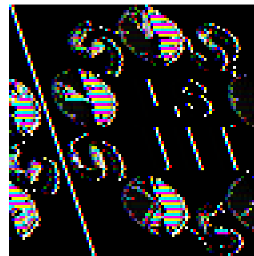
Diplomarbeit

The Implementation of the File Service Module for the Distributed File System "Dragon Slayer III"

Veye Wirngo Tatah



Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

18. October 2001

Supervisor
Dipl.-Inform. Mario Lischka
First Reader
Prof. Dr. Horst F. Wedde
Second Reader
Prof. Dr. Krumm

Abstract

This thesis discusses the design and implementation of a distributed file service for the Dragon Slayer system. Dragon Slayer is a distributed and decentralized file system. Distributed and decentralized systems offer the potential for a degree of concurrency, modularity and reliability higher than that which can be achieved in a centralized system.

In the Dragon Slayer system, all the services are autonomous but they communicate with other services resident at the same machine or on different machines through the exchange of messages. The file service for example needs information from other services present in the system in order to carry out it's various functions. The challenging aspect of this work lies in the design of a distributed file service, whereby communication between the services is very important for the system to function correctly. At the same time, communication overhead should be minimized because this can degrade the performance of the system. A distributed file service is used to support the sharing of persistent storage and of information. The file service will provide file operations to clients and application programs to used when accessing and manipulating data stored in the system. To improve data availability in the system, file replicating across multiple, wide-area sites will also be implemented. File replication does not only bring advantages to the system, one has to deal with the problem of maintaining data consistency among the replicas.

In this work, other issues dealt with include working with partitioned files and with extremely large files. These features are important because Dragon Slayer was designed to support team work and application programs, based on computer-supported cooperative work (CSCW). Trends toward team work are visible in almost all companies today.

To realize the functionalities listed above, a file service module for the Dragon Slayer system have been designed and implemented.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Overview 3
1.3 Problems to be solved 4
1.4 Organization of this work 6

2 An overview of the Dragon Slayer III system **9**
2.1 The Dragon Slayer III architecture 9
2.2 Partitioning of files 11
2.2.1 Motivation 11
2.2.2 Overview 11
2.2.3 Previous concepts on parts and fragments 12
2.3 Command Manager 14
2.4 Node Manager 15
2.5 Services 15
2.6 Client Interface 16

3 An overview of other File Systems **17**
3.1 The Sun Network File System (NFS) 17
3.2 The Amoeba File system 19
3.3 The Andrew File System (AFS) 22
3.4 The Virtual File Switch Interface 24

4 Related works on storage management **27**
4.1 The Network Storage Pool in the Global File System 27
4.1.1 Resource Groups 29
4.1.2 Device Locks 29
4.1.3 Memory Hierarchy 30
4.2 The Volume Manager in the Extended File System (XFS) 30
4.2.1 Storage Scalability 32
4.2.2 Allocation Groups 32
4.2.3 Performance Scalability 33
4.3 Logical Volume Manager (LVM) 34

5 Related works on the Replication of files 37
5.1 Replication Design Alternatives 37
5.1.1 Conservative vs. optimistic update 38
5.1.2 Client-server vs. peer-to-peer 38
5.1.3 Immediate propagation vs. periodic reconciliation 38
5.1.4 A continuous consistency model 38
5.2 Update Protocols 39
5.2.1 Primary copy 39
5.2.2 Voting or simple majority 39
5.2.3 Quorum based voting or weighted voting 40
5.3 Replication strategies in other file systems 41
5.3.1 Replication in the Ficus Distributed File Systems 42
5.3.2 Replication in the Harp File System 44

6 Related works on file version management 47
6.1 Related works on file version management 47
6.1.1 File version control in the Global File System 47
6.1.2 File version control in the Amoeba File Service 48
6.1.3 The shadow pages 48
6.2 Mutual Exclusion and Concurrent accesses 48

7 Concept and Design 51
7.1 Designing the File Service 51
7.1.1 User requirements 52
7.2 Stateless service 53
7.3 Data modification in Dragon Slayer 54
7.4 Version consistency mechanism 56
7.5 File replication 57
7.5.1 Replication On Demand algorithm 57
7.5.2 Algorithm for reconciling the replicas of a file 58
7.5.3 Replication control 59
7.6 Crash recovery 60
7.7 Components of the Dragon Slayer system 60
7.7.1 Client Interface 60
7.7.2 Command Manager 61
7.7.3 Node Manager 62
7.8 Partitioning of files 63
7.8.1 Composition and decomposition of a file 63
7.8.2 Motivation 63
7.8.3 Boundary definition 63
7.8.4 Composition of parts 64
7.9 Manipulating with parts of a file 64
7.9.1 Reading data from the various parts of a file 65

7.9.2 Writing data to the various parts of a file 65
7.9.3 Deleting data from a partitioned file 66
7.10 Manipulating with a fragmented file 66
7.10.1 Reading from a fragmented file 66
7.10.2 Writing data to a fragmented file 67
7.10.3 Deleting the fragments of a file 68

8 The File Service Module Implementation 71
8.1 Tools used for the implementation 71
8.2 The File Service Modules 72
8.3 Client Program 75
8.4 The File System Commands 78
8.5 Command line parsing 82
8.6 The Open File Table components 83
8.7 The File System Requests 85
8.8 The File System Service Interface 88
8.9 Creating files and fragments 88
8.10 File replication and relocation 91
8.11 Testing the modules 99
8.11.1 Testing the communication classes 99
8.11.2 Working with log files 101
8.11.3 Interactive test 101

9 Conclusions and future work 103
9.1 Quality of Service 103
9.2 Parts and fragments 105
9.3 Integrating VFS, the File System and LVM 105
9.4 File version management 107
9.5 Replication 107

List of Figures

1.1 Two file servers at site A and B 3

2.1 The Dragon Slayer III with the client API 10
2.2 An example of a partitioned file 12
2.3 Dividing a file into three parts 13
2.4 An example of the Dragon Slayer III components 14

3.1 The architecture of NFS 18
3.2 The Amoeba architecture 20
3.3 The architecture of AFS 23
3.4 The VFS, the file system and the LVM 25

4.1 The GFS cluster 28
4.2 The XFS Architecture 31
4.3 The Logical Volume, Volume Groups and Physical volumes 36

5.1 The primary copy method 40
5.2 The simple majority voting 40
5.3 The Quorum based voting when reading from a file 41
5.4 The Quorum based voting when writing to a file 42
5.5 Ficus Stack of replication layers 42
5.6 Harp system structure 44

7.1 Sequence diagram for the open command 69
7.2 The example shows how the system stores parts of a file at different nodes 70
7.3 This is an example of a fragmented file 70

8.1 Class diagram illustrating the communication between the different modules 74
8.2 The client module 76
8.3 The class diagram shows the file system commands 78
8.4 Class Diagram for the Open file table components 84
8.5 Class diagram for the Request 86
8.6 Class diagram showing the replies 87

8.7 The class diagram illustrates the file system service Interface . . . 89
8.8 Class diagram illustrates the create method 90
8.9 Sequence diagram for the create method 92
8.10 Class diagram for replication and relocation 93
8.11 Collaboration diagram for relocating a file 95

9.1 The integration of the LVM, the VFS and the File System 106

Acknowledgments

It is my privilege in this brief but sincere statement to thank my family and friends who have helped me directly and indirectly, in the pursuit of the work described in this thesis. Special thanks to my mom Kinyuy Tatah and my dad Tumi Tatah for the upbringing and the opportunities they've given me in life. My special thanks also goes to my husband Vincent and my sons Verki and Doh for the moral support and understanding they gave me during these period. I would also like to thank Irmgard, Dietmar, Matthias and Christoph for supporting me throughout my stay in Germany, you'll always be in my memory.

My special thanks goes to Prof. Horst Wedde for making it possible for me to do my thesis in his department. I would also like to thank my supervisor Mario for giving me this opportunity and the necessary support in completing my work. Not forgetting the friends I made at the department especially Wolfgang who always took time and patience to answer my questions.

Lastly I would like to mention that this work did broaden my experience and I am very happy to have completed it.

Chapter 1

Introduction

This thesis presents the design and implementation of a distributed file service for the Dragon Slayer III system. A set of file operation primitives will be provided which will enable application programs to access and manipulate data stored in the system. Other topics to be discussed in this work are file partitioning, replication of files and working with extremely large files.

1.1 Motivation

A distributed system is a collection of heterogeneous computers and processors connected via network. A distributed computer system offers the potential for a degree of concurrency, modularity and reliability higher than that which can be achieved in a centralized system. Centralized systems do not scale well, changes in the system usually require global synchronization. In centralized systems, a decision is made in a central mechanism and transmitted to executive components while in decentralized systems, each executive component makes its own decision.

The main reason for the present trend towards distributed and decentralized systems is that these systems potentially have a much better price/performance ratio than a single large centralized system would. The attractions of such a system are widely recognized: each user has autonomy, control over the fate of his own resources, but each user benefits from sharing. These systems are also gaining popularity because of higher reliability, shorter response time and higher throughput, extensibility and incremental growth, better flexibility in meeting the users needs. In such systems, the responsibility of coordination and scheduling is distributed to all system components.

Designing a file service for a distributed system is more difficult than designing one for a centralized system. In the case of centralized systems, access to complete and accurate information about the system environment is stored at a central site. The difficulties to be faced when designing a service for a decentralized system are: Ineffective decision making must be avoided, internal conflicts should also

be minimized and the lack of authority over all the sites lead to data consistency problems.

To add more features to a distributed system, other issues to be discussed in this thesis are file partitioning and working with extremely large files. File partitioning is a process, where a file is split-ted into many independent parts, each containing a portion of the file's data. Each part can then be processed in parallel by a separate task to let one perform large computational tasks faster and more efficiently. Each part can be accessed independently, governed by individual authorization profiles regarding user, group or role access. This phenomena can be useful in administering large projects, where different users, having different access rights can work simultaneously on a project file which has been splitted into many parts. These parts can later on be combined to a single project file if desired. The key aspects of supporting file partitioning were the increased availability of computer networks and the trend towards team work and concurrent work. Activities in this domain are known by the notions of group-ware or computer-supported cooperative work (CSCW).

The system can further split large files into fragments which is transparent to the users and stores them locally on different disk partitions. A file is said to be fragmented, when it is stored on disk not as a whole file but rather broken into scattered parts. These can be done for two reason: The first reason is to overcome storage limitations and the second reason is to improve the rate of data access, since smaller files are faster to access than larger files.

The basic architecture of Dragon Slayer III for a distributed and decentralized system has led to the existence of many autonomous services, with dedicated functions. This services have to communicate with other services resident at different nodes, when processing users commands. The challenge to be faced by the file service in such a distributed environment when designing and implementing the file operations, is that of maintaining data consistency among the files due to the multiple data access approach used in the system.

In order to increase reliability and availability of data, file replication will also be discussed. The replication problem has received particular attention because of the central role it plays when designing a highly distributed file system. The file service will be in charge of creating the replicas and maintaining data consistency among these replicas before and after each transaction.

Most of the existing file system were designed as client/server models. In such systems, reliability and availability can not always be guaranteed, because of its centralized nature. Some of the aspects discussed above have already been implemented in some systems, but it is rare to find a file system that supports all these features. Some of the file systems for example, have not implemented replication nor do they directly support CSCW applications. Dragon Slayer tries to put together most of the important features necessary for a file system to function correctly and to guarantee that the users commands are processed to their satisfaction.

1.2 Overview

A file system is the most visible aspect of an operating system which is designed to store and manage large numbers of files, thereby facilitating their accessibility and modification.

It is often difficult to differentiate between a File Service and a file server. The file service is the specification of what the file system offers to its clients. A true file service is concerned with the operations on individual files, such as reading, writing, and appending. A distributed File Service is used to support the sharing of persistent storage and of information. Figure 1.1 illustrate the file service and the file server.

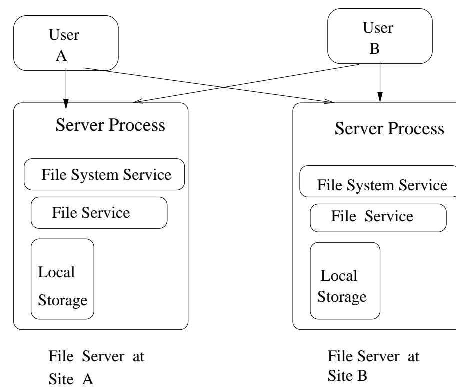


Figure 1.1: Two file servers at site A and B

A file server on the other hand is a software product, which acts as a repository for shared files and programs. It is a process that runs on some machine and helps implement the file service. File server machines store the files in the distributed file system, and a server process running on the file server machine delivers and receives files from its clients.

A distributed File Service fulfills a function similar to the file system component in conventional operating systems. The File Service is usually the most heavily-used service in a general-purpose distributed system, so its functionality and performance are critical.

In DRAGON SLAYER the File Service works intensively with the Directory Service and the Resource Manager. The File Service implement operations on the content of files, such as opening a file, reading from a file etc. When the File Service intends to carry out any file operation, it has to know where the data is located on disk. It sends a request to the directory service to locate the data. After the data has been located, the File Service needs to request locks from the Resource Manager before processing can start. The File Service needs the Resource Manager to control concurrent data access; For example, if a file has been requested for reading, no other process should request this same file for

writing. In Dragon Slayer, the services are autonomous but they need other services in order to carry out their functions properly.

A distributed File Service should support file replication, the sharing of persistent storage and the sharing of files. The file service should also support transparency requirements for distributed systems such as access transparency, location transparency, concurrency transparency and performance transparency as stated in [Zas99].

Presently there are so many distributed file systems, and each tries to improve upon it's performance, reliability, availability and portability. Each file system has it's own pros and cons. They have been designed differently depending on the applications they support. During my research, I'll also take a look at the file service designs in other file systems. The Network File System for example does not support file replication [Sun95], we'll examine how NFS implements data availability. Amoeba on the other hand, offers only operations for creating, reading and deleting files [VRsTK90].

1.3 Problems to be solved

In this section, I'll outline the requirements for the file service for Dragon Slayer III, to be dealt with in this thesis. A brief discussion on the approach to be used when solving the problems will be presented. Due to the design and architecture of Dragon Slayer, certain issues have to be taken care of when designing the file service. Dragon Slayer employs strong consistency in data maintenance.

The majority rule is also used to support these approach. With the majority rule, more than 50 % of the replicas must be present before data modification can take place. This rule holds, if more than 50 % of the replicas present have the same version number and the same replication grade. The File Service commands must be in the position of detecting that a node has failed and can no longer response to the various requests issued by the commands. This can be caused by network problems or hardware failures. The services to be offered by the file service, should always be available to the clients even if one or two nodes in the system have failed.

As mentioned earlier, the File Service will provide file operations for applications to access and manipulate data stored in the system. The file operations to be offered by the File Service must guarantee that after a request has been completed, data consistency amongst the replicas must be maintained. The following file operations will be supported: opening, reading, writing, creating, closing, deleting, copying, replicating and relocating of data.

Each of the file operations has certain criteria which have to be taken into consideration separately, especially as we are dealing with a distributed and decentralized system. The main problem to be tackled here is that of applying file

operations from clients to multiple replicas in a consistent way.

Opening a file When using session semantics, it is impossible to read from or write to a file without having opened it first. That implies that an open file table will be needed in the system to register the opened files. Managing the information stored in the open file table will be a complex issue, because of the decentralized structure of DRAGON SLAYER. When opening a file for a read or a write, the File Service has to acquire read and write locks from the Resource Manager. Writing to a file can cause data inconsistency, therefor when opening a file for writing, exclusive locks will be acquired.

Reading from a file When a read request is issued by the client, any replica can be used to serve the clients request because all replicas are equal. Multiple processes can access a single file for reading.

Writing to a file To avoid data inconsistency, the write operation has to take place simultaneously on more than 50 % of the replicas of a file present in the system. The write transactions must complete successfully on all these replicas before a commit request is issued or else a rollback request will be issued to overturn the changes made on the replicas.

Creating a file When creating a file, a lookup request has to be sent to the directory service to verify the existence of the file. If the file does not exist, a lock request will be sent to the resource manager to lock the path of the file before the create request is processed.

Deleting a file When deleting a file, the File Service needs the cooperation of the Directory Service and the Resource Manager to locate and lock the files before deleting them.

Copying a file This involves copying a file within a node from one path to another. All replicas on other nodes will also be copied.

Replicating a file The management of the read/write replication scheme is of utmost importance because it leads to the problem of maintaining data consistency among replicas in such a decentralized system. The problems to be dealt with here are detecting data inconsistency among replicas and propagating updates to outdated replicas. The File Service will concentrate on version consistency while the Resource Manager will take care of mutual consistency in the system. Replication of data is an elaborate topic which needs to be taken care of by a dedicated Replication Service.

Relocation of data In order to regulate the network traffic and improve on load balancing, files will be moved from one node to another if desired.

Closing a file After the transactions on a file have been completed, the file entries in the open file table will be removed and the locks acquired will also be released before the file is closed.

A file version management scheme will be needed by the File Service in order to detect inconsistency among data.

Another issue to be dealt with here is the management and manipulation of partitioned files. I'll be revisiting this problem which had already been discussed in the thesis of [FR95]. The main issues to be dealt with here are:

- Which approach should be considered when decomposing and composing parts and fragments.
- What procedures would be used when accessing a partitioned file, whose parts are stored at different nodes.
- How would a fragmented file be accessed, which will be stored on different disk partitions all on the same node.

This master thesis will also design and implement a client program, which will be the interface for the users to access the file system. It will be used to demonstrate the various issues covered in this thesis and in other related works.

1.4 Organization of this work

Chapter 2 In this chapter, the introduction of the Dragon Slayer system with some of its important components will be discussed. Before we proceed, it is necessary to present the Dragon Slayer system in detail, because this will lead to a better understanding of other topics to be dealt with in this thesis.

Chapter 3 After the presentation of the Dragon Slayer system, this chapter will take a look at the architecture and design of three file systems: The Network File System, the Andrew File System and Amoeba File System. The Virtual File System Switch is also dealt with here because it acts like an interface to many file systems in Linux.

Chapter 4 As mentioned before, a distributed File Service is also used to support the sharing of persistent storage and information. This chapter will concentrate on the different storage technologies that have already been implemented. We will examined the different approaches used in the design of the Network Storage Pool of the Global File System, the Volume manager of the Extended File System and the Logical Volume Manager for Linux.

Chapter 5 After data has been stored, the next thing to do is to look for ways of multiplying these data, so that it is available to many users. This chapter deals with the replication of files. Some replication design alternatives already implemented in other file systems and data consistency protocols will be revisited.

Chapter 6 The presence of replicas in a system means that data consistency among the replicas must be maintained. Some systems use mutual consistency while others use version consistency. In this chapter, discussion is based on the different approaches taken by other file systems in dealing with File version management.

Chapter 7 After presenting background information and related works concerning file systems and the File Service, this chapter will finally present the design of the File Service and the solutions of the problems mentioned in section 1.3.

Chapter 8 When a system has been designed, the next step is the implementation. This chapter concentrates on the implementation of the file operations as described in section 1.3 and in chapter 7. Class diagrams and sequence diagrams will be used for illustration purposes. The client program will also be discussed.

Chapter 9 This chapter summarizes main findings and ideas for future works.

Chapter 2

An overview of the Dragon Slayer III system

This chapter will present a detailed background information to the Dragon Slayer III system and it's components. Section 2.1 will presents the basic architecture of the Dragon Slayer system. In section 2.2 discussions will be on file partitioning. Section 2.3 will explain the functions of the Command Manager and section 2.4 will concentrate on the Node Manager and the agents. In section 2.5, the different services present in Dragon Slayer will be presented. The client interface will be introduced in section 2.6.

2.1 The Dragon Slayer III architecture

The Dragon Slayer I project started in 1988 at the Wayne State University in Detroit. The first phase of this project focused on novel distributed file services, particularly efficient to simultaneous access of distributed data, on top of heterogeneous operating system platforms with decentralized control [WL98]. Later on came the Dragon Slayer II project which is now being developed to Dragon Slayer III. The Dragon Slayer III offers a high amount of fault tolerance and the service reliability is guaranteed through the cooperation of autonomous local services.

Dragon Slayer is a decentralized system, where control of resources is not dedicated to a single central instance. The Dragon Slayer environment consists of a set of computers [1], which are connected to each other via networks. The Dragon Slayer system presents a uniform view of the file system to the users independent of their location. The concept of Dragon Slayer is that all the nodes should host the same services offered by the system so as to improve on reliability and availability. The fact that all Dragon Slayer nodes offer identical services, means failure of one node will not cause a great impact on the

[1] referred to here as nodes

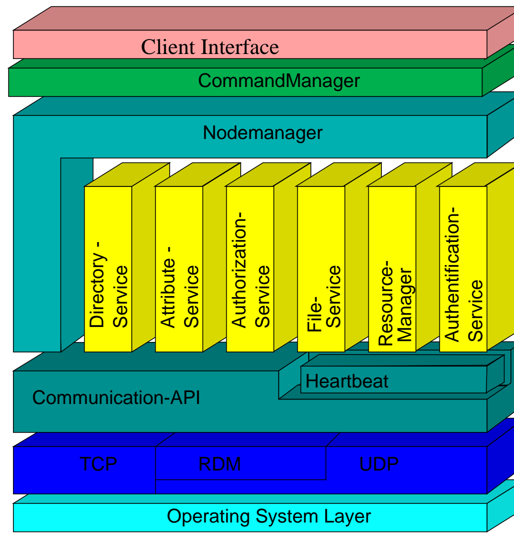


Figure 2.1: The Dragon Slayer III with the client API

users, because they will continue to access the same services from another node in the system. Recovery is much more faster as compared to centralized system, where the failure of the server puts a halt to all processing. Dragon Slayer was designed to support large files and partitioning of files into a variable number of parts. It was important to have a platform on which new application programs that support team work will utilize, without the need of adapting their software.

Objects in Dragon Slayer III can be accessed by a system-wide unique ID, called the vnodeID. The other design goals of Dragon Slayer are:

Data consistency: Dragon Slayer must make sure that the integrity of data is guaranteed.

Transparent access: Processes are allowed to access remote files in the same way as local files.

Concurrency transparency: It will be achieved by implementing a locking mechanism.

The Dragon Slayer III file system also supports location transparency which makes it easier for data objects to be migrated from one node to another

Figure 2.1 shows the system components of Dragon Slayer III. The Command Manager, the Node Manager and the various services listed on the diagram make up the foundation of this file system and are present on each Dragon Slayer III node or machine. As can be seen from the diagram, all the services are independent but they interact with each other by using the different communication facilities offered by the system. The Dragon Slayer III client is a

software that also runs on a node and helps to channel the users request to the Command Manager on that node.

At each node, there is a heartbeat service which provides the node with an up-to-date information on the system. These services send out messages to the nodes which contain a list with service names and their communication ports.

There are three types of communication in DRAGON SLAYER III, broadcast, multicast and unicast. It is accomplished by using the following transport protocols: Transmission control protocol (TCP), User Datagram Protocol (UDP) and Reliably Delivered Messages protocol already dealt with in the Thesis of [Sie98].

2.2 Partitioning of files

2.2.1 Motivation

In Dragon Slayer system, a file can exist as one part or it can be partitioned into many parts. The users decide if their files should be stored in many independent parts or in a single part. The aim of partitioning a file is to make it possible for different users to work simultaneously on each part of the file, that means modification can take place simultaneously on the different parts without causing problems of data consistency. Each part can be assigned to a group of users and each part can have different access rights defined on it [FR95][WL98].

Authorized users can rename, insert new parts or delete some parts without preventing access to other users who may be working on the other parts of the file. This phenomena can be useful in administering projects, where different users can be working on different sections of the project's report, which will later be combined to a single report. These parts on the other hand can be decomposed by the system into fragments if need be as described in [FR95]. This situation can only occur when the system faces lack of storage on a particular machine. When this situation occurs, the system has to fragment that part and store the fragments at another disk with available storage space. It's clear that the storage capacity at the nodes are not identical. Some nodes may have mass storage facilities while others may have just smaller disks.

