82] Needham, R.M., Herbert, A.J., "The Cambridge Distributed Computing System," Addison-Wesley Publishers Limited, 1982.
- [Rich 69] Richards, M., "BCPL : A tool for compiler writing and system programming," *AFIPS SJCC Conference Proceeding*, 35, 1969.
- [Ritc 78] Ritchie, D.M., Thompson, K., "The UNIX Time-Sharing System," *Bell System Technical Journal*, 57, 6, Part 2, July 1978.
- [Sant 84] Santoro, N., Thomas, D., Zaks, S., "Some hard problems in decentralized management of a distributed file system," School of Computer Science, Carleton University, August 1984.
- [Stan 84] Stankovic, J.A., "A Perspective on Distributed Computer Systems," *IEEE Trans. Computers*, Vol. 33, No. 12, Dec. 1984.
- [Stur 80] Sturgis, H., Mitchell, J., Israel, J., "Issues in the design and use of a Distributed File System," *SIGOPS Operating System Review*, July 1980.
- [Svob 84] Svoboda, L., "Resilient Distributed Computing," *IEEE Trans. Software Engineering*, Vol. 10, No. 3, May 1984.
- [Swin 79] Swinehart, D., McDaniel, G., Boggs, G., "WFS: A simple shared file system for a distributed environment," *Proc. 7th Symp. Oper. Syst. Princ.*, Dec 1979.
- [Tane 81] Tanenbaum, A.S., "Computer Networks," Englewood Cliffs, NJ., Prentice-Hall, 1981.
- [Wilk 79a] Wilkes, M.V., Needham, R.M., "The Cambridge CAP computer and its operating system," *Operating and Program System Series*, North Holland, 1979.
- [Wilk 79b] Wilkes, M.V., Wheeler, D.J., "The Cambridge digital communication ring," *Proc. Local Area Comm. Network Symp.*, Boston, May 1979.

END

2008.88

FIN