Parts are made up of one or more fragments. To increase data availability, parts can be replicated likewise fragments. Replication will be discussed again in section 7.5.

2.2.2 Overview

This topic was already analyzed in a previous thesis [FR95]. The aim of this analysis was to implement file management, especially parts of files and their fragments. Fragmentation transparency has always been of great importance to the Dragon Slayer design[WL98]. In this section, the discussions will focus on the

concept of parts in Dragon Slayer II design and the modifications that have been made in regard to Dragon Slayer III. This thesis discusses in-depth the design of implementing parts and fragments in Dragon Slayer III. In the next sections, we will cover the following topics: An overview of the concept in Dragon Slayer II and the changes in Dragon Slayer III.

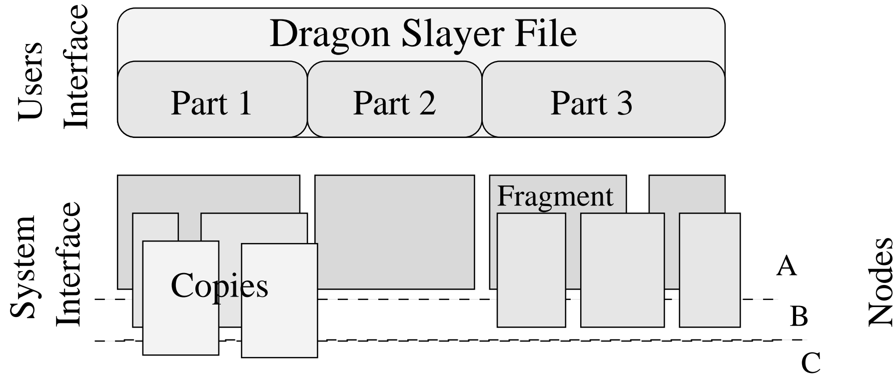


Figure 2.2: An example of a partitioned file

Figure 2.2, shows the structure of a file in Dragon Slayer III. As mentioned above, the main goal is that, users should be responsible for the division of their files. They decide into how many parts the file should be partitioned and they give names to the various parts created. Once these files have been divided, the system only stores and retrieves them, it doesn't alter the number of parts created by the user, but it can further decompose the parts into fragments. The user has no idea whether the file or the parts have been fragmented.

The size of a part can change dynamically. If one part is deleted from a partitioned file or a new part added to it, then the system has to reorganize the numbering of the parts.

2.2.3 Previous concepts on parts and fragments

A part can still be split-ted into a variable number of fragments by the system. All fragments have a unique attribute which makes it easy to fit them together in the right order. The diagram in figure 2.3 shows an example of a partitioned file. The file history.txt has been partitioned into introduction.txt, body.txt and conclusion.txt. History.txt is the main file and it stores information in it's vnode relating to the three parts. Each of the parts could be stored at different nodes.

The user decides the number of parts to be created, and the operating system carries out the users orders. The File Service decides on which node to store the parts and the operating system searches for a suitable place on a specific disk for storing these parts. The system can further split the parts into fragments whereby this process is transparent to the users.

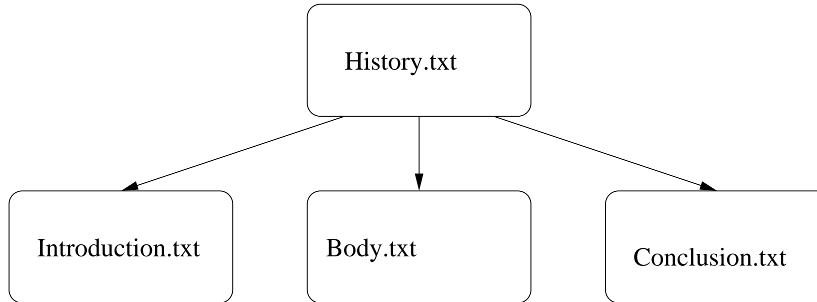


Figure 2.3: Dividing a file into three parts

Fragmentation refers to the condition of a disk in which files are divided into pieces scattered around the disk. Fragmentation occurs naturally when one uses a disk frequently, to create, delete and modify files. At some point, the system needs to store parts of a file in non-contiguous blocks. This is invisible to the users but it can slow down the speed at which data is being accessed because the disk drive must search through different sectors of the disk to put a single file or a part together [SS94].

Our focus here will be on file fragmentation. File fragmentation causes performance problems when reading files, while disk fragmentation causes performance problems when creating and extending files. For more information on internal and external fragmentation, read Siberschatz [SG94]. Neither condition has anything to do with the contents of a file. Attention is focused only on files as containers for data and the arrangement of these containers on the disk.

Concept in Dragon Slayer II

In Dragon Slayer II, the fragments could be stored on different machines. It was also important that all the replicas of a part should have the same number of fragments in order to avoid any shortcomings [FR95]. This means that all the replicas of the parts must have the same number of fragments, whereby the boundaries dividing the fragments of one replica must be the same with the boundaries dividing the fragments of the other replicas. This was to ensure that if one part was not available at a node, the other parts stored at the different nodes could be used instead, to serve the same purpose. Writing to a part could only take place if the majority of the replicated fragments were available in order to ensure data consistency.

Concept in Dragon Slayer III

In Dragon Slayer III, it has been maintained that the parts will be stored at different nodes but the fragments of a file will be stored only at a single node.

That means, the fragments could be stored on different disk partitions being mounted locally at that node. A file can be splitted into varying numbers of fragments differently on each node. When storing parts, the meta-data of each part is also stored with the parts at that nodes. This will ease the process of locating the other parts belonging to a file.

By implementing Partitioning of file in Dragon Slayer, the file system can make good use of the different storage facilities discussed in chapter 4. The advantages of these storage facilities are that files can be placed intelligently according to the characteristics of the disks present in the system.

2.3 Command Manager

The Command Manager is stateful, that means it maintains detailed information about the data objects for example, what operations are being applied and which user is issuing the request.

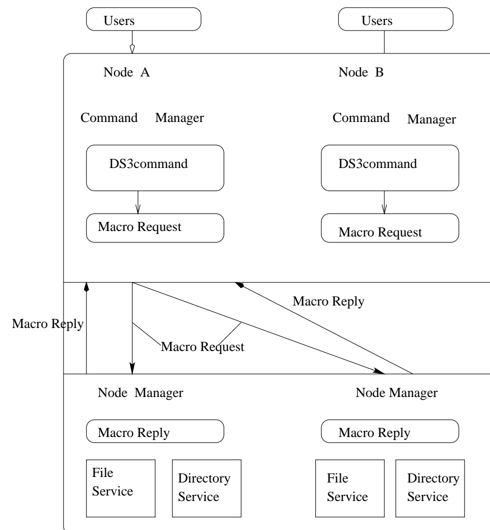


Figure 2.4: An example of the Dragon Slayer III components

Figure 2.4 illustrates the Command Manager, the Node Manager, DS3command, Macro-Request, Macro Reply and some of the services at node A and B. The Command Manager receives the users commands as strings from the clients, evaluates and parse the command line, before it forwards them to a node or to many nodes depending on what the user may have requested. When the users commands arrive the Command Manager, it evaluates the request and creates a DS3command from it. Then these DS3commands are queued for the agents (threads) to remove from the queue and process.

From the DS3commands, Macro-Request are generated. The Macro-Request are broadcasted to all the nodes. The Macro-Request are mostly service calls put together by the various services in Dragon Slayer III. These are the functions that complete the users request. For example the File Service offers a read method to read data from a file. The File Service Interface also offers functions which are called by Macro-Request from other concurrent projects in Dragon Slayer III.

After the nodes have executed the Macro-Request, Macro Replies are sent to the Command Manager. The Command Manager evaluates the Macro Replies to know what actions to undertake. If the request was completed successfully, the Command Manager will forward the message to the user but if the request failed, the Command Manager might decide to repeat it or send a message to the user concerning the request that failed.

2.4 Node Manager

The execution of all the requests in the system takes place at the Node Manager. The Node Manager presents an Interface to all the services in Dragon Slayer III. The Macro-Request received from the Command Manager are executed at the Node Manager. When the Node Manager receives the Macro-Request, they are also queued for the agents to process. As mentioned above, a Macro-Request is made up of service calls. When a Macro-Request has been executed by the agent, it forwards the result of it's execution [2] to the Node Manager. The Node Manager forwards the Marco Reply to the Command Manager which issued the Macro-Request as illustrated in figure 2.4.

2.5 Services

Figure 2.1 illustrates the different services present in Dragon Slayer III. Each service has a particular task it carries out in the system. The services in Dragon Slayer III are stateless. That means all the request should come along with all the information needed for the task to be completed. It is therefore easier for the system to recovery after a break down. In order to carry out the task successfully, each service puts some methods together called service calls.

The services cannot interact with each other directly. They can only issue service calls to the other services. At the moment, the following services are to be implemented in Dragon Slayer III:

1. Directory Service - It provides the required mappings between objects and their unique vnodeID.

[2] the Macro Reply

2. Attribute Service - it manipulates the attributes of the object and can also insert meta-data to each object when given the vnode ID.

3. Authorization Service - it is responsible for managing the access permissions of the users to the objects present in the system.

4. File Service - It enables users to access and manipulate data in the system, by providing a set of file operations. It is also responsible for the replication of data objects.

5. Resource Manager - is responsible for maintaining mutual consistency in the system. A locking mechanism will be implemented to control concurrent data access. It locks the objects when requested by other services before the operations can proceed and releases the locks after the operations have been completed.

6. Authentification Service - it is in charge of authenticating the users who want to access to the system.

2.6 Client Interface

The client program gets the arguments passed with the users commands to the system. It is the only interface for the Dragon Slayer file system to the outside world. The diagram in figure 2.1 illustrates the location of the client interface. Detailed information on the client interface will be presented in chapter 7 and the client program will be described in chapter 8.

Chapter 3

An overview of other File Systems

This chapter will present a brief summary of the design of three distributed file systems. File systems are designed to store and manage large numbers of files, with facilities for creating, naming and deleting files. They are also responsible for the control of access to files. When carrying out research in this topic, I came across many existing file systems. I finally decided to choose the Network File System (NFS), Amoeba file system and the Andrew File System (AFS) for case study. NFS has been chosen because it is the most commonly used file system. The first section will discuss the design and architecture of NFS. Amoeba has been chosen because it supports high availability, parallelism and high performance. The Details of Amoeba will be presented in the second section. The third section will present some of the features of the AFS. The AFS has been chosen because it has a lot of similarities with NFS but it has certain features like volume replication which is not implemented in NFS. In the last section, the Virtual File System (VFS) will be discussed.

3.1 The Sun Network File System (NFS)

NFS is designed to give users high performance, transparent access to server file systems on global networks. NFS provides transparent access to remote files for client programs running on Unix and other systems as explained in [Zas99] [Sun95]. The NFS service is stateless and most of the operations of the file access protocol are repeatable. Some of the most important design principles of this file system are listed below:

Transparent Access Users and applications can access remote files as if they were local. They are not required to know whether the files reside on the local disk or on remote servers.

Statelessness NFS operates in a stateless manner using remote procedure calls (RPC) built on top of an external data representation (XDR) protocol. The RPC protocol enables version and authentication parameters to be exchanged for security over the network.

Portability NFS is machine and operating system independent. This allows it to be ported easily to multiple OS and hardware platforms from PCs to mainframes.

Recovery from failure NFS is designed to recover quickly from system failures and network problems, causing minimal disruption of service to users.

Network Protocol Independence NFS has the flexibility to run on multiple transport protocols instead of being restricted just to one.

Performance NFS is designed for high performance so that users can access remote files as quickly as they can access local files.

Security The NFS architecture enables the utilization of multiple security mechanisms.

The NFS clients and server modules communicate using remote procedure calls [SG94]. NFS employs a client/server architecture and therefore consists of a client program, a server program and a protocol that is used to communicate between the two, over the network. See figure 3.1.

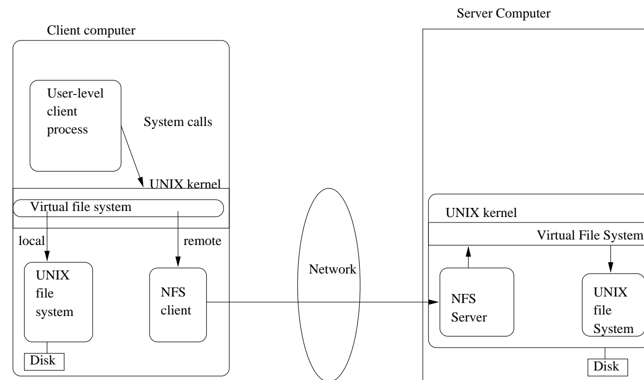


Figure 3.1: The architecture of NFS

The role of the NFS server is to allow its disk file systems to be accessed [1] by other machines [2] on the network.

[1] in other words "shared"
[2] clients

NFS clients gain access to server files by mounting the servers exported file systems. The mount process results in integrating the remote file system into the client's file system tree. An enhanced client side service called the auto-mounter, automatically and transparently mounts and unmounts file systems as needed. The auto-mounter gives clients transparent access to server file systems, whether those servers are connected to clients over LANs and/or WANs. The auto-mounter does not, distinguish between different physical network technologies as stated in [CD01].

The NFS protocol consists of a set of remote procedures enabling clients to manipulate remote files and directories on server systems as if they were local. The routine in this protocol enables clients to send requests for file operations to servers, and servers respond by attempting to perform the operation [3] and then sends back successful results or error values.

NFS reduces data latency by implementing client disk caching. By caching significant amounts of file data on the local disk, clients can drastically reduce the amount of time they would spend waiting for data to be transferred from the remote server. Caching significantly increases the performance of the NFS client and also enhances server scalability. Clients get faster access to files by storing large chunks of data in a local fast access cache. If the file is being read sequentially, the NFS client can anticipate future data requirements through a process called "read- ahead" and can store this information in its local cache for future reference as explained in [Sun96].

Cache Consistency and File Integrity is achieved in NFS by applying File locking. NFS includes a file locking feature called the lock manager. The lock manager and another process called the status monitor work together to guarantee that, multiple readers and writers do not collide with each other. This is accomplished using file locking and access control mechanisms. To minimize data inconsistencies, NFS uses a timestamp-based method to validate cached blocks.

Replication is not supported by NFS. The Sun Network Information Service is a separate service available for use with NFS that supports the replication of simple databases [CD01]. The primary copy approach is used for replication. NFS currently supports replication of read-only file systems [Sun96]. A map entry for a read-only file system may describe several locations for alternative replica servers. At mount time, the auto-mounter mounts the file system on the server that has the nearest proximity to the client. This reduces network routing delays and helps keep NFS traffic off the corporate backbone.

3.2 The Amoeba File system

Amoeba was designed and developed under the direction of Andrew Tannenbaum. Amoeba combines high availability, parallelism, scalability and high performance.

[3] provided the user has proper permission

Amoeba connects the multiple Amoeba systems at different sites into a single coherent system. This goal is obtained by using the object and capabilities in a uniform way [VRsTK90].

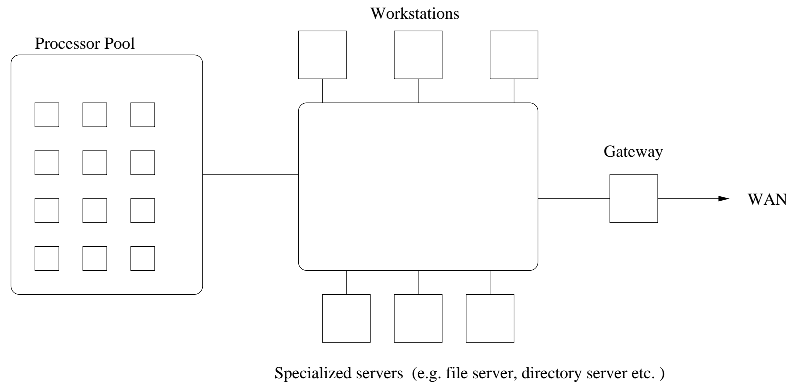


Figure 3.2: The Amoeba architecture

The architecture of Amoeba consists of : The workstations, one per user which can be used for editing and other interactive tasks. Figure 3.2 illustrates the architecture of Amoeba. There is a processor Pool and a group of CPU present which can be dynamically allocated as desired. There are specialized servers like the directory, file, block, database, boot servers etc. Gateways are used to connect Amoeba with the other Amoeba systems located at different sites as stated in [VRsTK90]. All Amoeba machines run the same kernel.

The communication process uses client-server model. The conceptual model for Amoeba communication is the abstract data type of object model, in which clients perform operations independent of their location. Amoeba uses a minimal remote procedure call for communication between client and server. When a message is sent to the server, the client is blocked until the result from server has arrived. GET_REQUEST and PUT_REPLY are the basic primitives used in this case. These are not embedded in the language of the environment, but they are implemented as a small library routine. The detection and retransmission of lost messages, acknowledgment processing, message to packet and packet to message management, are done transparently by the kernel. The messages are unbuffered.

The kernel has three basic system calls for users processes: DO_OPERATION, GET_REQUEST, and SEND_REPLY. These primitives are too primitive for most application programmers. More user oriented interface have been built on top of these mechanism as stated in [VRVST88].

Amoeba uses a single, uniform mechanism for naming and protection. That mechanism is SPARSE CAPABILITIES [VRsTK90]. This mechanism bases on the object description than the process description. The system supports objects such as: directories, files, disk blocks, process, bank accounts, and devices. Small

objects such as integer are not supported. Each object is managed and owned by some services. This naming mechanism can be described as follows: A process request to create an object. The object is created and the capability is given to the process issuing the request. The process can carry out operations on the object by using this capability such as reading or writing a block of a file [VRVST88].

The number and type of the operations of an object are determined by the service which created the object. A bit map in the capability represent the permission of the process. The whole of Amoeba is based on the conceptual model of abstract data types managed by services [VRsTK90]. From the users point of view, the Amoeba environment is regarded as a collection of objects, named by capability on which they can perform operations. Therefore Amoeba is a contrast to the other systems which are process oriented.

An object is a piece of data on which well defined operations can be performed by authorized user, independent of the user's and object's location. Objects are managed by server processes as described in [DKOT91].

The process management allows remote process creation, debugging, check pointing and migration. Each object has a globally unique name contained in its capabilities. The capability is managed entirely by user process not by any kernel. The capabilities are protected cryptographically. The capability has enough redundancy and cryptographic protection to avoid the user guessing the capability. [VRsTK90]

A capability consists of 4 fields:

- The service port: A sparse address of the service that owns the object
- The object number: An internal identifier used by the service to tell which of its object it is.
- The right field: Contains the operations which are permitted on objects.
- The check field: A large random number used to authenticate the capability.

The protection scheme of each server depends on the server. Usually each one builds a capability containing its port, the object number, the rights and a known constant. Thus when a server is asked to create an object it picks an available slot in its internal table, puts the information about the new object and picks a new random number. The rights and check field are mixed by encrypting them with the random number as key, which is then saved in the internal process.

This capability is sent to the server when a process carries out an operation on the object. The server will check the right by using the object number to find the relevant internal table entry and get the random number. This number is used to decrypt the check and right field. If the decryption process produces the correct constant,then the right field can be used to check the operation permitted on the object [VRsTK90].

These capabilities can be stored in directories managed by the directory service. It is a set of (ASCII string, capability) pairs. The most common operation is for the user to present the directory server with a capability for a directory and an ASCII string. Then, ask for the capability that corresponds to that string in the given directory. Other operations involve the entering and deleting of these pairs.

In general, port numbers which are 48 bits must be kept secret, because they are used as a key for accessing the servers. Amoeba has two level of protection:

- Ports for protecting access to servers
- Capabilities for protecting access to individual objects.

Amoeba provides a boot service with which servers can register. The boot service contacts each registered service at intervals. If the server does not reply properly, the boot service determines whether the server is broken, and requests the process server to start up a new process of the server on one of the pool processors [VRsTK90].

Each remote procedure in Amoeba is completely self contained and does not depend on any previous setup. It does not depend on any volatile information in the servers memory. In the case where a server crashes before sending the reply, the kernel on client will time out and try again. When the new server comes up the client's kernel will find the new server and send the request, this process can be carried out without the client knowing something has happened. [DKOT91]

Some services which had been implemented in Amoeba are the block, file and directory service. The most simple one is a server that provides a file server functionally equivalent to the UNIX system call interface. To run most UNIX programs on Amoeba, the program is re-linked with a special library. Other key services are; directory service, bank service and boot service as described in [VRVST88].

Amoeba has a unique file system. The file system is split-ted into two parts. First is the bullets service for storing immutable files contiguously on the disk. The second is the directory service which gives capability symbolic names and handles replication and atomicity and eliminates the need for separate transaction management system as stated in [DKOT91].

The amoeba file system can be represented as a large tree. This service is immutable file service and supports only three principal operations : read file, create file, and delete file. The file cannot be changed after it is created [VRVST88].

3.3 The Andrew File System (AFS)

The Andrew File System is designed to provide an information-sharing mechanism to its users [SG94]. Like NFS, AFS provides transparent access to remote

shared files for UNIX programs running on workstations. Access to AFS files is through the normal UNIX file primitives. AFS was designed to perform well with larger numbers of active users. File transfer between clients and servers is also supported by AFS as stated in [CD01].

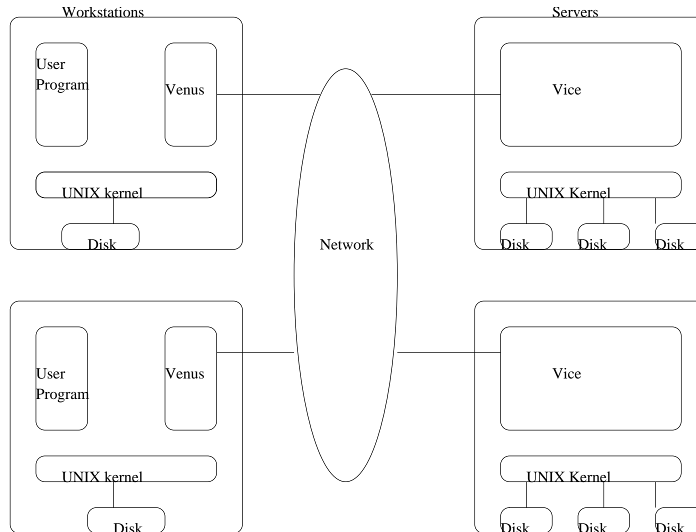


Figure 3.3: The architecture of AFS

AFS is implemented as two software components that exist as UNIX processes, called Vice and Venus. Vice is the name of the server software and Venus is the name of the client software. The diagram on figure 3.3 illustrates the distribution of processes in the Andrew file system.

Scalability It is achieved in AFS by the caching of whole files in client nodes [Zas99].

Global Name Space Each directory and file is identified by the same path name specification, regardless of what machine you are currently logged into.

Cell The administrative domain in which all administrators of the file servers must cooperate on certain configuration decisions and use of privileges. A cell does not necessarily have any geographical limitation [ARSW92].

Volumes The AFS container that keeps a set of related files and directories. This is the smallest unit on which administration is performed (e.g. backup, restoring, moving, quotas etc.) Files in AFS are grouped into volumes to ease relocation and movement [Zay91].

Cache Manager It runs on the local client and caches data requested from the file servers, and if the same data is requested again, it resolves the request from local cache rather than from the file server. The Cache Manager also does pre-fetching from the file server, which could also increase single user performance over NFS [Zay91].

Cache coherence It is maintained in AFS through the Callback Mechanism. Callback Mechanism - When a client caches a file, it is given a "promise" from the file server to notify (call back) the client if any other client modifies that file. When this callback is received, the client then invalidates the cache for that file and subsequent requests for that file will be resolved from the file server, thus getting the changes to the client [SG94].

Volume Replication Volumes which are not highly modified can be replicated on multiple file servers for increased availability and performance. Note that the replicated copies are read-only, thus changes must be made to the master. Then, a manual one line command must be entered to re-copy the entire volume for the changes to be seen in the replicated volumes.

Security AFS uses the Kerberos authentication system to provide reliable identification of the users attempting to operate on the files in its central store. Authentication information is mediated through the use of tickets. A ticket is an object which contains an encrypted version of the user's name and other information [ARSW92].

Protection In addition to standard UNIX rights, AFS has more granularity of rights as well as Access Control Lists (ACLs) which can contain groups and individuals for directories and files.

Inter-operability/Coexistence Due to its VFS style of implementation, the AFS client code may be easily installed in the machine's kernel, and may service file requests without interfering in the operation of any other installed file system.

3.4 The Virtual File Switch Interface

Virtual File System is an interface providing a clearly defined link between the operating system kernel and the different File systems [Bro99] [BB99].

When the Linux kernel has to access a File System, it uses a file-system-type independent interface, which allows the system to carry out operations on a File System without knowing its construction or type [Rub97] [CD01]. Since the kernel is independent of the File System type or construction, it is flexible enough to accommodate future File Systems too [Bro99].

The VFS supplies the application with the system calls for file management [4], maintains internal data structures and passes tasks onto the appropriate actual File System [Bro99]. Another important job of the VFS is, performing standard actions. For example, as a rule, no File System implementation will actually provide an lseek() function, as the functions of lseek() [5] are provided by a standard action of the VFS.

Kernel's representation of the File Systems

The representation or layout of data on floppy disk, hard disk or any other storage media may differ considerably from one implementation of File System to another. But the actual representation of this data in Linux kernel's memory is the same for all File System implementations [Aiv00]. The Linux management structures for the File Systems are similar to the logical structure of a UNIX File System as stated in [Rub97].

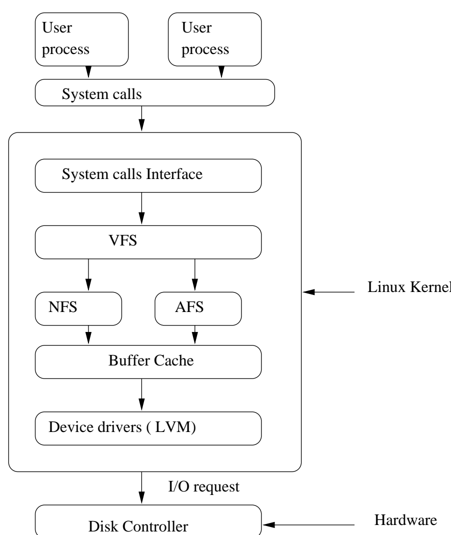


Figure 3.4: The VFS, the file system and the LVM

The VFS calls the file-system-specific functions for various implementations to fill up these structures. These functions are provided by every File System implementation and are made known to the VFS via the function register_filesystem(). This function sets up the file_system_type structure it has passed, in a singly linked list headed by the pointer "file_systems"[BB99]. The file_system_type structure gives information about a specific File System implementation.

In Linux, all files are accessed through the Virtual File Switch [BB99]. The VFS is in the Linux kernel and is used during system calls when accessing files.

[4]like open, read, write etc.

[5]This function sets the file pointer to the position given in the files offset

The VFS is an indirection layer which handles the file oriented system calls and calls necessary functions in the physical file system code to do the I/O.

The VFS defines a set of function that every file system has to implement [Rub97]. This interface is made up of a set of operations associated to three kinds of objects: file systems, inodes and opened files [Bro99]. A mount function is responsible for reading the superblock from the disk, initializing its internal variables, and returning a mounted file system descriptor to the VFS. After the file system is mounted, the VFS functions can use this descriptor to access the physical file system routines [BB99].

The function read_super() forms the mount interface, i.e. it is only via this function that further functions of the File System implementation will be made known to the VFS. This function is also used to initialize the super-blocks in the VFS. After the read_super() functions returns successfully, VFS obtains the reference to the file systems module via call to get_filesystem(fs_type) in fs/super.c:get_sb_bdev() and a reference to the block device.

In order to access a file, the file system containing the file must be mounted onto some mount point in the Linux directory hierarchy [Rub97]. This is done using either the mount system call or the mount_root() function.

A separate superblock structure is maintained for every mounted File System. These structures are held in the static table super_block. The superblock contains information on the entire File System, such as block size, access rights and time of the last modification [BB99].

The superblock provides pointers to functions for accessing the file system, in the function vector s_op. These functions are used to perform all the operations on the File System. The functions in the super_operations structure serve to read and write an individual inode, to write the superblock and to read file system information.

There is also the file_operations structure which contains the methods that can be invoked on files. This is a layer of code which implements a generic file system, where actions and vectors request to the correct specific code to handle the request. Two main types of code modules take advantage of the VFS services, device drivers and file systems.

Two other types of descriptors are used by the VFS: the inode descriptor and the opened file descriptor. Each descriptor contains information related to files in use and a set of operations provided by the physical file system code [SG94]. While the inode descriptor contains pointers to functions that can be used to act on any file(e.g. create, unlink), the file descriptors contains pointer to functions which can only act on open files (e.g. read, write).

Chapter 4

Related works on storage management

Files that are laid out contiguously on the disk can be read and written quickly, in contrast to files whose blocks are scattered across the disk. To achieve optimal file layout, the system requires contiguous free space on the file system. Fragmentation had been a problem in the computer world for a very long time. There are many different implementations of the general concept of logical volume management. One was created by the Open Software Foundation (OSF) and was integrated into many UNIX operating systems like HP/UX etc. This also served as a base for the Linux implementation of Logical Volume Manager (LVM). At present, the emerging solutions are the LVM for Linux, the Network Storage Pool in the Global File System (GFS) and the Volume Manager in the Extended File System (XFS). These three technologies will be discussed and compared in this chapter in detail. The first section presents the design of the GFS, the second section deals with the architecture of XFS and the last section concentrates on LVM.

4.1 The Network Storage Pool in the Global File System

In this section, the important components of the Global File System (GFS) will be discussed. The resource groups, device Locks and the memory Hierarchy will be presented.

The Global File System is a distributed file system based on shared network storage [Aga99][Sol96]. Client file managers exclusively service local file system requests. Network attached storage devices directly serve clients. Each client views storage as locally attached to their machines, though no single computer owns or controls these network attached devices [RSK96]. No direct communication exists between computers; GFS clients remain independent from failures

and bottlenecks of other clients.

GFS achieves client independence by atomically modifying shared data. A storage-device-managed locking mechanism [1] facilitates atomic operations [Aga99]. Before data modification, clients have to acquire locks. After clients modify and write data back to the storage devices, these locks are released.

Storage devices are connected to client computers through switched channel networks called Storage Area Networks (SAN) [Sol96]. GFS logically groups storage devices to provide clients with a unified storage space. This collection of network attached storage devices is a network storage pool(NSP). Sub-pools divide NSPs into groups of similar device types [RSK96].

Figure 4.1 illustrates GFS distributed environment. A shared storage pool may include a variety of devices with differing performance characteristics. A cluster of independent clients are connected to the network storage pool via a SAN. Each client may have multiple network connection to the NSP [Sol96]. The storage facilities are grouped in sub-pools depending on their characteristics.

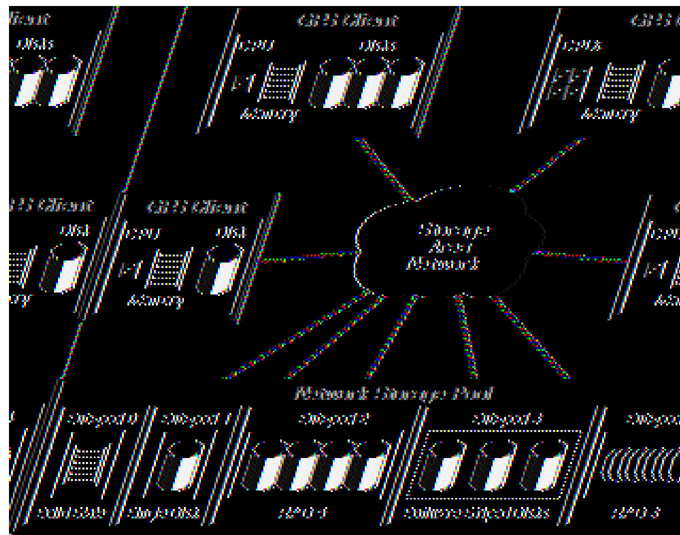


Figure 4.1: The GFS cluster
[Sol96]

GFS provides transparent parallel access to storage devices while maintaining standard UNIX file system semantics: user applications see only a single logical device via the standard open, close, read, write and fcntl [2].

In the GFS, heterogeneous collection of shared storage into a single logical volume called the network Storage Pool is present as described in [Sol96]. The

[1] Also called Dlocks

[2] This function can change the properties of a file that is already open

pool driver [3] is built on top of the SCSI and Fiber Channel drivers [Kee98] and is similar to Linux's multiple disk (md) as stated in [Tei99]. It allows stripping across multiple devices and provides a pool of Dlocks for GFS, thereby hiding the implementation details. New storage pools can be written, assembled and added to the kernel dynamically [Kee98].

Network Storage Pools are collections of physically shared devices. Sub-pools partition NSPs according to the characteristics of the devices [Kee98]. Sub-pools inherit characteristics from underlying devices and network connections. Device characteristics range from low-latency to high-latency bandwidth performance. A sub pool of high-bandwidth devices inherits high-bandwidth characteristics. A high-bandwidth sub pool includes disk arrays or software striped disks connecting to clients with one or more network links. A low-latency sub pool consists of solid state devices.

GFS exploits different sub pool characteristics. GFS places frequently referenced files, like directories, on low-latency sub-pools [RSK96]. Large files benefit from high-bandwidth sub-pools. Single files may be split across sub-pools of different types. Although splitting real data in files is not useful, positioning real data and meta-data on different sub-pools has performance advantages. Real data is placed on high-bandwidth sub-pools and meta-data on low-latency sub-pools, because meta-data requires as little as one percent of the total storage space [Sol96].

4.1.1 Resource Groups

GFS organizes file systems into several resource groups (RG) [Aga99]. Resource groups distribute file system resources across the entire Network Storage Pools; multiple resource groups can exist per device. Resource groups are essentially mini-file systems. Each group possesses an information block, data bitmaps, dinodes, and data blocks. Resource group blocks contain information similar to traditional super-blocks. In normal operation, resource groups are transparent to users [RSK96]. File data and meta-data may span multiple resource groups and sub-pools.

4.1.2 Device Locks

The GFS uses device locks to maintain data coherence. Device locks are mutual exclusion mechanisms managed by storage devices [Aga99]. GFS associates locks with file system data; devices have no knowledge of locked data. SCSI DLOCK commands manipulate device locks as described in [Sol96].

Each device lock has a state bit, activity bit, and a multi-bit clock. At completion of every DLOCK command, devices return the current values of each

[3] Pool driver is a collection of low level functions in the operating system, used to access hardware devices.

bit and clock. Clients save these values for lock activity measurements. Activity measurements are useful for failure recovery, load balancing, shared resources and maintaining data coherence [RSK96].

Device locks support four primary actions: lock, unlock, unlock increment and reset lock. These actions use test- and set-clear operations to modify state bits. Devices increment clocks after successful completion of "unlock incr" and reset lock actions. Reset lock clears a lock, if the current clock value equals an input clock value.

Device locks support three secondary actions: no action, activity on, and activity off. These actions do not modify state bits. The "no action" command returns state bits, activity bits, and clocks. The "activity on" and "activity off" actions set and clear activity bits, respectively. Clocks increment after activity off actions. A set activity bit causes clocks to also increment after successful unlock actions.

4.1.3 Memory Hierarchy

The GFS memory hierarchy includes client memory, storage devices caches, and storage device media. The GFS memory hierarchy is similar to that of a local file system; however, maintaining consistency between GFS levels is more complex. GFS uses a mutual exclusion locking scheme with a write-through approach for coherence [RSK96].

Data consistency within and between all three memory levels of GFS hierarchy is essential. Three different consistency mechanisms maintain coherence. Memory locks ensure mutual exclusion among processes in system memory [Sol96]. Storage devices manage consistency between local caches and media. GFS uses a locking mechanism, called device locks, to maintain consistency between system memories and devices. Storage device controllers manage device locks. Clients must acquire device locks before modifying shared data. The locks are released after writing data back to device storage. Device locks not only ensure mutual exclusion but also provide a facility for client memory caching. This scheme has the simplicity of a centralized mechanism yet it is distributed across several devices. GFS mechanisms guarantee strong consistency; every device read returns the most recently written data. Lets take a look at the design of XFS.

4.2 The Volume Manager in the Extended File System (XFS)

This section will present the components of XFS and their functionalities. The architecture of XFS is quite different from that of GFS but the management of the storage facilities is similar to that of GFS. The Allocation Groups will also be discussed, the storage scalability and performance scalability.

We will describe the architecture and design of the new file system XFS for the Silicon Graphics' IRIX operating system. It is a general purpose file system for use on both workstations and servers [SDH99].

XFS adapts many techniques used in the field of high performance multiprocessor design [AW99]. It organizes hosts into a hierarchical structure so that locality within clusters of workstations can be better exploited. By using an invalidation-based write back cache coherence protocol, XFS minimizes network usage. It exploits the file system naming structure to reduce cache coherence state. XFS also integrates different storage technologies in a uniform manner [Lor00].

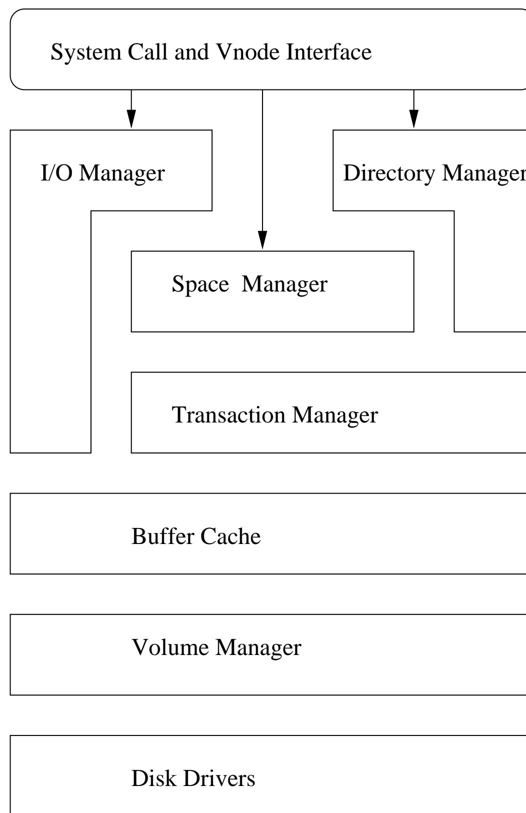


Figure 4.2: The XFS Architecture

The figure 4.2 shows a block diagram of the general structure of the XFS file system.

XFS is modularized into several parts, each of which is responsible for a separate piece of the file system's functionality [AW99]. The central and most important piece of the file system is the space manager, which manages the file system free space, the allocation of inodes and the allocation of space within individual files [SDH99].

The high level structure of XFS is similar to a conventional file system with the addition of a transaction manager and a volume manager. XFS support all of the standard UNIX file interfaces and is entirely POSIX [4] and XPG4 [5] compliant [SDH99]. It sits below the vnode interface in the IRIX kernel and takes full advantage of services provided by the kernel, including the buffer/page cache, the directory name lookup cache, and the dynamic vnode cache, see figure 4.2.

The I/O manager is responsible for satisfying file I/O requests and depends on the space manager for allocating and keeping track of the space for files.

The directory manager implements the XFS file system name space. The buffer cache is used by all of these pieces to cache the contents of frequently accessed blocks from the underlying volume in memory. It is an integrated page and file cache shared by all file systems in the kernel. The transaction manager is used by the other pieces of the file system to make all updates to the meta-data of the file system atomic. This enables the quick recovery of the file system after a crash.

The volume manager used in XFS, known as XLV, provides a layer of abstraction between XFS and its underlying disk devices. XLV provides all of the disk striping, concatenation, and mirroring used by XFS. XFS itself knows nothing of the layout of the devices upon which it is stored. This separation of the disk management from the file system simplifies the file system implementation, its application interfaces, and the management of the file system.

4.2.1 Storage Scalability

XFS concentrates more on efficiently supporting large files, large file systems, large numbers of files, and large directories [SDH99]. The mechanisms used to achieve such scalability will be described below.

4.2.2 Allocation Groups

XFS supports full 64 bit file systems. All of the global counters in the system are 64 bits in length. Block addresses and inode numbers are also 64 bits in length. To avoid requiring all structures in XFS to scale to the 64 bit size of the file system, the file system is partitioned into regions called allocation groups (AGs). Allocation groups keep the size of the XFS data structures in a range where they can operate efficiently without breaking the file system into an unmanageable number of pieces. AGs are typically 0.5 to 4 gigabytes in size. Each AG has its own separate data structures for managing the free space and inodes within its boundaries. Partitioning the file system into AGs limits the size of the individual structures used for tracking free space and inodes [SDH99].

[4] Portable Operating System Interface

[5] X /Open Portability Guide, issue 4

AGs are only used occasionally for disk locality [6]. They are generally far too large to be of much use in this respect. Instead, locality is established around individual files and directories. By making the structures in each AG independent of those in the other Ags, XFS enables free space and inode management operations to proceed in parallel throughout the file system. Thus, processes running concurrently can allocate space in the file system concurrently without interfering with each other.

Space management is the key to good file system performance and scalability. Efficiently allocating, freeing space and keeping the file system from becoming fragmented are essential to good file system performance. XFS has replaced the block oriented bitmaps of other file systems with an extent oriented structure consisting of a pair of B+ trees for each AG. The entries in the B+ trees are descriptors of the free extents in the AG [SDH99]. Each descriptor consists of an AG relative starting block and a length. One of the B+ trees is indexed by the starting block of the free extents, and the other is indexed by the length of the free extents. This double indexing allows for very flexible and efficient searching for free extents based on the type of allocation being performed.

XFS provides a 64 bit, sparse address space for each file. The support for sparse files allows files to have holes in them for which no disk space is allocated. In order to keep the number of entries in the file allocation map small, XFS uses an extent map rather than a block map for each file.

XFS uses a write ahead logging scheme that enables atomic updates of the file system [AW99]. Logging new copies of the modified items makes recovering the XFS log independent of both the size and complexity of the file system. Recovering the data structures from the log requires nothing but replaying the block and inode images in the log out to their real locations in the file system. In the next sub section, we will see how the factors described above influence the performance of the XFS.

4.2.3 Performance Scalability

In addition to managing large amounts of disk space, XFS is designed for high performance file and file system access. XFS is designed to run well over large, stripped disk array where the aggregate bandwidth of the underlying drives ranges in the tens to hundreds of megabytes per second. The key to performance in these arrays are I/O request size and I/O request parallelism. Modern disk drives have much higher bandwidth when requests are made in large chunks. Since there are

[6] Small file performance in most file systems is limited by slowly improving disk access times, even though current file systems improve on-disk locality by allocating related data objects in the same general region. The key insight for why current file systems perform poorly is that locality is insufficient - exploiting disk bandwidth for small data objects requires that they be placed adjacently

practical limits to individual request sizes, it is important to issue many requests in parallel in order to keep all of the drives in a stripes array busy.

The first step in allowing large I/O requests to a file is to allocate the file as continuously as possible. This is because the size of a request to the underlying drives is limited by the range of contiguous blocks in the file being read or written.

XFS uses a combination of clustering, read ahead, write behind, and request parallelism in order to exploit its underlying disk array [AW99]. For high performance I/O, XFS allows applications to use direct I/O to move data directly between application memory and the disk array using DMA [7], which thereby provides them with access to the full bandwidth of the underlying disk array without the complexity of managing raw disk devices [SDH99]. In the next section, we'll examine the LVM and see how it differs from the Volume Manager in XFS and GFS.

4.3 Logical Volume Manager (LVM)

This section will describe the Logical Volume Manager, how it functions and the components that influence it's functionality. Current disks on the PCs can be partitioned with primary partitions and additionally extended partitions. A minimum of one primary partition is required and a maximum of four primary partitions per disk is allowed as explained in [Mau99]. As a method to handle disk partitions more flexible, there is also the option to create one extended partition per disk at the cost of losing one primary partition. The extended partition itself is unusable for file systems. Instead it is used as a container to overcome the limitation of primary partitions: within an extended partition, logical partitions can be created which in turn can be used. The BIOS normally requires that at least one primary partition is present which is tagged active and contains the operating system [8].

A volume manager is a subsystem for online disk storage management which has become increasingly used across UNIX implementations and is a serious enabler for Linux in the Enterprise Computing area. It adds an additional layer between the physical peripherals and the I/O interface in the kernel to present a logical view of disks, unlike current partition schemes where disks are divided into fixed-sizes.

Advanced Logical management tools and software RAID are the specialties of the LVM and Multiple Devices (MD) [Zyn99] drivers respectively. These are the two most widely used Linux Volume Managers today [Mau99].

[7] Direct Memory Access

[8] This is true at least for older operating systems - Linux a clever operating system has a number of boot managers which are able to boot the kernel even from logical partitions

How LVM operates

LVM allows you to handle disk space dynamically instead of statically [Mau99]. The additional abstraction level allows logical groupings of storage to be manipulated independent of the actual logical devices which are being used. The user gains flexibility in allocating, moving, and replacing specific devices.

It allows to handle disk space very flexibly not only at installation time but also any time later. But for this, some new ideas have to be adopted. Basically the BIOS limitation of four primary partitions can't be overcome but in effect this is what happens. Let's take a look at the terms of LVM.

A physical Volume(PV) is the basic building block. It is a primary partition on a disk, usually only one out of the four possibilities. PV is the lowest level in the Linux LVM storage hierarchy [Mau99]. A PV is a single device or partition and is created with the command: pvcreate /dev/hda1. This step initializes a partition for later use.

Multiple already initialized PVs are merged into a Volume Group(VG). This is done with the command: vgcreate testVg /dev/hda1 /dev/hda2. Both hda1 and hda2 must be PVs. One or more (up to 255) physical Volumes form a Volume Group(VG). The VG named testVg now has the capacity of two PVs and more PVs can be added at any time. This steps also registers testVg in the LVM kernel module. Volume Groups are "containers" of disk space. VGs can be imagined as a pool of disk space or as an extensible, huge stock of storage where you don't have to buy all of the goods at once. As long as there is some storage you can get it and if your stock is running low, you can add some more storage by adding disks without the need to repartition or creating new mount points. Up to 99 VGs are possible per machine at the moment.

Figure 4.3 illustrates the storage architecture of LVM. Two physical Volumes have been used to create a Volume Group. Three Logical Volumes have also been created from this VG.

LVs are actual block devices on which file systems can be created. Creating an LV is done by: lvcreate -L1500 -ntestLv testVg. This command creates a 1500 MB linear LV named testLv. The device node is /dev/testVg/testLv. Every LV from testVg will be in the /dev/testVg/ directory [TM99].

Like partitions, Logical Volumes (LV) can handle "raw partitions" like databases [Mau99]. A maximum of 255 LV can be defined per VG. LVs can reside anywhere in a Volume Group, but they can't span the borders of a VG. A VG can be viewed as a large pool of storage from which Logical Volumes(LV) can be allocated. PVs can be splitted further into Physical Extents (PEs), similarly the smallest unit of a Logical Volume is called a Logical Extent.

A file system on testLv may need to be extended. To do this the LV and possibly the underlying VG need to be extended. More space may be added to a VG by adding PV's with the command: vgextend testVg /dev/hda3 where hda3 has been "pvcreated". The device testLv can then be extended by 100 MB with

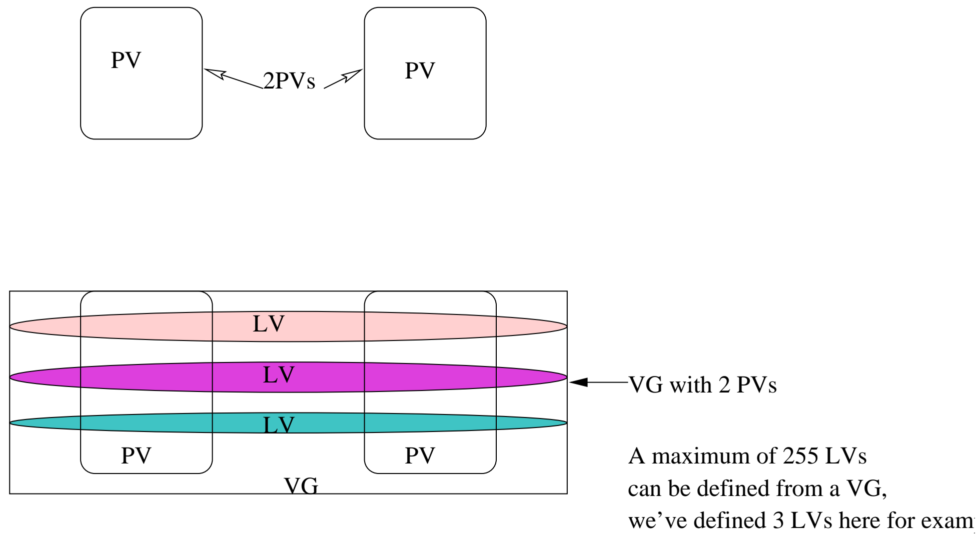


Figure 4.3: The Logical Volume, Volume Groups and Physical volumes

the command: lvextend -L+100 /dev/testVg/testLv if there is 100 MB of space available in testVg. The commands vgreduce and lvreduce can be used to shrink VG's and LV's [TM99].

When shrinking volumes or replacing physical devices, all LEs must sometimes be moved off a specific device [Mau99]. The command pvmove can be used in several ways to move any LEs off a device to available PEs elsewhere. There are also many more commands to rename, remove, split, merge, activate, deactivate and get extend information about current PVs, VGs and LVs [TM99].

There developments underway for a method for "snapshotting" with Linux LVM. This is a way of doing efficient on-line backups. A user command is used to initiate a snapshot at which time the current stable state of the volume is "frozen". The file system and volume can continue to be used but any writes will result in a copy of the original block to a separate volume. The old and new location of each modified chunk is maintained in the LVM. Using pointers to the original data, the snapshotted version can be recovered later [TM99].

Chapter 5

Related works on the Replication of files

Overview

In order to improve the performance and high availability of the file system, it is important to implement replication [Ben00]. Replication is the maintenance of many copies of a file in a system [HHB96]. Performance improvement occurs when access to data from a local storage medium is faster, cheaper, or less subjected to congestion than from a remote store. A suitable replication service must enhance availability in the face of communication outages, as well as system or storage failures [SG94]. In this chapter, replication design alternatives will be discussed, followed by the different update protocols. Section 5.3.1 will describe the replication procedures in the Ficus file system while section 5.3.2 covers the Harp file system.

5.1 Replication Design Alternatives

There exist three different methods of implementing replication. Replication can be implemented manually or explicitly by the systems administrator. There is also the lazy-replication, carried out by the system during low load periods in a transparent way. The last one is based on the group communication, where every write request is sent to all servers at the same time.

There exist different implementations of replication modes in some file systems [Gos94]. Replication systems can be usefully classified along several dimensions: conservative vs. optimistic update, client-server vs. peer-to-peer, and immediate propagation vs. periodic reconciliation [GRR99]. A fundamental question in replicated data systems is how to handle updates to multiple copies of the same data item [HHB96]. If copies cannot communicate instantaneously, then concurrent updates to different replicas of the same data item are not possible.

5.1.1 Conservative vs. optimistic update

Conservative update replication systems prevent all concurrent updates, causing mobile users who store replicas of data items to have their updates rejected frequently, particularly if connectivity is poor or non-existent [HHB96]. Even when connected, mobile users will spend bandwidth to check consistency at every update. This strategy is often used in the wired world. The Harp file system implements this approach. Optimistic replication on the other hand allows any machine storing a replica to perform an update locally, rather than requiring the machine to acquire locks or votes from other replicas. Optimistic replication minimizes the bandwidth and connectivity requirements for performing updates [GRR99]. The Ficus file system uses optimistic updates.

5.1.2 Client-server vs. peer-to-peer

In client-server replication, all updates must be propagated first to the server machine that further propagates them to all clients. For example when updating the cache of the clients in NFS. Peer-to-peer systems allow any replica to propagate updates to any other replica [Ben00] [GRR99]. Client-server systems simplify replication systems by limiting costs thereby imposing a bottleneck at the server. Peer systems can propagate updates faster than Client-server systems by making use of any available connectivity. For example Rumor file system [GRR99]. Client-server replication is a good choice for some mobile systems that disconnect from a central network meanwhile peer replication is a good choice when connectivity patterns of the mobile computers are less predictable [HHB96].

5.1.3 Immediate propagation vs. periodic reconciliation

Updates to data replicas must be propagated to all other replicas. Immediate propagation notifies other replicas of the new state of the data as quickly as possible, when it works. Rumor also implements the Immediate propagation update. Alternatively, updates can be propagated at a later and more convenient time, typically batched. This option of periodic reconciliation does not spread updates as quickly, but allows propagation to occur when it is cheap or convenient [HHB96].

5.1.4 A continuous consistency model

This approaches depends on dynamically trading consistency for availability using a continuous consistency model. In this model, applications specify a maximum distance from strong consistency [1]. Decreasing consistency results in a corresponding increase in overall availability. Thus, a continuous consistency model

[1] where existing optimistic models leave this distance unbounded

exposes a tradeoff between consistency and availability that can be dynamically varied based on changing network and service characteristics [YV01].

5.2 Update Protocols

Maintaining consistency among copies when a replicated file is updated is the major design problem of a replicated file system [CD01]. Since it may be desirable to update all copies of a file immediately, servers must be able to determine, based on version numbers or timestamps, whether copies are up to date before a read operation is performed. If version numbers are used, a server which receives a request to perform a read or write operation on a replicated file can request version numbers from other servers listed in the file suite. The latest version number will be considered the current version of the file.

The problem is how many copies of the file should be updated immediately. There are two types of solutions to this problem. Voting solutions and non voting solutions [HHB96]. Voting solutions are results of negotiations between sites to reach an agreement on control decisions. Non voting solutions are not the result of any negotiations. One site makes the decision.

5.2.1 Primary copy

The primary copy approach is based on the concept of centralization. There is a primary server which is in charge of making all sequencing and synchronization decisions for all transactions [Ben00]. Writes are always done to the primary server [SG94]. Before write is done, it has to be written to stable storage at the primary site. If the primary server crashes, updates to other sites are stopped and the primary server becomes a bottleneck to the system. Reads can be done from any server storing a replica. The system is more fault tolerant for reads and the load is well balanced [CD94].

The diagram in figure 5.1 shows the primary copy model. When a client intends to modify a file called f, the request gets to the primary server which then propagates the updates of file f to the secondary servers. If the client intended to read the data, it could read it directly from one of the secondary servers. The Harp file system uses this approach.

5.2.2 Voting or simple majority

According to the majority voting approach, to the multiple copy update problem, a site on which a transaction updating a given replicated data is to be performed, must obtain the consent of at least a majority of sites storing a copy of this data. If the site does not obtain a majority of votes, the transaction is not allowed to proceed until some later time [Gos94]. The advantages of this method are: It

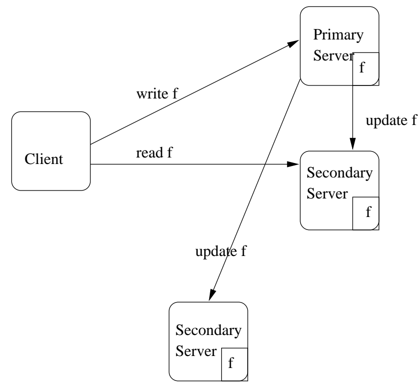


Figure 5.1: The primary copy method

forms the basis of concurrency control to preserve the initial consistency of the data. It also maintains mutual consistency in the system because the majority of data copies are guaranteed to be the same [CD94].

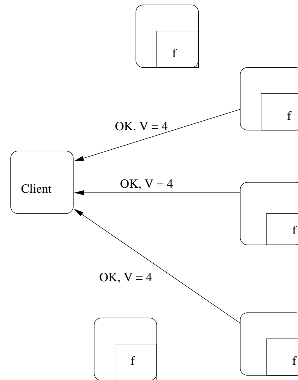


Figure 5.2: The simple majority voting

Figure 5.2 illustrates the simple majority voting protocol. The client intends to modify the file f and from the five sites storing a copy of this data, three of them responded positively and version number of the file was the same. In this case, the client will proceed with it's modification because the majority responded.

5.2.3 Quorum based voting or weighted voting

With this method, each replica of a file is assigned some number of votes. To perform a read operation, a transaction must collect a read quorum of r votes [HHB96]. To perform a write operation, a transaction must gather a write quorum

of w votes. The values r and w must be such that r + w is greater than the total number of votes allocated to that file [Gos94].

Read and write quorums intersect in at least one site. This method, guarantees serial consistency of transactions. It continues to operate correctly with inaccessible copies. It does not insist that a majority of the replicas are updated. The flexibility offered by this method is very important to distributed operating system environment, where different applications have different requirements [CD01].

The diagram in figure 5.3 illustrates the quorum based voting for a read request. If the servers C-L have the last version number, any read quorum will include at least any of these sites. By picking the one with the highest number, the reader knows it is getting the most recent one.

The diagram in figure 5.4 illustrates the quorum based voting for a write request. If the servers D,H,L,I,J,K have the last version number, any write quorum will include at least any of these sites. By picking the one with the highest number, the writer knows it is getting the most recent one.

5.3 Replication strategies in other file systems

There are different approaches to replication. NFS doesn't support replication but the Sun Network Information Service (NIS) is a separate available service for use with NFS, and it supports the primary copy model for replication [CD01]. The caching of file and data in clients computers is another form of replication, like in the Andrew File System (AFS). In this section, we will take a look at the replication strategies in the Ficus Distributed File Systems and the Harp File System.

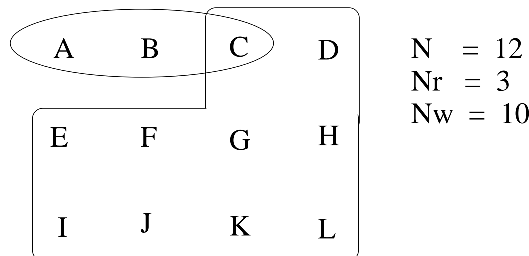


Figure 5.3: The Quorum based voting when reading from a file

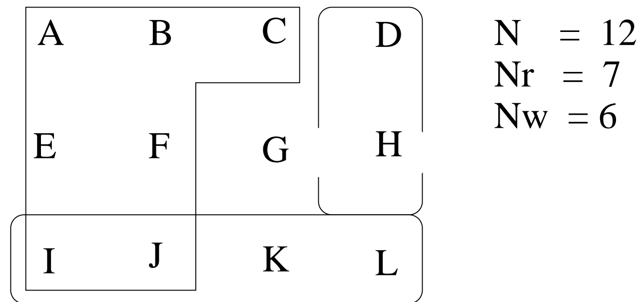


Figure 5.4: The Quorum based voting when writing to a file

5.3.1 Replication in the Ficus Distributed File Systems

Ficus is a replicated general filing environment for UNIX, intended to scale to very large networks. The system employs an optimistic "one copy availability" model [Hei98] in which conflicting updates to the file system's directory information are automatically reconciled, while conflicting file updates are reliably detected and reported [Pag90]. The system architecture is based on stackable layers, which permits a high degree of modularity and extensibility of the file system services.

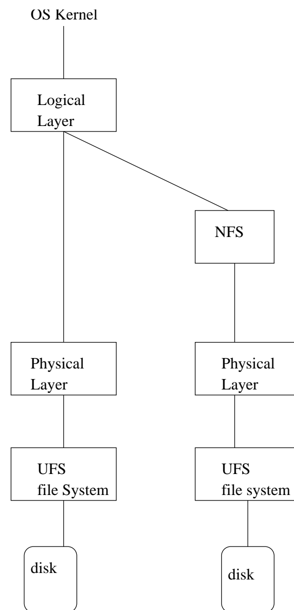


Figure 5.5: Ficus Stack of replication layers

Figure 5.5 shows a file with two replicas, one of which is accessed via NFS. The layered model provides a high degree of flexibility in the Ficus architecture. The logical layer is responsible for replica selection, notifying physical replicas about

the existence of updates, and managing reconciliation. The physical layer on the other hand implements the abstraction of an individual replica of a replicated file. It uses whatever underlying storage service is available [2] to store persistent copies of files.

The primary difficulty with replication services in a distributed environment concerns what to do about updates. The issue is mutual consistency keeping the multiple copies of an object consistent with one another. To solve this problem, the optimistic model is used in Ficus [Hei98]. Let's imagine an environment in which updates are allowed whenever a copy of the needed data is available. When multiple copies are reconnected and if any were out of date, the new version would be automatically propagated to make them current.

Ficus has been designed in such a way that it can function with other file systems. The concept is that of a stack of services, with the interface between all layers having an identical structure, so that one could assemble the services from an available set of building blocks [Hei98]. An important aspect of such an approach is the definition of the interface. It must be one that supports full function and allows high performance both when adjacent layers are in the same address space on a single machine, and when separated by network communication. That's why the stackable layers interface was proposed for Ficus.

Ficus software may be installed at the Virtual File System (VFS) [BB99] layer of a file system, where Sun's Network File System is typically connected [Pag90]. The layered model provides a high degree of flexibility in Ficus architecture. Since the interface to each layer in the system is the standard vnode interface, NFS may be inserted between any pair of layers. When NFS is used above the logical layer, Ficus can be accessible to any type of machine for which an NFS client implementation exists.

Two algorithms are used in Ficus: The first detects update conflicts among data objects, while the second automatically reconciles directory conflicts to produce a correct integrated result. The conflict algorithm associates a version vector with each replica of each object and compares vectors to detect conflicts [Hei98]. The version vector encodes the update history of the replica. Version vectors are used to support concurrent, unsynchronized updates to file replicas managed by non communicating physical layers.

The reconciliation algorithm examines the state of two replicas, determines which operations have been performed on each, selects a set of operations to perform on the local replica which reflect previously unseen activity at the remote replica, and then applies those operations to the local replica. The directory reconciliation algorithm determines which entries have been added to or deleted from the remote replica, and applies appropriate entry insertion or deletion operations to the local replica. The reconciliation algorithm and the basic file update propagation services are both incorporated into the general Ficus file system

[2] such as a UNIX file system

reconciliation protocol.

The next section describes the replication technique in the Harp file system.

5.3.2 Replication in the Harp File System

Now we will take a look at the replication strategy used in the Harp(Highly Available, Reliable, Persistent) File System. Harp provides highly available and reliable storage for files: With very high probability, information in files will not be lost or corrupted and will be accessible when needed, in spite of failures such as node and media crashes [Lis99]. Harp uses a novel variation of the primary copy replication technique that provides good performance because it lays more emphases on network communication rather than improving on disk access. Harp is intended to be used within a file service in a distributed network; for example with NFS or AFS [Lis99]. The idea is that users should continue to use the file service just as they always did. However, the server code of the file service calls Harp [3] and achieves higher reliability and availability as a result. Harp makes calls to the low-level UNIX file system operations.

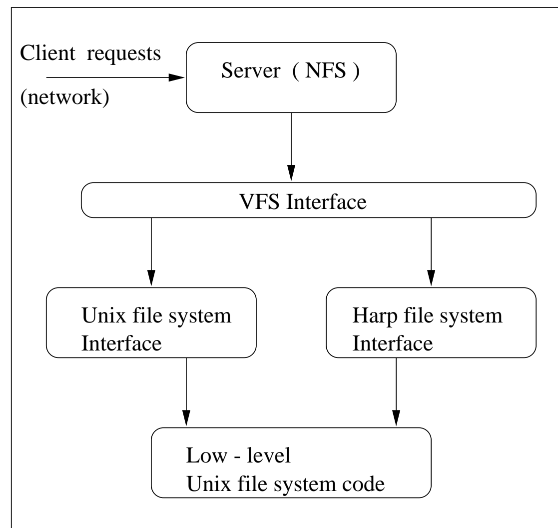


Figure 5.6: Harp system structure

As illustrated in figure 5.6, the Harp code is just a small layer in the overall system. Clients use Harp through NFS.

All modifications to a file are reliably recorded at several server nodes. As described above, with the primary copy replication technique [Gos94], the client's calls are directed to a single primary server, which communicates with other backup servers and waits for them to respond before replying the client. The

[3] via the VFS interface

system masks failures, by performing a failover algorithm in which an inaccessible server is removed from service. When a primary server performs an operation, it must inform enough backups to guarantee that the effects of that operation will survive all subsequent failovers.

Modification operations require a two-phase protocol. In phase 1, the primary server informs the backups about the operation. When the backups acknowledge receipt of this information, the operation can commit.

Harp is one of the first implementations of a primary copy scheme that runs on conventional hardware. Harp achieves good performance by recording the effects of modification operations in a log that resides in volatile memory; operations in the log are applied to the file system in the background [Lis99]. Essentially, it removes disk accesses from the critical path, replacing them with communication (from the primary to the backups), which is substantially faster if the servers are reasonably close together.

In using the log to record recent modifications, Harp is relying on a write-behind strategy, but the strategy is safe because log entries are not lost in failures. Each server is equipped with a small un-interruptible power supply(UPS) that allows it to run for a short while after a power failure occurs. After power failure, the servers uses a few minutes to copy the information in the log to disk. The combination of the volatile log and the UPS is one of the novel features of Harp [Lis99].

Harp also supports the Virtual File System(VFS) interface [BB99]. All operations in Harp are implemented atomically: an operation either completes entirely, or has no effect in spite of concurrency and failures.

When a failure or a recovery from a failure occurs, the group runs a fail-over protocol called view change [Lis99]. The result of a view change is a reorganization within the group, in which a failed node is removed from service, or a recovered node is put back into service.

The performance results of Harp indicate that, it provides equal or better response time and system capacity than an un-replicated implementation of NFS that uses UNIX files directly [Lis99].

In the chapter 7, we will evaluate the different replication techniques discussed in this chapter.

Chapter 6

Related works on file version management

A file may have a sequence of versions which are the results of file modification [CD01]. Each of these versions are written only once and are tentative. This sequence of file versions forms a chronological history of the file [Gos94]. Why is it important to keep a series of file versions in the system? There are two important reasons for keeping file versions: The first is for recovery purposes and the second is to retain an archive for use when a client needs to revert to an earlier state. In this chapter, the first section will discuss file version control in GFS, Amoeba and the shadow pages. The second section covers mutual exclusions and concurrent accesses.

6.1 Related works on file version management

Version numbers are used to document the modifications undergone by a file. With version number management, it is possible to retrieve old copies of a file if need be.

6.1.1 File version control in the Global File System

In the Global File System (GFS) a version number is associated with each lock. Whenever the data associated with a lock is changed, the version number is incremented. Clients may use cached data instead of rereading from disk as long as the version number on the dlock (see chapter 4) is unchanged since the data was last read. The drawback with version numbers is that a client must still read the version number which is located on the dlock storage device or dlock server; this is often a high-latency operation even simple SCSI commands that do not touch the disk often require at least one millisecond [Pre99].

6.1.2 File version control in the Amoeba File Service

One distinctive feature of optimistic concurrency control is that old versions of files corresponding to currently committed transactions are stored by the server [Ben00]. The Amoeba File Service implements optimistic concurrency controlled by a version mechanism. It uses a combination of an optimistic concurrency control mechanism and a locking mechanism to prevent conflicts in simultaneous updates [Gos94]. The basic model is that, a file is a time-ordered sequence of versions, where each version is a snapshot of the file made at a moment determined by a client. The current state of a file is contained in the current version. At any instant, one version of the file is current. Committed versions represent past states of the file, whereas uncommitted versions represent possible future states of the file.

6.1.3 The shadow pages

The shadow pages technique may be used in making new versions of files. Initially the new version of a file can be constructed just by copying the file index of the current version. Each index entry contains a pointer to the block containing a page in the file and a version number. The first time a write operation modifies a page, the service gets a new disk block and writes the new tentative version in it [Ben00]. That means the original page is left unchanged. The tentative version of the page is called a shadow page [CD01]. When the transactions is committed, the tentative version of a file created by the transaction becomes a current version and the file index is updated.

6.2 Mutual Exclusion and Concurrent accesses

Many approaches have been used or proposed to solve the concurrency control problem in transaction processing systems. The most commonly used are locking, optimistic concurrency control and timestamp ordering [Gos94] [CD01].

- Locks are used to order transactions that access the same data items. Data that is about to be accessed by any process must first be locked before any operation can be applied on it. For example, it is preferable to adopt a locking scheme that controls the access to each data item so that there can be several concurrent transactions reading a data item, or a single transaction writing a data item, but not both.

- Optimistic concurrency control allows transactions to proceed until they are ready to commit, where upon a check is made to see whether they have performed conflicting operations on data items.

- Timestamps ordering uses timestamps to order transactions that access the same data items according to their starting times. For example each transaction is assigned a unique timestamp value when it starts. This timestamp is later on used to define its position in the time sequence of transactions.

The Dragon Slayer system was designed to implement the locking mechanism for synchronization purposes as stated in [WKC+94] and other scheduling algorithm have been examined in [Fre99]. Locking facilitates access shared resources. A transaction or atomic action may lock objects to ensure their inaccessibility during a temporarily inconsistent state, that is, to achieve mutual exclusion. This implies that other transactions that attempt to access a locked object will wait for the transaction to end. In Dragon Slayer III all transactions must be well-formed, that means a process must lock an object before accessing it. It does not lock an object which is already locked and before it completes, it must unlock the object it locked.

Read-only transactions in Dragon Slayer III will use locking in order to guarantee that the data being read is not modified by other transactions at the same time. If a process has locked an object for reading, another process can still lock the same object for reading, but if a process intends to lock the object for writing, the lock will not be granted. If a process intends to do modification on an object, it has to lock the object exclusively. In this situation locks will not be granted to processes which intend to read this object. To maintain data consistency in Dragon Slayer III, the single writer and multiple readers policy will be applied. This issue will be a thesis topic in future.

Chapter 7

Concept and Design

This chapter will present the design goals of the File Service. The Problems to be solved have already been outlined in section 1.3. The requirements of the File Service can be divided into two basic groups: the user goals and the system goals. From a users point of view, the system should be convenient to use, reliable, safe and fast. On the other hand DRAGON SLAYER III expects the File Service to provide operations for data manipulation, a high degree of data availability through file replication and relocation of files between nodes.

This chapter will concentrate on the design and specification of the File Service. Section 7.1 will deal with designing the key aspects of the file services. In the section 7.2 discussions will be on stateless service. Section 7.3 will explain the procedures of data modification in DRAGON SLAYER while section 7.4 covers the mechanism used for the maintenance of data consistency. File replication will be explained in section 7.5 and in section 7.6 crash recovery will be discussed. Section 7.7 will describe the components of the DRAGON SLAYER system. File partitioning will be covered in section 7.8 to 7.9 while fragmentation of files will be discuss in section 7.10.

7.1 Designing the File Service

During the design process, considerations were made on the following areas:

Users requirements Fulfilling the users requirements and at the same time respecting the basis architecture of the DRAGON SLAYER system.

Stateless Service From the discussions in chapter 2, we know that all the services in DRAGON SLAYER are stateless. Which strategies will the file service use to maintain information about the open files in the system?

Data Modification procedures Writing to the replicas of a file without implementing a good concurrency control mechanism can lead to data inconsistencies among replicas.

Maintaining data consistency Being aware of the fact that a distributed file service is responsible for operations on files, considerations were being made on the issue of file sharing, avoiding and detecting conflict among processes trying to access the same file thereby making sure that the files accessed by the users are up-to-date.

File replication Due to the distributed nature of Dragon Slayer, the support of file replication will be centered around distributed or dynamic replication services. Which of the replication strategies discussed in chapter 5 will best suit the architecture of Dragon Slayer.

Crash recovery After a system failure, how will the file service continue with it's activities and what happens to the transactions that were interrupted?

7.1.1 User requirements

Functionality The minimal requirement for a distributed file service is to guarantee that the users and client programs are able to access their files locally or remotely without the knowledge of what site is storing the file.

Quality of service The question to be answered here is: What is the added value arising from a distributed service? The first issue to be considered here is performance. To offer a reasonable speed during file access, the file service will always provide the users by default with local copies of their files. In the case where a copy of the file is not stored at the local node, the file service will automatically replicate the file to the users local node or it can direct the request to the next nearest node.

- Another issue I've considered in relation to performance is the case of accessing multimedia data. At present, there are new technologies like the Hierarchical Storage Management (HSM) which provides automated management of data for a multilevel storage hierarchy using predefined user policies which specify service levels for document size, access rates, and performance. With HSM, the Dragon Slayer system can satisfy the need to manage data more granularly using automatic and selective migration, archiving and dynamic retrieval [Ren99]. HSM is being discussed in this work because, Dragon Slayer III aims at being a high performance file system, which should also be in a position of supporting advance hierarchical storage management services. This issue will not be implemented in this work but may be the feature will be added to the design of the file service in future.
- The second issue to be considered is reliability and availability. Due to the fact that the instance of the file service will be running on each

DRAGON SLAYER node, means that even if a node fails, this will not have critical impact on the users who were carrying out different file transactions at the node. The transactions will be interrupted only for a short period of time and the request will be directed to other nodes storing replicas of the files being accessed.

7.2 Stateless service

When designing the file service, I took into consideration the modularity structure of DRAGON SLAYER. We now know that it is only the Command Manager in the DRAGON SLAYER system that is stateful. Having this in mind, I had to consider at what level it will be better for the file service to store information on the open files in the system. If the open file table is stored locally by the service instance at the node, data access rates will become faster. On the other hand, it will create inconsistencies among the other file tables resident on other nodes. A synchronization mechanism will be needed to keep all the local tables consistent. This will be more costly to the system because the communication overhead will increase. I therefore decided to reside the open file table at the Command Manager level. This decision has two advantages. The fact that the Command Manager is stateful unlike the Node Manager, facilitates information storage at the Command Manager. The Command Manager on the other hand is responsible for coordinating the communication among the different nodes in the system. Therefore it becomes much more faster for messages to be sent to other nodes if the open file table needs to be queried. I'll demonstrate how to open a file as an example to describe the algorithm used by all the file operations and also demonstrates the interaction between the file service, directory service and resource manager.

Algorithm to open a file

The client program issues an open command. The sequence diagram in figure 7.1 illustrates the open method. The Command Manager receives the command line from the client and forwards it to the open command. When the open command receives the command line, it parses the command line in order to set the values of the parameters. After the parameters have been set, a lookup request is prepared and sent out via broadcast to the nodes. When this request arrives the nodes, the Node Manager agent invokes an instance of the directory service. The directory service returns the vnode of the file which contains all information about the file if it exists.

After a locate request has been issued, replies are expected from all the active nodes. When the locate replies have arrived, they are evaluated to know how many nodes are alive in the system at that moment. These replies are also

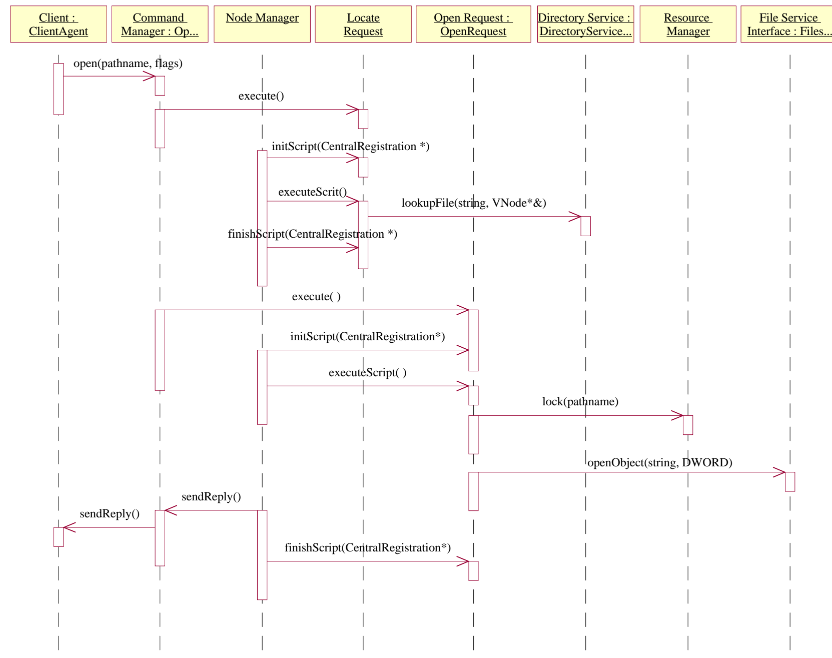


Figure 7.1: Sequence diagram for the open command

evaluated to know on which node a replica exist. The majority voting rule is applied when waiting for the replies and it holds, if more than 50 % of the nodes replied unanimously.

The Command Manager waits for the replies coming from the nodes. When the majority of the replies have arrived, these replies are evaluated in order to find out which of the nodes stores a copy of the file, or if the file exists at all. If the file does not exist, the Command Manager sends a message to the client and stops the transaction. But if the file exists, the flag is evaluated to find out if it is set to read only or write only.

If the flag is set to read only, the addresses of the nodes which sent in replies with the address of the client are compared. The local nodes' address will be used for subsequent requests. A read lock is requested from the Resource Manager at the node. If the lock is granted, send the open request to that particular node. Read request are sent to the local nodes by default if they store a replica of the file.

If the flag is set to write only, get all the addresses of the nodes that replied will be compiled. A multi-cast is sent to the resource managers at those node requesting for a write lock on the files. If the write locks have been granted by all the resource managers, the open request is generated and sent out to the nodes which have locked the files.

When the open request arrives the node, the Node Manager agent invokes an

instance of the file service. The file is opened and the file descriptor is sent to the Command Manager with the open reply. When the open reply arrives the Command Manager, the file record of the opened file are put together and with the file descriptor returned from the system, these information will be inserted into the open file table map.

If the process is successful, a message is sent to the client with the file descriptor. If the reply arrives the client, it gets the file descriptor and uses it in the read and the write commands.

7.3 Data modification in Dragon Slayer

As mentioned earlier, the concurrency control policy that is supported by Dragon Slayer III is single writer and multiple readers policy. In future Dragon Slayer intends to implement a locking mechanism to be used for controlling concurrent data access.

The File Service will use the shadow copy approach to implement file operations that modify data objects in the system. This ensures that, a transaction is either completed or it must be returned to it's initial state. The shadow copy approach guarantees mutual consistency of replicated data. Data modification takes place on a shadow copy, after a transaction completes successfully and has been committed, then the shadow copy will be stored as the current copy. When a file has been modified, after committing the transaction, the most currently committed version of the file becomes the current version.

If a file has been opened successfully for writing and a file descriptor has been returned to the client, the client uses it to complete the write command before sending it to the Command Manager. When the Command Manager receives the command line from the client, it is forwarded to the write command. The write request is prepared and sent out via multicast to those nodes having replicas.

When the write request has been processed at the nodes, a reply is sent to the Command Manager. The Command Manager evaluates the replies. If all the transaction were successful at all the nodes, the Command Manager will send a commit request.

When the commit request reaches the nodes, the shadow copy on which the data was written to, will be turned into the current copy and the version number will be incremented.

If the write transaction is a failure because of one reason or the other, the Command Manager will send a rollback request to all the nodes involved to destroy the shadow copies. After the shadow copies have been destroyed, the write request will be repeated.

What happens if a node fails during a write transaction

Inconsistencies often occur when data modification on some replicas is successful and on others not. As mentioned earlier, a write transaction can only proceed in Dragon Slayer when more than 50 % of the nodes with replicas are present. Consider this situation, where an open request has been sent to the majority of nodes with replica. The Command Manager sends a write request to the nodes which have already opened the replicas for writing. If during the write transaction, one or two nodes crashes, which were already processing the write request, then the following steps will be taken to ensure that after a write transaction the replicas are mutually consistent.

- Let us assume that four nodes are involve in the write transaction. If during the transaction, one node fails and the three nodes leftover write successfully to the replicas. In this situation, the write transaction will be committed on the three nodes that wrote successfully and an update will be carried out later with the node that failed.
- Let's take a look at another scenario. If during the transaction, two nodes fail and two other nodes writes successfully to the replicas. In this situation, we don't have any majority that has completed the write transaction. Half of the nodes did not complete the transaction and the other half completed. In this case, the majority voting rule will not be respected if the transaction is committed. The solution in this case, will be to stop the transaction, register it as a failure and issue a rollback in order to delete the shadow copies at the nodes. That means a write request will be sent by multicast to all the nodes concern. The communication overhead when sending a multicast to all the four nodes involves is relatively lower than the communication overhead when carrying out an update with two or more nodes that failed.

I came to the conclusion that, if more than one node fails during a write transaction, it is better to repeat the write request, because the communication overhead will be relatively lower than if we had to send updates to many nodes at once.

7.4 Version consistency mechanism for the maintenance of data consistency

Ideas that are used in the design of concurrency control mechanisms have been discussed in section 6.2. In this section, we will concentrate on the approach to be used when maintaining data consistency among replicas. The Dragon Slayer system has been designed to implement strong data consistency. The Resource Manager will be responsible for implementing a protocol that will guarantee mutual consistency among data in the Dragon Slayer system. The File

Service on the other hand will implement version consistency to guarantee mutual consistency among the replicas of a file.

As mentioned in section 7.3, the File Service uses the shadow copy approach to implement file operations that modify data objects in the system, in order to ensures that a transaction is either completed or it must be returned to it's initial state.

Data modification takes place on a shadow copy and after a commit has been issued, the shadow copy will be stored as the current copy. The previous current version is added to the sequence of old versions. In order to avoid file version conflicts, the File Service has to look for the best method of managing the file versions.

How should the old file versions be stored in order to make them accessible to the users when needed? How many old versions should be kept consecutively and for how long should they be stored in the system? To make sure that the old versions are always accessible, they should be stored on stable storage. A logical volume will be created and used for storing these files. Well, these old versions cannot be stored infinitely, for that reason they could be deleted after three months in order to create space for the most recent ones. It might be reasonable to keep five sequences of a file's version.

The tentative, current and old versions of a file can all be assigned unique version numbers, because each node that stores a file is responsible for creating all of its versions and numbering them as they are being created. Dragon Slayer III still intends to implement a locking mechanism, the file version control will be coupled with the locks. A file version counter will be implemented alongside the locks. If a file is granted a read lock, the version counter will not change it's state, because modification does not take place during reading. On the other hand if a file is locked for writing, the file version counter will be incremented as soon as the write operation has terminated before the lock is released.

Two versions of a file having the same update history should have the same content, whereas two versions of a file having the same content do not necessarily share the same update history. The decision to use version control in detecting inconsistent data rather than the content of the data is based on the fact that it is cheaper to compare version numbers of a file than the content of the file. When the version numbers have been compared, if there are any differences within the version numbers of the replicas, the current version number will be detected and the outdated replicas will be updated.

7.5 File replication

In chapter 5 we've seen the different methods used in creating and updating the replicas of a file in a system. In Dragon Slayer III a file is created only at the local node where the client issued the create command. That means

there is only one copy of the file system wide. Data is replicated at multiple network nodes to improve on the system's performance, high availability and fault tolerance. Storing many replicas in the system leads to faster data access rates, thereby enhancing the performance of the system. With replicas stored on failure-independent nodes, the client will always access a replica on another node should the default node fail or become unreachable.

7.5.1 Replication On Demand algorithm

After considering the different replication strategies, I came to the conclusion that Replication On Demand will be a good strategy for the DRAGON SLAYER system. How does it function? Access to files stored in the system will be documented in a log file. From the various requests issued on a particular file, one could estimate the user's demand and react to it by replicating the files that are highly demanded. By so doing, we avoid the creation of replicas of a file, which is rarely demanded. With demand replication, files that are highly demanded will be replicated more often than files that are rarely demanded.

To satisfy the goals of high availability and fault tolerance, a minimum number of replicas will be created from those files which are not highly demanded. This is to make sure that if a node fails, the file can still be accessed on another node.

As was discussed in section 5.1 most file systems use either the optimistic or the pessimistic approach in maintaining data consistency among replicas and in propagating updates. From the different replication strategies discussed in section 5.1 we've seen that optimistic consistency models can dramatically increase availability, an unbounded rate of conflicting updates can quickly leave the system in a delusional state.

Taking into consideration the DRAGON SLAYER architecture, I'll prefer to use a pessimistic approach, because the file system was designed to use a locking mechanism as stated in [WKBC89]. Concurrent data access will be achieved in DRAGON SLAYER III by implementing the lock mechanism. Write operations on data can only be executed by one process at a time. Many processes can access the same data for reading but only one process can access data for modification purposes.

By employing a pessimistic approach in DRAGON SLAYER III, inconsistencies are prevented by restricting data updates which could lead to conflicts. Optimistic approaches on the other hand, take the view that it is expensive in terms of performance to obtain locks and unacceptably costly in terms of availability to restrict updates, particularly when conflicts are rare anyway.

What approach should be used in DRAGON SLAYER III to update replicas at nodes which were not accessible when the other replicas were being modified?

The DRAGON SLAYER system has a heartbeat, whose function is to monitor the status of the nodes and keep information about all the nodes in the system. To acquire information about nodes that are alive or dead, the heartbeat is always

in the position of supplying up-to-date information as described in the thesis of [Sie98]. After retrieving the information from the heartbeat service, the method used for reconciliation will be called in order to update the file at the nodes having an out-of-date replica. The reconciliation procedures will be discussed in the next sub section.

To replicate a file and to transfer a file between nodes in the system, leads to the fact that data has to be copied from one node to another. The implementation of data transfer and replication will be based on the interprocess communication in UNIX. The Command Manager will send a Macro-Request to the source node and the target node to create a socket for use in communicating with the other node.

7.5.2 Algorithm for reconciling the replicas of a file

- When the file service instance at the nodes receives an open request, a lookup request is issued to the directory service at the nodes.

- The information concerning the replication grade and the version number is extracted. These two values are sent to the Command Manager, which will compare all of them in order to detect any inconsistency among the replicas. For data modification to take place in Dragon Slayer III, the voting approach is used among the active nodes storing replicas of the file. The majority of the nodes must have the same file version number and the replication grade must also be the same. After the actual version of the file has been agreed on, the rest of the nodes having older versions must be updated before a request can be processed.

- Depending on the replication grade and the version number of the replicas, it is easy to detect the changes in the other replicas by comparing the version numbers. If all the replicas have the same version number and the replication grade is the same then the File Service will proceed with the read or write transactions.

- If the version numbers and the replication grades differ, then there are inconsistencies among the replicas. The replicas having the same version number and replication grade will be sorted out.

- If the majority of replicas did have the same version and replication grade, then the rest of the replicas will be identified as out-dated.

- When the File Service has detected that modifications did take place on a replicated file, it will choose one of the nodes that responded to the broadcast message for synchronization. In this case, an update request will be issued for one of the replicas with the current information to be copied to

the node having an out-dated replica. During this synchronization process, data is copied between the two nodes via unicast.

- When all the replicas have been reconciled, the File Service will proceed with the read or write request. This ensures that the process will get an up-to-date file for reading or writing. When reconciliation has been completed, all the replicas will be granted a write lock if a write request had been issued and a multicast will be sent to the nodes which responded positively to the locate request.

7.5.3 Replication control

The file name and replication grade are fixed at creation time. When a file is created, it's replication grade is one and the version number isone. A file will always be created at the local node by default. The degree of replication will be determined by the growth of the number of nodes in the system.

The distribution policy to be used in DRAGON SLAYER will be based on randomly distributing replicas. Random distribution of replicas throughout the system can sharply decrease the latency between a client and the nodes.

Replica selection primarily occurs when opening a file. A lookup operation is issued by the File Service to the Directory Service. If a client issues a request, any replica can be used to serve the client's request because all replicas are equal. During replica selection, the locally stored replica will be chosen first to serve the client if it exists. If there is no local replica present and the maximum replication grade has been reached, the replica nearest to the client's node will be used.

Proper dynamic selection of replicas will require a characterization of the load on at a node as well as a measurement of the available bandwidth to each node in the system.

To summarize this section, replication control includes determination of the degree of replication and of the placement of replicas.

7.6 Crash recovery

Because of the statelessness of the file service, if the system crashes and recovers, the transactions that were interrupted will be restarted. For the read file operations, there will be no effect but for the write file operation, the changes carried out in the shadow copy will be discarded and the request will be repeated. An update scenario will also take place to ensure that the replicas are mutually consistent.

7.7 Components of the Dragon Slayer system

The architecture of the Dragon Slayer III has already been discussed in chapter 2. In this section, I will concentrate on the components of the system which the File Service has to implement. A client program will also be implemented, which will act as the only interface for the clients to forward their commands to the system.

As earlier said, clients will be able to access and manipulate data in the Dragon Slayer file system through the file operations offered by the File Service. Looking at the structure and components of Dragon Slayer, one can easily realize that the Command Manager is the interface offered by the Dragon Slayer system to the outside world. That means that a fraction of the file operations will reside at the Command Manager and the implementation will reside at the nodes. All the same, there is still a need for a client program which will act as an interface between the clients and the Command Manager.

7.7.1 Client Interface

The client program gets the arguments passed with the users commands to the system. Information maintained at the client contains the process ID, the Host ID and the file handle. This information will be forwarded to the Command Manager with the users commands. The Command Manager will map this entry into the open file table.

The File Service will implement the File Command Application Interface. The File Command Application Interface is the users interface with the system. It gets user's commands and forwards them to the Command Manager. The commands received by the command line interpreter are parsed and later forwarded to the Command Manager for further processing. The figure on page 10 illustrates the client interface and the other components of the Dragon Slayer system.

A client interface for a file service is formed by a set of primitive file operations, such as create, read and write a file. We would adhere to the same file-access semantics with which standard Unix presents to clients, whereby a file must first be opened before a read or write operation can be processed.

7.7.2 Command Manager

The Command Manager is stateful and keeps relevant information about the system, so that after a system failure, it's easier to recover using the information stored at the Command Manager and from the log files. The Command Manager on each node stores information on all the opened objects, their owners and the type of operations to be carried out on the opened objects. This information will be stored in form of the open file table. Each file that is opened in the system,

will be registered in the open file table. Table 7.1 shows the information to be stored in the open file table.

Attribute name	Attribute Contents
Object name	The user's name of the object
object owner	The creator of the object
VnodeID	The systemwide unique ID of each object in DRAGON SLAYER
file descriptor	The non-negative number returned by the open system call
replication grade	The number of replicas present in the system
version number	The version number of the files
number of replicas	Copies present in the system
nodes storing replicas	Nodes having a copy of a file
number of parts	The number of a partitioned file
file operations	The operations to be executed on the objects
subjectID	The userID of the person who issued the command
processID	The process ID
hostID	The host address from where the command was issued

Table 7.1: Entries of the open file table

The File Service fetches the file from disk, stores it in memory and gives the client a connection identifier that is unique to the client and the open file at the Command Manager, referred to here as the file descriptor. The File Service maintains the table of opened files and maps the file descriptors to a file record containing all the attributes of the file including the vnodeID. By so doing, it would become easier to trace if a file has been opened by querying the open file table using the vnodeID of the file.

The File Service implements a list of commands which are managed by the Command Manager and executed by agents. When the Command Manager receives a command through the command line interface, it puts the commands in a queue. The idle agents then remove the commands from the queue and fetch the appropriate service instance from the database for execution. These are the File Service commands which will then be executed by the agents at the Command Manager and a Macro-Request will be generated and broadcasted to all the Node Managers.

When they arrive at the nodes, they will be processed by the Node Manager agents and each node sends a Macro-Reply to the Command Manager. The Macro-Replies received from the nodes will be evaluated by the File Service. After evaluation of these replies, it can decide to terminate the request or to

proceed with it, depending on whether the file exists or not. If a file does not exist, open, copy, delete, transfer and replicate requests will be terminated. But if the file exists, the create request will be terminated.

During the evaluation procedure, the Command Manager stores relevant information relating to the users which issued the request, the host machine from where the command was issued, and the file handle to the command. The Command Manager then forwards the replies to the users after the command has been completely processed. After each close command has been issued and completed, the file entries in the open file table relating to the previous command will be deleted.

7.7.3 Node Manager

The Node Manager is responsible for coordinating all the services locally. The services at the nodes are stateless. That means the request must contain all the information needed, for them to be executed by the agents. During the execution of a command, Macro Requests are sent out to other Dragon Slayer services like the Directory Service and the Resource Manager.

The Macro-Request received from the Command Manager are executed at the nodes by agents. These agents invoke an instance of the file system service after receiving a request directed to the file service. The file service module at the node consist of a collection of system calls and routines used to process the request. When execution of the request is completed, a Macro-Reply will be sent to the Command Manager with the results of the execution. When the client request data, the data will be sent to the Command Manager which forwards it to the client.

7.8 Partitioning of files

7.8.1 Composition and decomposition of a file

Decomposition of a file is the process of breaking up a file into parts. Composition of a file is the process of putting together the different parts of a file to become one file. This section will explain how a file is decomposed and composed.

7.8.2 Motivation

Why should a file be decomposed into parts in the first place? What advantage does the user get from this process? What advantages does it bring to the performance of the system?

To answer the first question, it should be clear that a file can only be decomposed into parts when the user wishes to partition it. Some users might like to

work on smaller units of data rather than on a large file. At times this might occur in order to assist team work, so that each team member can have access to the different parts. Assuming, for example, a user has a file called history.txt and decides to partition the file into three parts, see figure on page 13. Due to the fact that accessing smaller files is faster than accessing larger files, working with parts can improve the performance of the system. File partitioning also promotes concurrent work on the different parts of the file thereby making the system more attractive to the users.

Storage device characteristics that impact file systems design are access times, transfer rates, addressability, cost, capacity and availability. Due to this changes with the concept of parts from Dragon Slayer II to Dragon Slayer III, a part can be split-ted later by the system into a varying number of fragments depending on the storage needs at each node.

Some of these changes have come about because Dragon Slayer III intends to make good use of the new storage technologies like LVM and SAN already mentioned in chapter 4.

7.8.3 Boundary definition

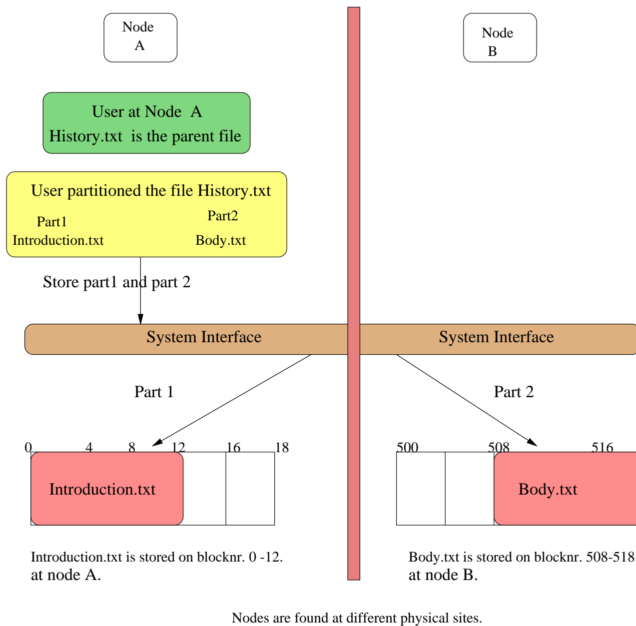


Figure 7.2: The example shows how the system stores parts of a file at different nodes

We know that each node fragments it's files according to it's storage needs.

Therefore a strategy should be developed for dynamically decomposing parts. It is obvious that in most systems, disk space is allocated in block sizes defined by the file system. Depending on the size of the block, a small file might occupy one or two blocks, and a large file might occupy from five to ten blocks or even more.

That means we don't need to know the content of the file, since we are not using characters, rows or spaces to decompose the file. We are only interested in the exact size of the data that will fit in a block. Decomposing a file dynamically makes it very flexible for each node to maximize it's storage facilities.

If history.txt is about to be stored on a disk and there is not enough space to take all the three parts. Take for example figure 7.2, assume that "introduction.txt" and "body.txt" will be stored at node A and B, because node A did not have enough space to hold both parts. Stored along side each part, will be the attributes of the file and information on the location of the other parts.

By decomposing a file dynamically, we start by storing the file on the free blocks until when there is no space left on that particular disk, then we continue on another disk until the whole part has been stored.

Additional information about the parts of the file will also be stored in the vnode. This information will consist of: The disk partition storing the part, block information, information about how many parts are present and where they are located. Having these information in the vnode makes it easier when composing the parts.

7.8.4 Composition of parts

When a locate request has been issued, the Directory Service returns the vnodeID of the parts and the vnode which indicates the nodes hosting that part. The information found in the vnode will be retrieved by the Directory Service and forwarded to the File Service to be stored locally during the time the request is being processed. When the request has been completed, the information will be deleted.

When composing the parts of a file, the File Service gets the relevant information about the file from the vnode. This information consist of the file name, file size, part names, part numbers and pointers to the blocks storing the file. Knowing the sizes of the parts and their storage locations, a system call will be made to read the first part into the users address space, read the second part again and append it to the first part which was already read.

In the case where the parts are stored on different nodes, like in figure 7.2, the first part will be read to a buffer, the data from the second part will also be read and appended to the first part already in the buffer. When the data from the remaining parts has been copied to the buffer, the buffer will be returned to the client. The parts which have already been composed will be deleted and the numbering of the parts will be readjusted.

7.9 Manipulating with parts of a file

Manipulating parts of a file is quiet different from accessing a single file. Parts of a file can be stored at different nodes which makes it difficult to access all of them at once. This section will describe the methods used when working with a partitioned file.

7.9.1 Reading data from the various parts of a file

When a file is being open for reading, the users have to submit as parameters, 0 for a normal file and 1 for a file that is partitioned. If it is a partitioned file, then they also need to give the parts' names or the parts' numbers which they intend to access. From the information submitted by the user, the File Service calls up the Directory Service to locate the parts as earlier described above. If the Directory Service returns the vnodeID of the parts, the vnode structure will be queried to get more information on the pointers to the data blocks. From the vnode table, one can access the disk blocks containing the data content of the file.

Reading data from one part of a file is just the same as reading from a non partitioned file. But reading from part 1 and part 2 at the same time is quite different. Since part 1 can be stored on node A and part 2 could be stored on node B like in figure 7.2. In this case, we will need to process it as two separate read request. The first request will access the data from part 1, return it to the users address space and the second request will access the remaining bytes to be read from part 2. The first request to be issued will be: read(introduction.txt, 0-12) and the second request will be: read(body.txt, 508-518)

If a user requests to read part two and part three of his file, with the entry in the vnode one could tell on which disk blocks the files are stored. Once the block numbers and the pointers to the parts have been retrieved, the file service will locate the starting position as described above, and then call on the I/O function to transfer the data from part 1 and put it in the user's buffer. The procedure is always the same, first of all the File service will call the lseek() function to locate the position in the parts and then the I/O function will be called to transfer the data blocks to the users buffer. This process continues until all the requested number of bytes have been read.

Direct I/O transfers data directly to and from a user-space buffer, rather than making intermediate copies to or from the buffer cache within the kernel.

7.9.2 Writing data to the various parts of a file

When a file is being open for writing, the users have to submit as parameters, 0 for a normal file and 1 for a file that is partitioned. If it is a partitioned file, then they also need to give the name of the part or the number of the part which they

want to access. From the information submitted by the user, the File Service calls up the Directory Service to locate the parts as earlier described.

From the information retrieved from the Directory Service, we know on which blocks the parts and their fragments have been stored. The parts to be modified are read into the users buffer. Many I/O system calls will be needed during the write operation, depending on the rate of data transfer, and the block sizes.

When writing to a part, the whole part must first be read into the users' address space, just like when writing to a non partitioned file. The data is then written to the file and a system call is made to the I/O function to flush the file back to the disk. After writing to a file, if it grows in size and can no longer fit in the old space, the system will relocate it to another disk partition.

7.9.3 Deleting data from a partitioned file

Deleting a file has directly visible effect: The file name disappears from the directory listings as stated in [Ven00]. When a file is about to be deleted, the users have to submit as parameters, 0 for a normal file and 1 for a file that is partitioned. If it is a partitioned file, then they also need to give the parts' names or the parts' numbers which they want to get ride of. From the information submitted by the user, the File Service calls up the Directory Service to locate the parts. The same procedure takes place as described above.

When deleting, two different scenarios can take place. Deleting a part of a file is similar to deleting a non partitioned file. In the first scenario a single part will be deleted. Take the example illustrated in the figure on page 13 and assume that the user intends to delete part 2 of his file "body.txt". If this part is not fragmented, it will be deleted just as a normal file but the numbering of the parts needs to be readjusted after the delete operation has been completed successfully.

In the second scenario, the user may intend to delete all the parts. In this case, the parts will first be composed as described above before they are deleted.

7.10 Manipulating with a fragmented file

At a particular node, a File Service has to deal with three types of file. The normal file, a partitioned file and a fragmented file. A file or a part can be fragmented at a node. To read or write from a fragmented file stored on different disk partitions needs another strategy as that used when accessing the parts of a file. This section will describe the methods used when reading, writing and deleting fragments of a file.

7.10.1 Reading from a fragmented file

As we earlier said, fragmentation of a file takes place locally at a particular node. Lets take for example that a file is about to be stored on the disk partition hda1 and there is not enough space, a fragment of this file will be stored here and the other fragment will be stored on another disk partition hda2. The header of this file will have an entry in it indicating that the file has been fragmented and location of the fragments will also be registered in this header file.

Information about fragmented files, will be stored locally at each node. Table 7.2 contains the information to be stored at the nodes.

Type of Attribute	Attribute Contents
FragmentNum	Number of fragments
FragmentSize	Size of the fragments
DiskName	Name of Disk storing the fragments
Diskpartitions	Which partitions are storing the fragments
StartBlock	The start Block number
BlockSize	The size of the Block

Table 7.2: Information on Fragments

Figure 7.3 shows an example of a fragmented file. The file introduction.txt has been split-ted into two fragments. The first fragment is stored on disk partition hda1 and the second fragment is stored on disk partition hda2.

To read this file, a single request will be issued with the total number of bytes to be read from the file e.g. read(introduction.txt, 25 bytes). The start block number will be retrieved from the fragmentation information stored at the nodes and forwarded to the file system. The number of bytes will be read until the end of the block on hda1. Then it will skip to the start block number of hda2 storing the second fragment to continue reading the remaining number of bytes left.

Actually, what happens here is that the read request will be issued including the total number of bytes to be read and an offset. Reading starts with the first fragment until all the data has been read from hda1 and then with the lseek() [1] function the right position will be determined in the second fragment on hda2. The number of bytes left will be read and returned to the user.

7.10.2 Writing data to a fragmented file

The UNIX operating system defines all files to be simply a stream of bytes [SG94]. Each byte is individually addressable by its offset from the beginning of the file or from the end. The file system automatically packs and unpacks bytes into physical disk blocks as necessary (e.g. 512 bytes per block).

[1] For repositioning within a file

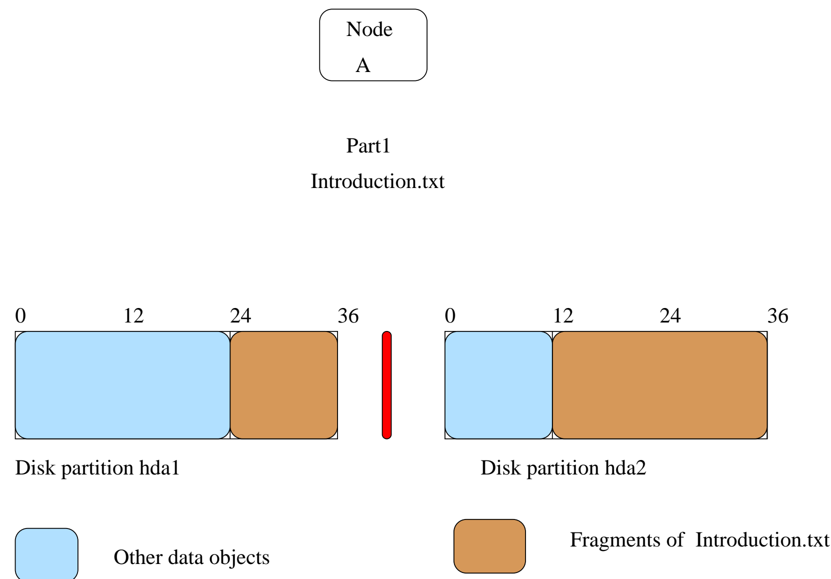


Figure 7.3: This is an example of a fragmented file

When writing to a fragmented file, the blocks storing the fragments are read into the users buffer one after another. The data will be accessed block after block from the offset position till the end of data on each block and transferred to the users address space. Figure 7.3 shows the fragments of the file. When writing to a fragmented file, all the fragments of the file must be read into the users' address space. Data will then be written and after it has been completed, the file will be flushed to the disk. It might happen that the file no longer fits in the old space and that there is storage on another disk partition to store the whole file. Then it will be preferable to store the file at the other disk partition with more storage, thereby avoiding fragmentation.

A write operation appends to the end of the file and advances to the end of the newly written material. A direct-access file allows arbitrary blocks to be read or written. There are no restrictions on the order of reading or writing for a direct-accessed file.

The basic file system needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.

7.10.3 Deleting the fragments of a file

Let's assume the case where the user wants to delete part 3 " conclusion.txt" which has been fragmented due to lack of storage and is stored non contigu-

ously on different disk partitions. From the information retrieved from the vnode structure, we know where the fragments are stored. The first fragment is deleted, through the pointer we access the second fragment and delete it. The fragments are numbered consecutively, after the pointer to the next fragment has been passed to the File service, it gets the number of that fragment and passes it to the delete function. The blocks on which the fragments were stored will become free.

Chapter 8

The File Service Module Implementation

This chapter will give a general description of the implementation procedures. Section 8.1 will give an overview of the implementation tools. In section 8.2, discussions will be on the File Service module and part of the basic architecture of Dragon Slayer. Section 8.3 concentrates on the client interface while section 8.4 explains the file system commands. The parser used by the commands will be dealt with in section 8.5 and in section 8.6 discussions will be on the open file table. In section 8.7, we will present the request and replies used in Dragon Slayer for communication. The file system interface will be discussed in section 8.8 while section 8.9 concentrates on creating files and fragments. Section 8.10 deals with file replication and file transfer. The last section will present the test module.

8.1 Tools used for the implementation

Object oriented analysis and object oriented design was used. Rational [Rat96] rose c++ was used as the modeling tool for object oriented analysis and object oriented design. C++ was used as the programming language. The implementation took place on the Linux and solaris operating system.

The Unified Modeling Language (UML) is now the standard modeling language for object-oriented development. UML [Bur97] essentially defines a number of diagrams that you can draw to describe a system, and what these diagrams mean. UML is a language for specifying, constructing, visualizing,and documenting the artifacts of a software-intensive system.

In the beginning of the project design, use–case diagrams are needed because they describe the external view of the system and its interactions with the outside world. From use–case diagrams we move over to class diagrams. Class diagram, is a central modeling technique that runs through nearly all object oriented meth-

ods. This diagrams describe the types of objects in the system and various kinds of static relationships which exist between them. State transition diagrams are needed after the class diagram for describing the behavior of a single object. The basic idea was to define a machine that has a number of states. The machine receives events from the outside world, and each event can cause the machine to change it's state. Activity diagrams on the other hand are very useful since they support parallel processes and can help one get away from unnecessary sequences. Interaction diagrams have been used in my design because they are very important in making the message structure very explicit. They describe how a group of objects collaborate in some behavior–typically a single use–case.

Wind River's SNiFF+ [Tak99] tool is a source code analysis environment for software developers and teams who work with large amounts of application code. These tool enables development teams to organize and manage code at maximum efficiency. Concurrent Version System (CVS) is a public domain Version and Configuration Management System. It is based on Revision Control System (RCS) repository files, therefore migrating from RCS to CVS and back is simple. Within the SNiFF+ environment, the most useful features are that CVS allows parallel development without the need for branches and it allows access to repositories not seen in the file system.

Rational Rose was used in this project for modeling and to generate the code. By integrating the modeling and development environments using the Unified Modeling Language (UML), Rational Rose enables all team members working on a particular project to develop individually, communicate collaboratively and deliver better software.

8.2 The File Service Modules

In the previous chapters we discussed the requirements and specification of the File Service. We also discussed the role played by the File Service in the Dragon Slayer file system.

The File Service module has been divided into four parts. There is a client module which acts as an interface to the Dragon Slayer system. The class ClientAgent in figure 8.1 represents the client module. The command modules consist of all the commands that a user can forward to the system. This module is resident at the command manager, represented in figure 8.1 by the class FileCommands . The request and reply module consist of requests/replies corresponding to the commands. The requests are generated by the Command Manager agents and the replies are generated by the Node Manager agents. Requests and replies are used by the Command Manager and the Node Manager for communication. The figure on page 14 illustrates the users commands getting to the Command Manager. The Command Manager generates Macro-Request from these commands. When the Macro-Request arrives the node, they are process by

the Node Manager agents. The Node Manager agents send Macro-Replies back to the Command Manager.

The class diagram in figure 8.1 shows part of the basic architecture of the Dragon Slayer system. The class ClientAgent has been used to model the client program. In this class, nine methods have been implemented to be used by the users programs. Through these methods, the user has the possibility of accessing files stored on disk through the Dragon Slayer system.

The class DS3CmdLine forwards the client's command to the Command Manager. This class acts as an intermediary between the client and the command manager. The third level on the diagram illustrates some classes at the Command Manager. There is the class DS3CommandManager which is responsible for the initialization of all the threads and objects as well as managing the command list, which is used to map command strings to objects derived from the class DS3Commands. The class DS3CommandMgrAgent is responsible for the processing of all the tasks at the Command Manager level. The file system commands I've implemented have been derived from the class DS3Command and are represented in figure 8.1 by the class FileCommands. Apart from the class FileCommands, the rest of the classes at the Command Manager level belong to the basic architecture of Dragon Slayer.

This class MacroMessage defines the basic functionality which every communication class in Dragon Slayer must possess. All messages send between components in the Dragon Slayer system must be derived from this class. As you can see from the diagram, the class MacroMessage is the base class to MacroRequest and MacroReply because these two classes have to be transported over the network. The classes MacroRequest and MacroReply are neither resident at the Command Manager nor at the Node Manager. They are used by the Command Manager and the Node Manager for communication purposes. See figure on page 14. The Command Manager agent generates Macro-Request from these commands. These Macro-Request arrives the Node Manager where the final processing takes place. The Node Manager sends the Macro-Replies back to the Command Manager

The class FileRequests which has been derived from the base class MacroRequest represents the request module of the file system, which corresponds to the commands in the class FileCommands.

The class FileReplies which has been derived from the base class MacroReply represents the reply module of the file system, which corresponds to the requests in the class FileRequests.

The next level belongs to the Node Manager. The class NodeManager is responsible for the coordination and management of all transactions taking place locally at the nodes. When the Node Manager receives a request, the request agents represented here by the class RequestAgent gets the request from a queue for processing. The agents obtain objects of the services from the class CentralRegistration needed to process a request. The classes shown in the diagram

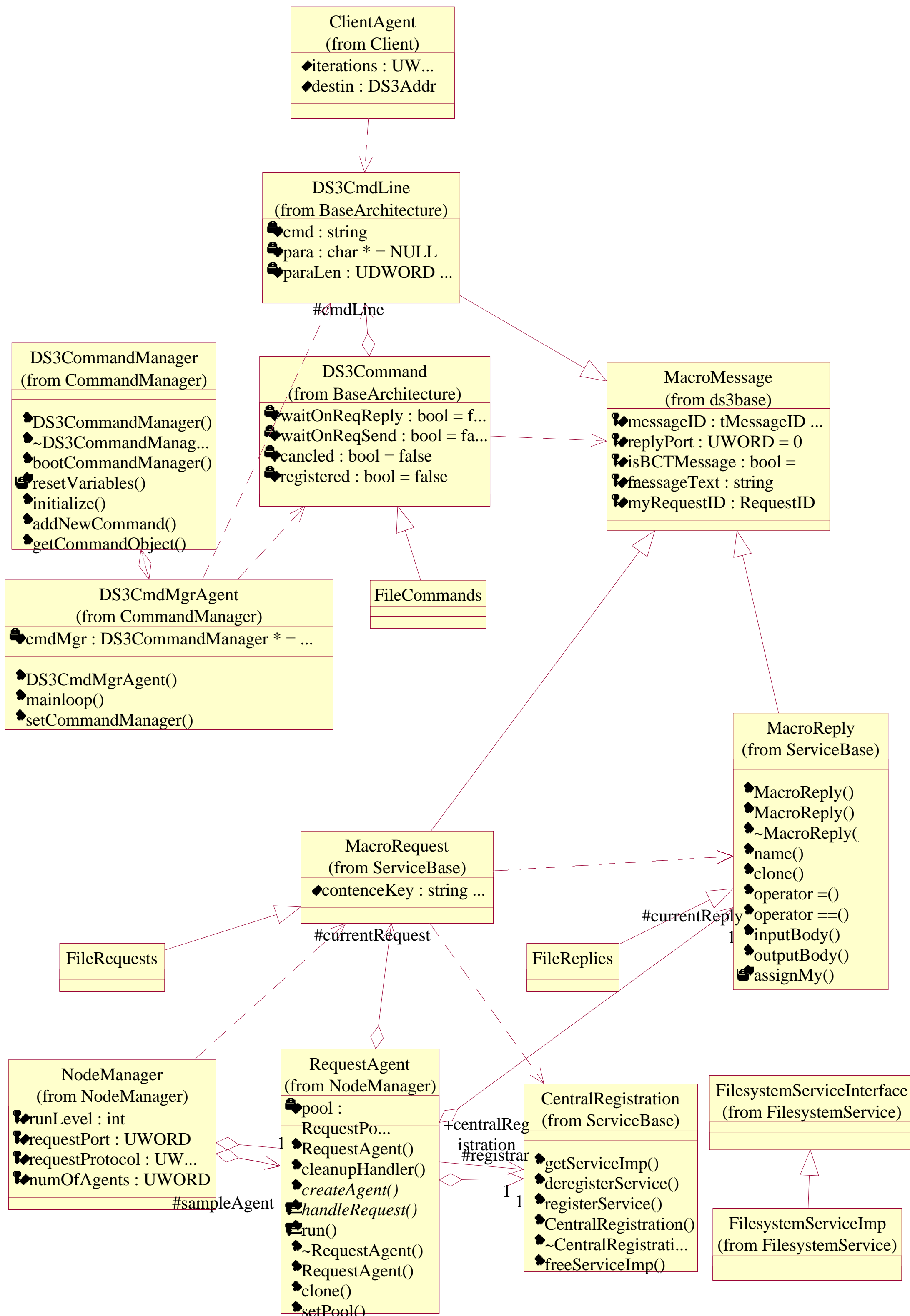


Figure 8.1: Class diagram illustrating the communication between the different modules

8.1 at the Node Manager level also belong to the basic architecture of Dragon Slayer.

The file system service module resides at the Node Manager level and consists of the class FilesystemService and the class FilesystemServiceImp. The final processing of the client's command takes place at the file system implementation. The File Service interface hides the real implementation of the File Service so that any changes undertaken at the implementation will not affect the processes which call these methods. In the coming sections, each of the modules presented here, will be dealt with in detailed..

8.3 Client Program

The commands issued by the client program are forwarded to the Command Manager through the class DS3CmdLine. This class offers methods with which one can set a command line. The class DS3Command is the base class of all commands in Dragon Slayer. That means, all the commands in Dragon Slayer inherit methods and attributes from the class DS3Command.

The class FSTableManager is the base class of all the commands offered by the file system service. This class inherits from the base class DS3Command. This class also depends on three other classes. The class MultipleAnswersContainer which is responsible for collecting all the replies sent from the node managers. It stores the replies in a vector waiting for the commands to collect. The second class StopCondition defines policies which can be used to inform the class MultipleAnswersContainer to abort waiting for a reply. These policy could depend on some time-out limits or on some other parameters.

The class SCFileReplyLocate inherits from the class StopCondition and it is used by the file system commands to know when to stop waiting for replies. The stop condition used at the moment depends on the majority voting rule and a time–out. It states that, if the number of replies received with a positive response are more than 50% of the active nodes in the system at that particular time, then one of the stop condition has been fulfilled. On the other hand, if this stop condition is not fulfilled and there is a time-out, then the second stop condition has been fulfilled. The time-out is used to ensure that the agents do not wait infinitely for the replies.

The client program uses threads to send the command line to the Command Manager. The listing in 8.1 is a fragment of the client's code using the example of the open command for creating a task, sending the task and waiting for the reply from the command manager.

Line 5 to 10 shows the initialization of the variables and the creation of the task.

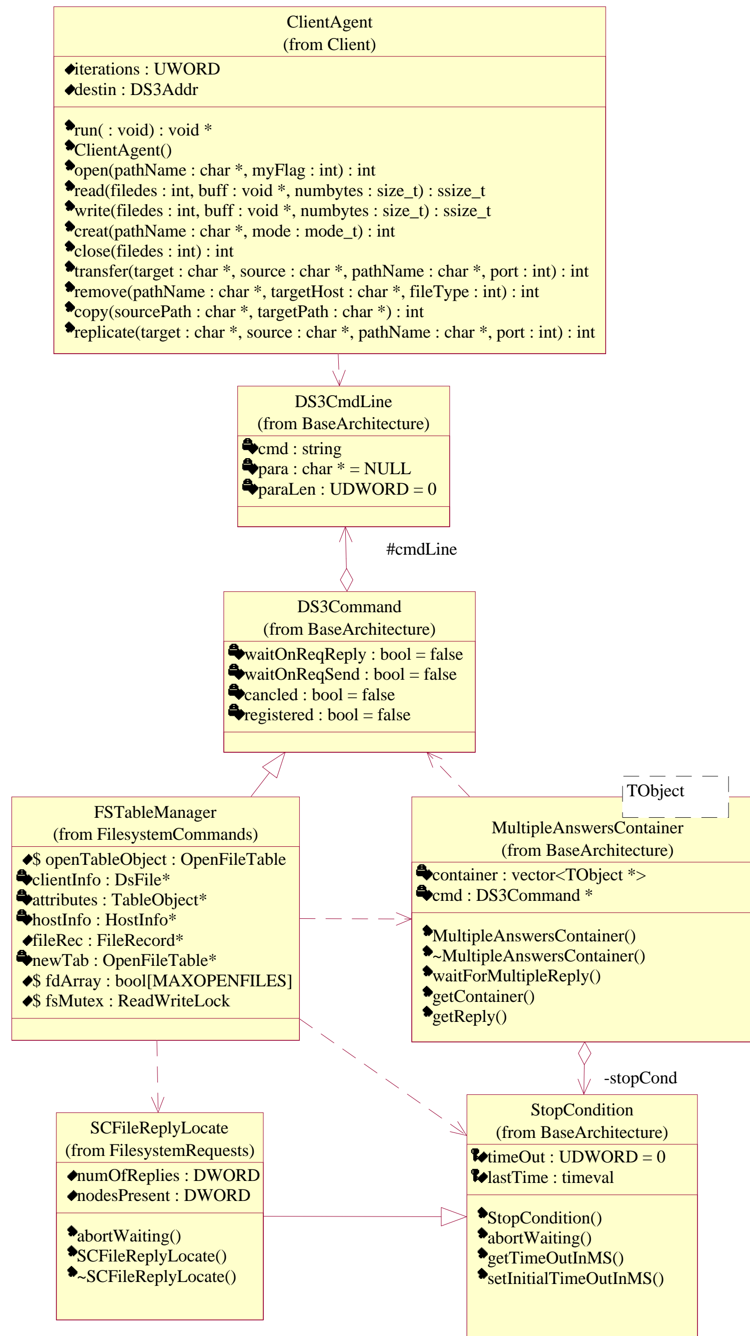


Figure 8.2: The client module

Line 15 to 20 The task is casted to the class DS3CmdLine, the parameters are set by using the method `setPara()` and the task is sent out by calling the `sendObject()` method.

Line 25 to 30 The threads are now waiting for the reply.

Line 30 to 40 The reply has arrived and is being evaluated.

Listing 8.1: client code

```
int ClientAgent::open (char *pathName, int myFlag)
{

RequestID bRID;
OpenReply *openReply;
reply = NULL;
char openCmdStr[512];
int fd = -1;
string openCmdToken = "OpenCommand";

sprintf(openCmdStr, "open(%s, %d)", pathName, myFlag);
task = new DS3CmdLine;

if (task) {

    char reqidbuf[200];
    task->sendTo(destin);
    ((DS3CmdLine *)task)->setCmd(openCmdToken);
    ((DS3CmdLine *)task)->setPara(openCmdStr, strlen(openCmdStr));

    if (sendReq->sendObject(task, true)) {
    DBUG_PRINT("ERROR",("Sending out request failed."));
    } else {
        DBUG_PRINT("INFO",("Request sent out successfully."));
        bRID = (((MacroMessage *)task)->getRequestID());

        waitOnReqReply = true;
        reply = (MacroReply *)recvReply->waitForObject(&bRID, 50000);
        waitOnReqReply = false;

        if (!reply) {
        bRID.PRINT();
        DBUG_PRINT("ERROR",("Error while receiving reply"));
        } else {
            bRID = (((MacroMessage *)reply)->getRequestID());

            openReply = (OpenReply *)reply;
            fd = openReply->getFileDescriptor();
            delete openReply;
            }
        }
```

```
    delete task;
    }
45  return fd;
```

8.4 The File System Commands

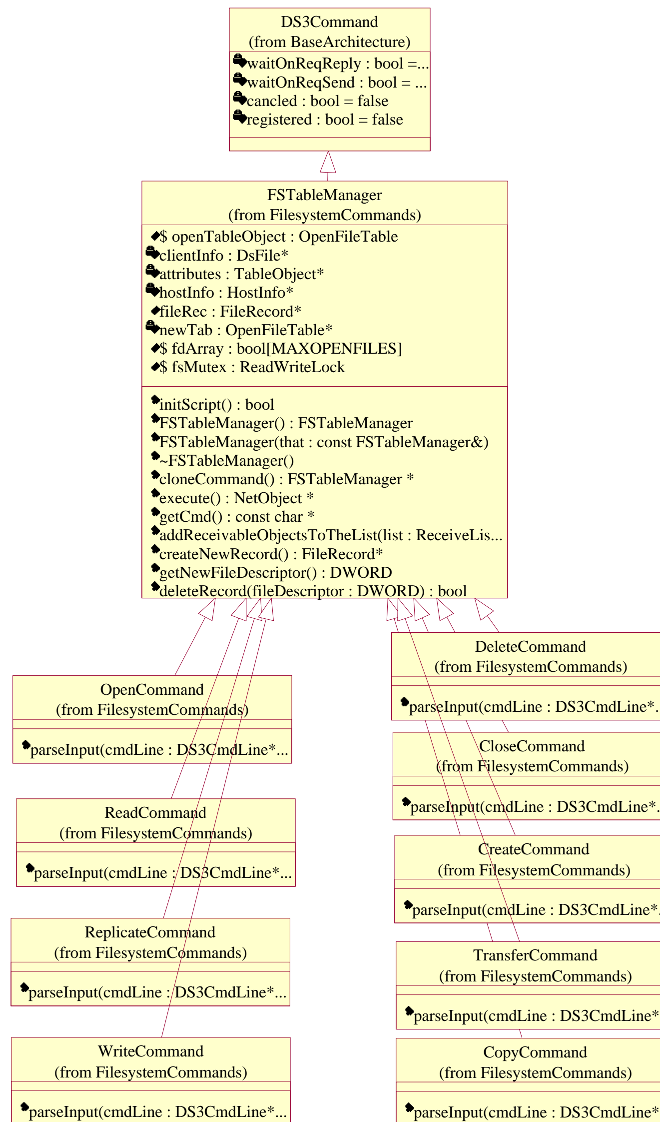


Figure 8.3: The class diagram shows the file system commands

As already mentioned above, the file system commands are nested at the Command Manager level. The class diagram in figure 8.3 illustrates the relation-

ship between the various commands implemented in this work. The commands illustrated in the diagram do correspond to the methods listed in the class clientAgent presented in figure 8.2. The diagram shows the inheritance of the different classes present. As mentioned earlier, the class FSTableManager is the base class of all the file system commands. This class FSTableManager on the other hand is derived from the class DS3Command.

Most of the methods and attributes of the file system command classes have been hidden because they all contain the same methods inherited from the base class FSTableManager. The method `parseInput` shown on figure 8.3 is used by all commands to parse a command line and set the values of the parameters. All the commands which have been derived from this base class have to implement the following methods:

`cloneCommand()` This function clones the command object by calling the copy constructor of the object.

`execute()` This function implements the actual steps necessary for creating a request and receiving the reply in the command. It is invoked by the agent.

`getCmd()` The method returns a constant pointer to the command string

`addReceivableObjectsToList()` Returns a list of macro replies which are expected to be received by the command.

Listing 8.2: Open file code

```
   NetObject * OpenCommand::execute ()
   {

   OpenReply *openFileReply = NULL;
 5 parseInput(cmdLine);

   if (!executeLocateReq()) {
     OpenReply  *openFileReply = new OpenReply;
     openFileReply->setResultFlag(false);
10   openFileReply->setRequestID(openID);
     return (NetObject *) openFileReply;
   }

   RequestID reqID;
15 UDWORD timeOut;
   SCFileReplyLocate *stopCond = new SCFileReplyLocate;
   OpenRequest *aRequest = new OpenRequest;

   aRequest->setPathName(pathName);
20 aRequest->setFlags(flags);
   aRequest->generateNewID();

   reqID = aRequest->getRequestID();
```

```
  makeReservation(reqID);
25
  if(flags == 1 ){
    //When opening for write send a broadcast

    DS3Addr bcstAddr((long)INADDR_BROADCAST, PF_NM_RQ_PORT->getDWord());
30  aRequest->sendTo(bcstAddr);
    sendObject(aRequest, true);

    // Wait for 50 sec for a reply

35  stopCond->setInitialTimeOutInMS(5000);
    stopCond->setNodesPresent(getCurrentNumOfNodes());
    MultipleAnswersContainer <OpenReply>  *answerContainer;

    answerContainer=new MultipleAnswersContainer<OpenReply>(this,stopCond);
40  answerContainer->waitForMultipleReply(reqID);
    openFileReply = answerContainer->getReply();
    while (openFileReply)
     {
      if (evaluateOpenReply(openFileReply))
45      break;
        openFileReply = answerContainer->getReply();

     }

50  delete answerContainer;

  }else{
    //When opening for read send the request to the local node

55  aRequest->sendTo(newTargetAddr);
    sendObject(aRequest, true);

    // Wait for 50 sec for a reply

60  stopCond->setInitialTimeOutInMS(5000);
    stopCond->setNodesPresent(getCurrentNumOfNodes());
    timeOut = stopCond->getTimeOutInMS();
    if(timeOut) {
        openFileReply = dynamic_cast<OpenReply *>
65                                 (recvObject(&reqID, timeOut));
        if(openFileReply){
        evaluateOpenReply(openFileReply);
        openFileReply->setRequestID(openID);
        }
70    }

   }

     if(!openFileReply) { //no answer received
```

```
75        OpenReply   *openFileReply = new OpenReply;
          openFileReply->setRequestID(openID);
      }

      cancelReservation(reqID);
80    DBUG_LEAVE;
      return (NetObject *)openFileReply;
   }
```

The listing in 8.2 documents the procedures taking place in the method `execute` which has to be implemented by all commands. I've used the open command to illustrate these procedures because all the commands follow the same schema.

Line 5 to 10 The method `parseInput` is called in order to evaluate the command line and set the values of the parameters. This method is documented in the listing 8.3. After the values of the parameters have been set, the method `executeLocateReq` is called up to initiate the lookup request. If this method returns a false, the execution will be terminated because the file does not exist.

Line 15 to 20 If the file exists, a request is created from the class OpenRequest. When the parameters of the request are set, a new ID is generated for this request and the request ID sent by the client is retrieved. The method `makeReservation()` is called to reserve a place for this request.

Line 25 to 30 The value of the flag is examined, if the flag is set to write only, a multi-cast will be sent to all the nodes with replicas by calling the method `sendObject()`.

Line 35 to 45 The object stopCond created from the class SCFileReplyLocate is used in combination with the container class MultipleAnswersContainer to wait for the series of open replies. The class MultipleAnswersContainer stops waiting for the replies if one of the stop conditions discussed in section 8.3 is fulfilled.

Line 52 to 55 If the flag is set to read only, the request will be sent by uni-cast to the local node, if it stores a replica of the file or else another node storing a replica will be chosen.

Line 60 to 70 In the case of read only where a request was sent to a single node, the class MultipleAnswersContainer is not needed because we know exactly that we are waiting for a single reply. If the reply did not arrive, a new object from the class OpenReply is created and sent back to the client, just to avoid sending a NULL pointer.

Line 75 to 80 The reservation made for the request ID is canceled by calling the method `cancelReservation()`. The open reply is then returned to the client.

FSTableManager also offers extra methods which can be accessed by all the derived commands if desired. These methods are:

createNewRecord() This method returns a new record of a file which has just been opened.

getNewFileDescriptor() This method is responsible for managing and controlling file descriptors.

deleteRecord() This method removes the record of a file which has been closed from the open file table map and sets free the file descriptor.

8.5 Command line parsing

The client program sends a command to the Command Manager for example an open command. The class DS3cmdLine gets the command line from the client and forwards it to the Command Manager. At the Command Manager, the instance of the open command is invoked and this command line is forwarded to the open command. The open command parses the command line with the method `parseInput(DS3cmdLine *cmdLine)`. The Listing 8.3 documents the method `parseInput`.

Line 10 to 20 A set of token delimiters are pre defined in order to use when parsing the string. The method `Strtok()` takes as parameter a string and the pre–defined token delimiters. This method `strtok()` parses a string into tokens. Each call to strtok returns a pointer to the next token, or NULL when no more tokens are found.

Line 25 to 30 After parsing the strings the values of the respective attributes are set.

Listing 8.3: Command line parsing code

```
void OpenCommand::parseInput (DS3CmdLine* cmdLine)
{


5   char myPara[i];
    int cmdLength = 0;

    cmdLength = cmdLine->getPara(myPara, i);
    myPara[cmdLength] = 0;
```

```
10      const int MAX_TOKEN = 10;
        char *token_delim = "(),␣\t";
        char *token_array[MAX_TOKEN];
        int index = 0;

15      token_array[index] = strtok(myPara, token_delim);
        while (token_array[index] != NULL && index < MAX_TOKEN){
              index++;
              token_array[index] = strtok(NULL, token_delim);
         }
20
         printf("Tokens:␣%d\n", index);

         for (int i = 0; i < index; i++)
         printf("Token␣%d:␣%s\n", i, token_array[i]);
25
         fileOperation = token_array[0];
        pathName = token_array[1];
         flags = atoi(token_array[2]);
```

8.6 The Open File Table components

When a file is opened in the Unix system, a file descriptor is returned to the calling process and it is also registered in the open file table. As earlier discussed in section 7.7.2, the Command Manager on each node stores information on all the open files in the open file table. The file descriptor returned by the system will be used for subsequent input and output operation on that particular file like read, write and close. The class diagram in figure 8.4 shows a couple of class diagrams which have been used to model the open file table.

The class OpenFileTable contains methods to be used when manipulating the open file table map represented by the class tOpenFileMap. The class TableObject contains the attributes of a file. The class DsFile contains the information about the process ID and the user who is currently logged in, at the client machine. The class HostInfo contains the system information and the class FileRecord acts as a collecting point to all the other three classes, that's why it has the same attributes as the three classes.

The file records collected at the class FileRecord will be inserted into the open file table with the corresponding file descriptor of each file. The file descriptor is the key used by the map and the value is the file record.

The three important methods used with the open file table map are described below. The open file table map is implemented as static, so that it will reside in memory. The open file table can only be manipulated by a single process at a time. That is why semaphores have been implemented. In order to query the map a read lock must be accessed and in order to modify the map a write lock must be granted.

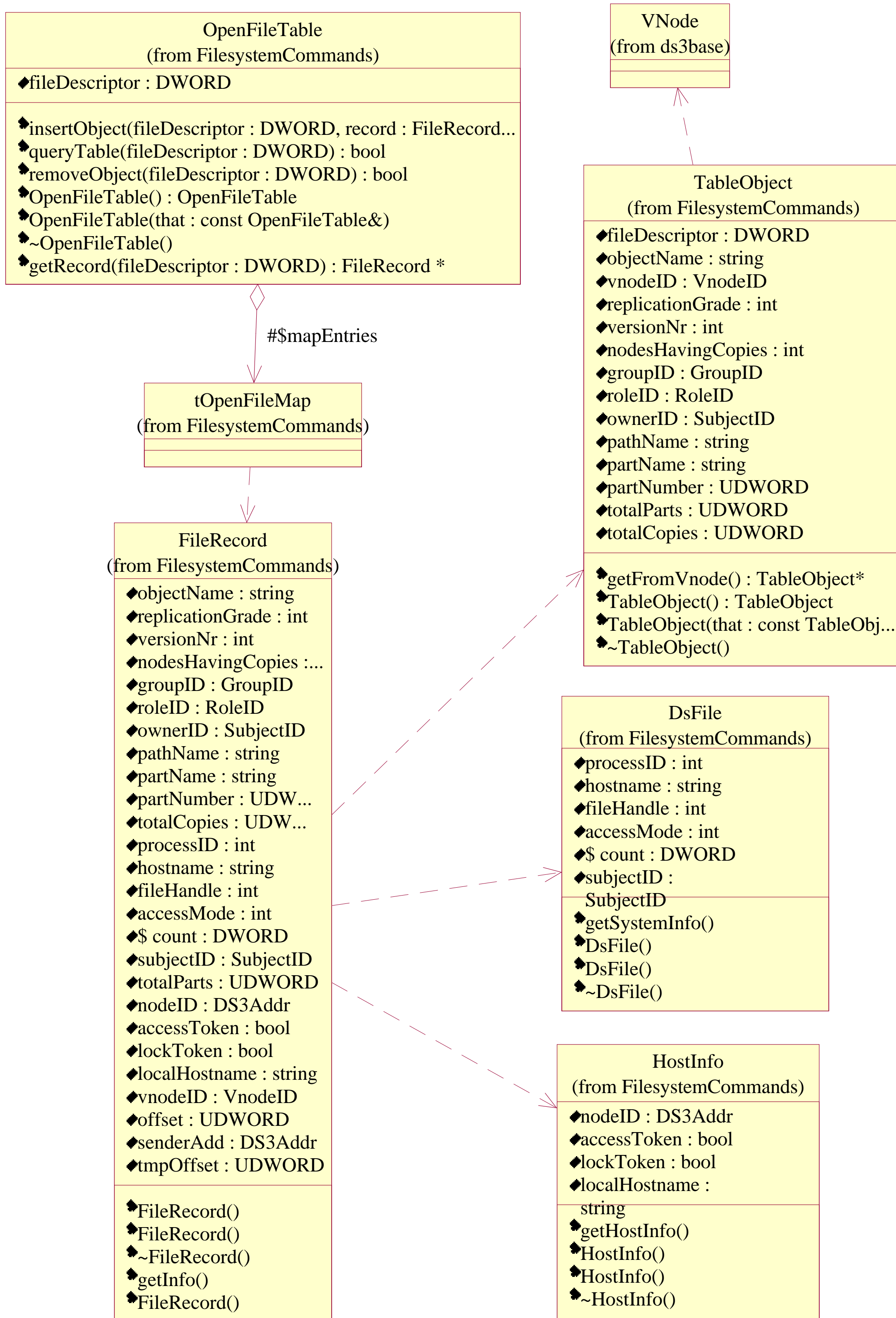


Figure 8.4: Class Diagram for the Open file table components

queryTable(fileDescriptor) The method `queryTable` gets a file descriptor as parameter and searches in the map for a similar key. A true is returned if the file descriptor already exist or else a false is returned.

insertObject(fileDescriptor, fileRecord) When the open command gets the open reply with the file descriptor, it calls the method `insertObject` from the class OpenFileTable. The method `insertObject` calls the method `queryTable` to find out if there is already a file descriptor in the open file table with the same number. If it does not exist, the new file descriptor will be inserted in the table. But if it already exists, the process will be terminated.

removeObject(fileDescriptor) After a close command has been issued, the file descriptor and it's file records will be removed from the open file table by calling the method `removeObject`.

8.7 The File System Requests

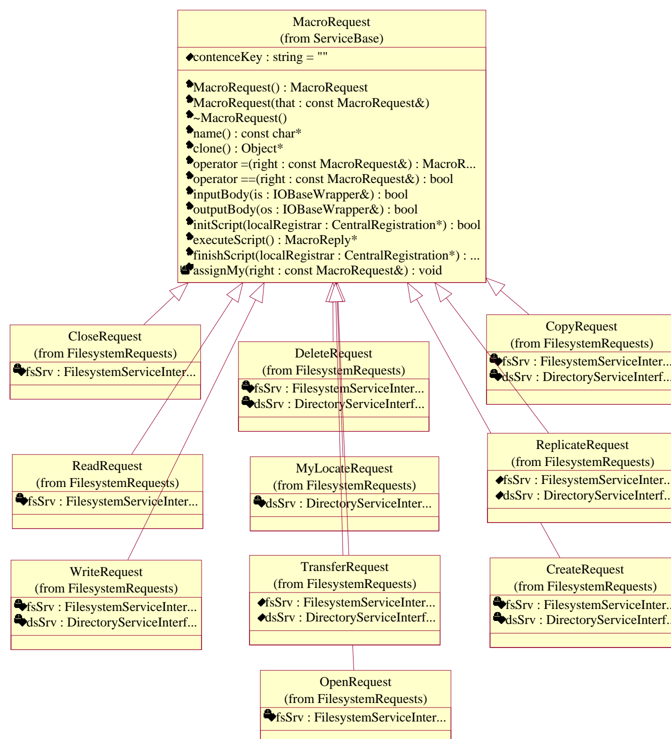


Figure 8.5: Class diagram for the Request

Figure 8.5 illustrates the request classes derived from the base class MacroRequest. In each request, the services called during the execution of the request have

been shown. The base class MacroRequest inherits from the class MacroMessage already presented in the class diagram of figure 8.1. Classes that are derived from this class can be transported using the different DRAGON SLAYER communication structures. The requests presented in the diagram match the commands in figure 8.3. These request are generated by their counterparts Commands at the Command Manager level.

For example, the OpenCommand generates an OpenRequest. After a request has been generated at the Command Manager level, it is sent to the Node Managers for processing. The requests inherit and overwrite all the methods from the base class MacroRequest. These methods are the same in each of the request classes presented in the diagram. There are a couple of very important methods present in each of the request like:

initScript() This method is used to demand the service instances from the centralRegistration which is referenced in the execution of the request.

executeScript() This method contains all the service calls issued by the request which are necessary for the processing of the users command. The replies are generated in this method and then returned to the Command Manager.

operator=() The assignment operated is overridden by all the request.

operator==() This operator tests the messages to see if they are the same.

clone() This method returns an exact copy of the object.

name() This method returns the name of the request.

inputBody() This method reads all the objects' data from a stream.

outputBody() This method on the other hand writes all the objects' data to a stream.

finischScript() When this method is called, all instances of the service implemented which were requested in the method `initScript` are returned to the centralRegistration.

The File System Replies

After the requests have been processed at the nodes, replies are sent to the commands at the Command Manager which issued the request. Figure 8.6 illustrates the reply classes derived from the base class MacroReply. The replies presented in the diagram matches the requests in figure 8.5 and are generated by their counterpart request. The base class MacroReply also inherits from the class MacroMessage already presented in the class diagram of figure 8.1. The reply classes illustrated on the diagram inherits some methods and variables from

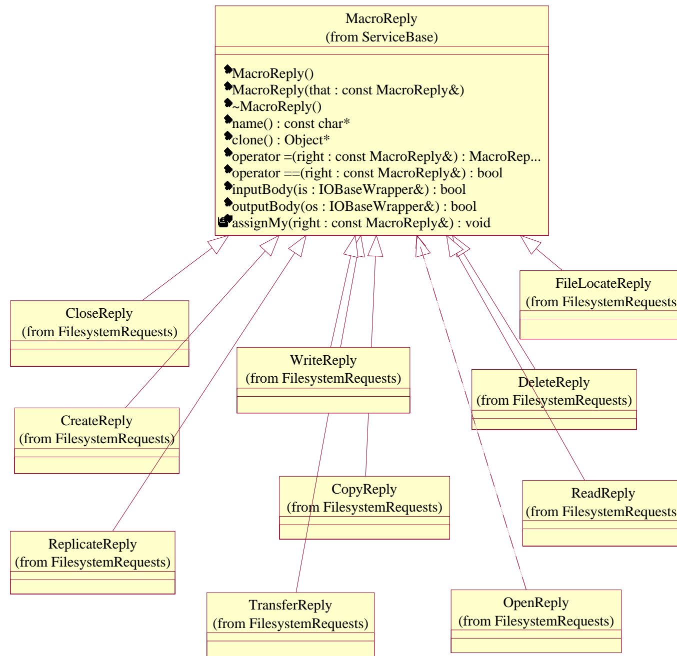


Figure 8.6: Class diagram showing the replies

the base class MacroReply. These methods are exactly the same methods implemented by the request in section 8.7 excluding the methods `initScript()`, `executeScript()` and `finishScript()`.

8.8 The File System Service Interface

When the file system commands issue various requests, these requests get to the Node Managers, which then invoke the required methods at the FilesystemInterface. The service modules have been split-ted into two classes, the class FilesystemInterface and FilesystemServiceImp.

Figure 8.7 shows the base class of all the Dragon Slayer services DS3ServiceInterface and DS3ServiceImp. The class diagram FilesystemServiceInterface is derived from the base class DS3ServiceInterface. A constructor is needed in the interface class which calls the constructor of the class DS3ServiceInterface in order to set the correct service identification number. The class FilesystemServiceImp has been derived from the FilesystemServiceInterface and from the base class DS3ServiceImp. As mentioned earlier, the file system service is situated at the Node Manager level.

The methods used by the request must be declared abstract in the class FilesystemInterface. These methods are overridden by the class FilesystemServiceImp. This design has been used so as to minimize the degree of interaction between

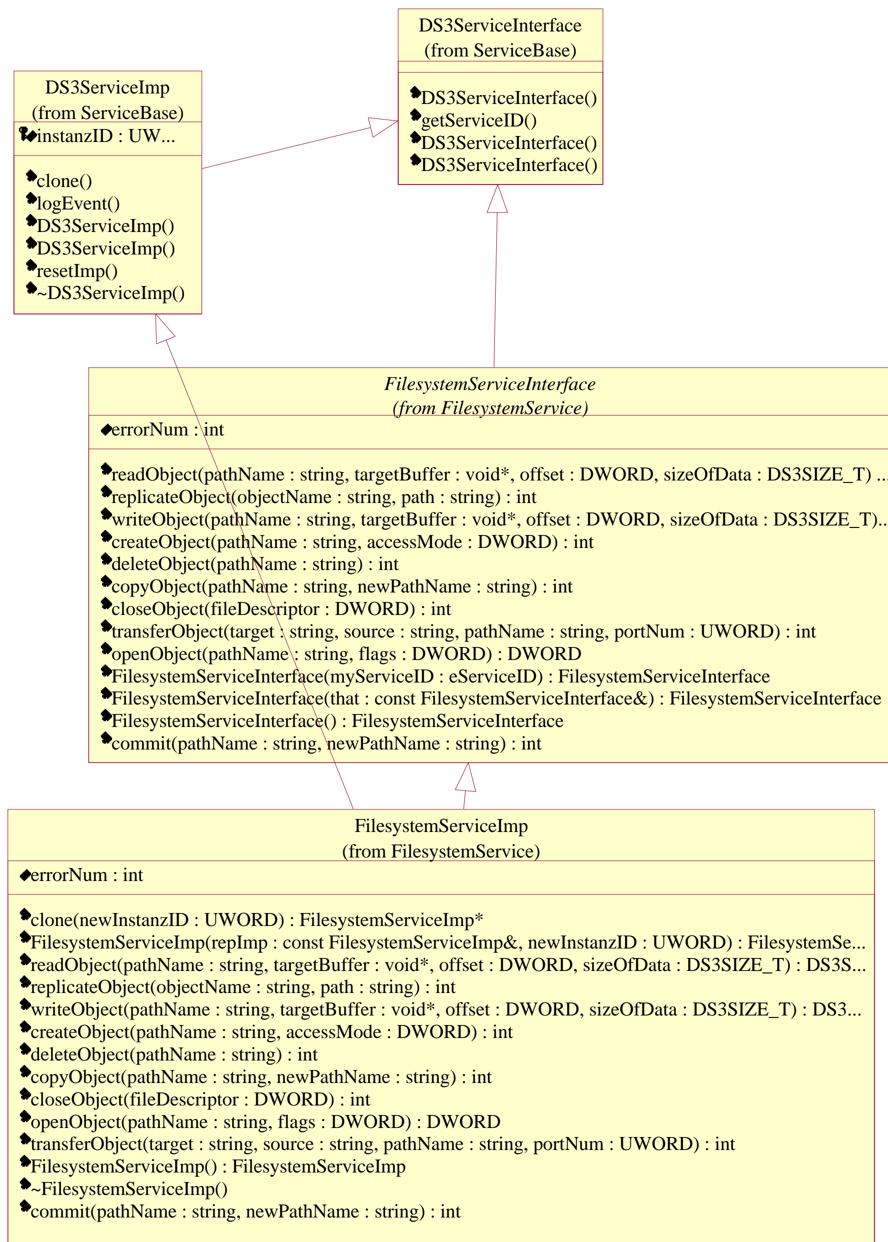


Figure 8.7: The class diagram illustrates the file system service Interface

the services and the Command Manager. The method `executeScript` found in all the requests which are created by the Command Manager does the actual calls to the services, by accessing the methods in the class FilesystemInterface. The Command Manager does not have any idea about the actual implementation of the services. Any changes undertaken at the service implementation will not affect the functioning of the Command Manager.

The real processing of the methods called by the requests take place at the FilesystemServiceImp. From earlier discussions, it was mentioned that the Node Managers are stateless, therefore it is necessary for all the request arriving at the node to contain all the information and parameters needed for processing.

8.9 Creating files and fragments

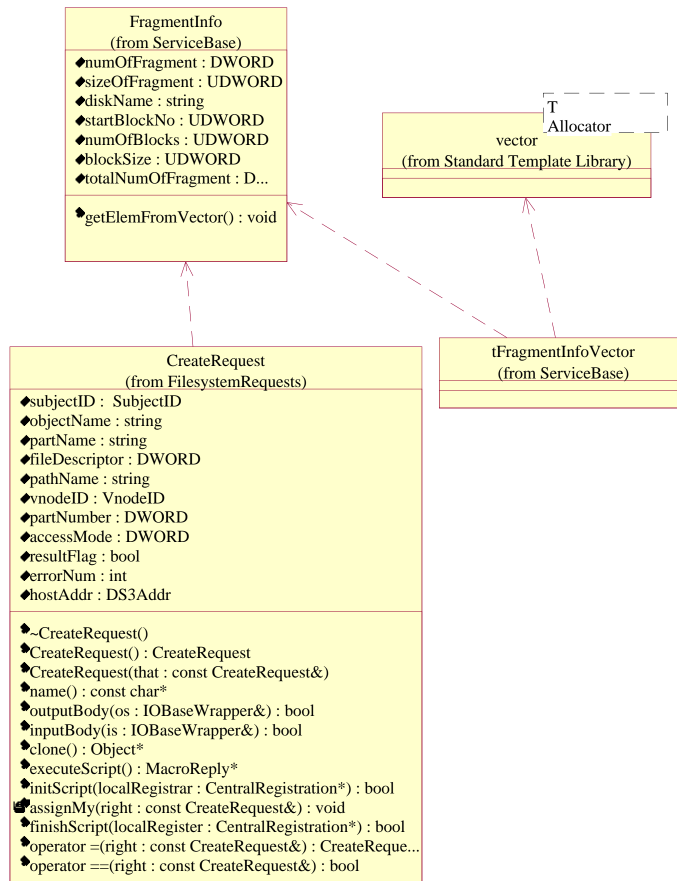


Figure 8.8: Class diagram illustrates the create method

Figure 8.8 shows the class FragmentInfo and tFragmentInfoVector which are used to create new fragments and to store information on fragmentated files.

The vector tFragmentVector is used to store the information of fragmented file at the local nodes. Also present on this diagram is the class CreateRequest which depends on the class FragmentInfo. When a file is being created, information about the file is gathered and forwarded to the Directory Service. If a file needs to be fragmented, the create request stores the information in the tFragmentVector after fragmentation has taken place and forwards it to the Directory Service.

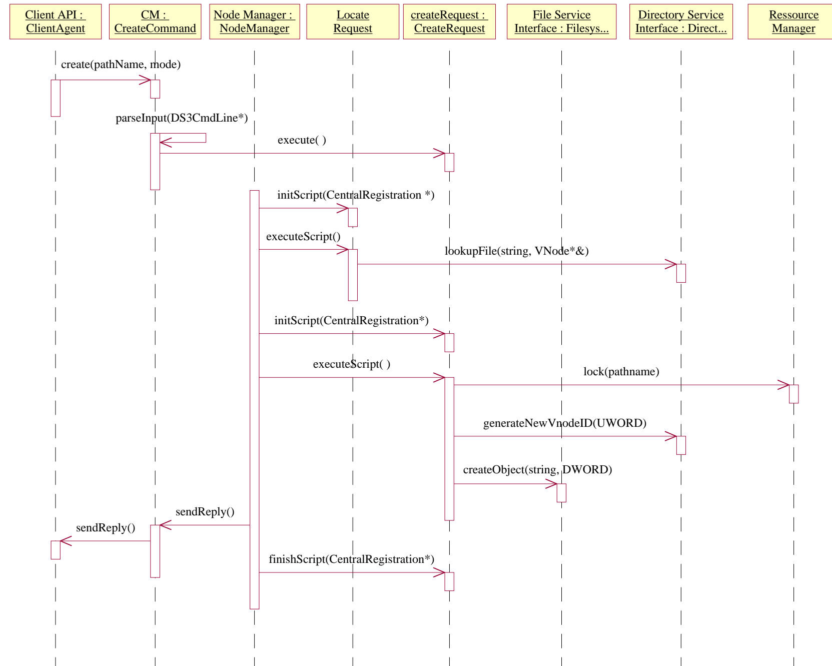


Figure 8.9: Sequence diagram for the create method

The sequence diagram in figure 8.9 illustrates the various methods called when a client program issues a create command. The client sends a command to the Command Manager, to create a file by calling the method `create(pathname, mode, targetNode)`. When the Command Manager receives the command line from the client, it forwards it to the create command. The command line is parsed by calling the method `parseInput` described in the listing 8.3 in order to set the values of the parameters. After the parameters have been set, a lookup request is prepared and sent out via broadcast by calling the method `lookupFile(pathname, vnode)`.

When this request arrives the node the Node Manager invokes an instance of the directory service to process it. If the file exists the Directory Service returns the vnode of the file, which contains all information about the file. If the file does not exist, an empty vnode is returned.

The Command Manager waits for the replies coming from the nodes after the stop condition has been set as discussed in section 8.3. When the majority of

the replies have arrived, these replies are evaluated in order to find out which of the nodes store a copy of the file, or if the file exists at all. If the file exists, the Command Manager sends a message to the client and stops the transaction.

If the file does not exist, a request is sent to the Resource Manager at the target node by calling the method `lock(pathname)` to grant a write lock. If the write lock is granted, a create request is sent to the target node. It is important for the client to explicitly give the name of the node on which the file will be created.

When the create request arrives the node, the Node Manager agent invokes an instance of the File Service. The file is created by calling the method `createObject(partname, mode)`. If the file has been created successfully, the method `createFile(pathname, vnodeID, fragmentVector)` is called to notify Directory Service that a new file has been created at that node.

If everything is successful, a result flag is sent to the Command Manager with a create reply which is forwarded to the client.

8.10 File replication and relocation

The class diagram in figure 8.10 illustrates the classes used for modeling file replication and file relocation. The structure of the module used for file replication is similar to that used for file relocation, that's why I've modeled the two operations in the same class diagram. The replication module uses an agent implemented in the class ReplicateRequest and a thread implemented in the class FileReplicateThread. This class FileReplicateThread is derived from the base class Thread. This class also depends on the communication class IODS3Socket to communicate with the agent at the target node.

The relocation module also uses an agent implemented in the class TransferRequest and a thread implementation in the class FileServiceThread. This class FileServiceThread is also derived from the base class Thread. This class also depends on the communication class IODS3Socket to communicate with the agent resident at the target node.

Figure 8.11 shows the collaboration diagram for the transfer process.

Listing 8.4: Transfer code

```
MacroReply* TransferRequest::executeScript ()
{

  TransferReply *reply = new TransferReply;
5 int bytesReceived = 0;
  int retReadVal = 0;
  int retWriteVal = 0;
  bool retConnect, retListen;

10 struct {
```

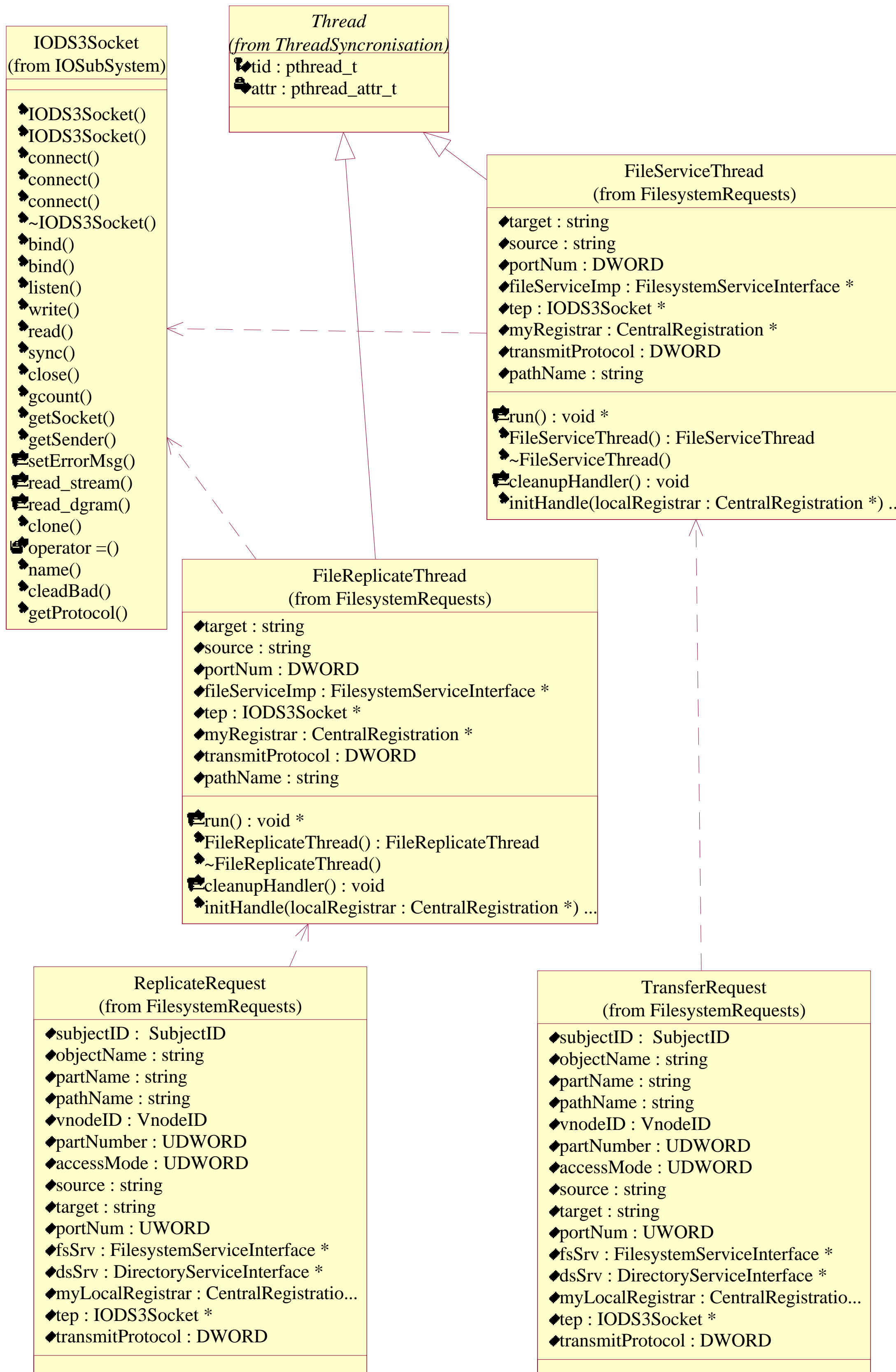


Figure 8.10: Class diagram for replication and relocation

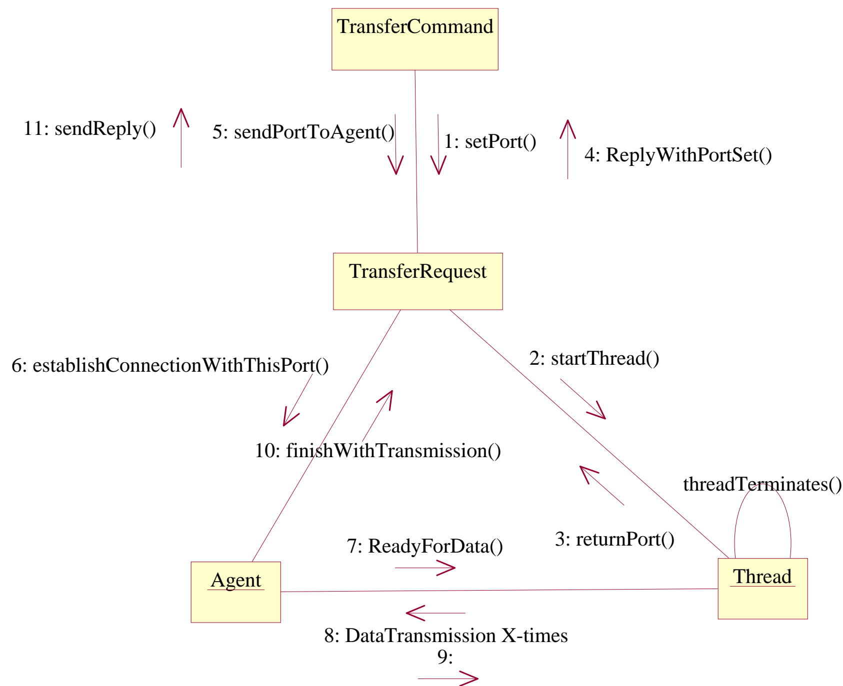


Figure 8.11: Collaboration diagram for relocating a file

```
    int length;
    int offset;
    bool fragment;
    char buffer[1000];
15  } transferData;

    int transferToken = 100;

    if (getPortNum() == 0) {
20    // Sender thread
      FileServiceThread *senderThread = new FileServiceThread;
      // Sender thread starten
      senderThread->setTarget(target);
      senderThread->setSource(source);
25    senderThread->setPathName(pathName);
      senderThread->vnodeID = vnodeID;
      portNum = senderThread->initHandle(myLocalRegistrar);
      senderThread->start();
    }
30  else {
     // Receiver agent

     tep = new IODS3Socket(SOCK_STREAM);
     retConnect = tep->connect(source, portNum);
35
```

```
        fsSrv->createObject(pathName, 0644);

        tep->write((unsigned char *)&transferToken,sizeof(transferToken));
        tep->sync();
40      while (true) {
        tep->read((unsigned char *) &transferData,
                                 sizeof(transferData),-1,-1);
        fsSrv->writeObject(pathName, transferData.buffer,
                           transferData.offset, transferData.length);
45          if (transferData.fragment)
              break;
                  }

        tep->write((unsigned char*)&transferToken,sizeof(transferToken));
50      tep->sync();
        tep->close();

      }

55      reply->setPortNum(portNum);
       DBUG_RETURN(reply);
   }
```

The listing in 8.4 documents the procedures involved when transferring a file from one node to another. The client sends a command to the Command Manager, to transfer a file. The Command Manager receives the command line from the client and forwards it to the transfer command. The command line is parsed as described in section 8.5. After the parameters have been set, a lookup request is prepared and sent out via broadcast. This request arrives the node and the Node Manager invokes an instance of the Directory Service.

The Command Manager waits for the replies coming from the nodes. When the stop condition has been fulfilled, these replies are evaluated in order to find out, which of the nodes stores a copy of the file, or if the file exists at all. By evaluating the replies, the Command Manager knows if the target node already has a copy of the file or not. If the file does not exist, the Command Manager sends a message to the client and stops the transaction.

Line 15 to 30 When the transfer request arrives the node, the Node Manager invokes an instance of the File Service. A sender thread is created at the node and a random port number is chosen by the thread to be forwarded to the Command Manager with the transfer reply. The thread is then started. When the Command Manager gets the port number from the reply, it is used to send a second transfer request to the target node.

Line 30 to 35 The second transfer request arrives the node and an agent is created at the target node. The agent gets the port number from the request which was chosen by the source node as the future communication

port for the transaction. The agent then calls the method `connect` in order to build up the TCP connection with the sender thread.

Line 35 to 50 The target nodes sends a message to the sender thread at the source node, indicating that it is ready to receive data. When the source node receives this message, transmission of data from source node to target node will proceed. The agent calls the service implementation to create the file at the target node. The data received by the agent from the sender will be written in the empty file just created. After the receiving agent has receives the last set of data, it sends a reply to the Command Manager with the result flag set to success or failure.

Listing 8.5: Transfer thread code

```
int FileServiceThread::initHandle(CentralRegistration *localRegistrar)
{
  bool retVal;
5 tep = new IODS3Socket(SOCK_STREAM);

  retVal = tep->bind(INADDR_ANY, 0);
  portNum = ntohs(tep->getPort());

10 DS3ServiceInterface *fsImp, *dsImp;
  myRegistrar = localRegistrar;
  fsImp = myRegistrar->getServiceImp(DS3_FileService);
  if(fsImp) {
    fileServiceImp = dynamic_cast<FilesystemServiceInterface*>(fsImp);
15  }

  //For the directory service
  dirRegistrar = localRegistrar;
  dsImp = dirRegistrar->getServiceImp(DS3_DirectoryService);
20
  if(dsImp){
  dsSrv = dynamic_cast<DirectoryServiceInterface*>(dsImp);
   }
  return portNum;
25 }

void * FileServiceThread::run ()
{

30 struct {
       int length;
       int offset;
       bool fragment;
       char buffer[1000];
35 } transferData;
```

```
    int transferToken = 100;
    int BUF_SIZE = 100;
    char dataBuf[BUF_SIZE];
40  char buf[BUF_SIZE];
    int bytesRead, bytesWritten;
    int bytesReceived;

    transferData.offset = 0;
45  transferData.fragment = 0;
    tep->listen(5, 0, 0);
    tep->read((unsigned char *) &transferToken,
                            sizeof(transferToken), -1, -1);

50  while (true) {
        transferData.length = fileServiceImp->readObject(pathName,
             transferData.buffer, transferData.offset, 1000);

        if (transferData.length < 1000)
55              transferData.fragment = true;

        tep->write((unsigned char *) &transferData,
                                            sizeof(transferData));

60      if (transferData.fragment)
           break;

        transferData.offset += transferData.length;
   }
65
   tep->sync();
   tep->read((unsigned char *) &transferToken,
                                  sizeof(transferToken), -1, -1);
   tep->close();
70 fileServiceImp->deleteObject(pathName);

   //Inform the directory service
   tErrorNo result = 0;
   VnodeID *delVnodeID;
75 delVnodeID = &vnodeID;

   result = dsSrv->deleteFile(delVnodeID);

   myRegistrar->freeServiceImp(fileServiceImp);
80 dirRegistrar->freeServiceImp(dsSrv);
   }
```

The listing in 8.5 documents the sender thread.

Line 5 to 25 The method `initHandle` gets a random port number to be returned to the transfer request and then it initializes the file system service implementation and the Directory Service implementation.

Line 26 to 50 The real processing takes place in the method `run`. The sender list calls the method `listen` to eavesdrop on that port number. As soon as data is on the stream, it uses the `read` to read the data from the stream.

Line 51 to 65 After the sender thread has received the token sent by the agent at the target node, it calls the service implementation to read the data from the file that has to be relocated into a buffer. The method `write` is called to read the content of the buffer and write it to the stream.

Line 70 to 75 When the sender thread is finished sending all the data, it deletes the file at the source node and the thread terminates. The file service notifies the Directory Service at the target node that a file has been created, and the Directory Service at the source node will be notified that the file has been deleted at that node.

The implementation of the replicate command differs from that of the transfer command described above only in one point. After the sender thread has finished with the transmission of the data, the file will not be deleted at the source node.

8.11 Testing the modules

8.11.1 Testing the communication classes

All the communication classes which have been derived from the base class MacroMessage must undergo the NetObject test. The class NetObject is the base class to the MacroMessage. The NetObject class provides an interface for transmitting objects over an TCP/IP network. When a class passes the NetObject test, the class can be transported over the network.

The listing below documents the NetObject test. In this listing, the class WriteRequest is to be tested. During the NetObject test, three objects are created from the class which is to be tested. Object A and B must have the same content while Object C differs from both of them. All the variables of the class WriteRequest have to be tested. First of all, the "set" methods are used to assign values to the variables. As you can see from the listing, the values of object A and B are always the same.

Listing 8.6: NetObject Test

```
  WriteRequest *objA, *objB, *objC;
      objA= new WriteRequest;
      objB= new WriteRequest;
      objC= new WriteRequest;
5
      objA->setPathName("me");
      objB->setPathName("me");
```

```
        objC->setPathName("you");

10      objA->setObjectName("test");
        objB->setObjectName("test");
        objC->setObjectName("test2");

        objA->setPathName("me");
15      objB->setPathName("me");
        objC->setPathName("you");

        objA->setFileDescriptor(2);
        objB->setFileDescriptor(2);
20      objC->setFileDescriptor(5);

        objA->setPartName("we");
        objB->setPartName("we");
        objC->setPartName("me");
25
        objA->setBytesToBeWritten(50);
        objB->setBytesToBeWritten(50);
        objC->setBytesToBeWritten(60);

30      objA->setBytesWritten(50);
        objB->setBytesWritten(50);
        objC->setBytesWritten(60);

    NetObjTest<WriteRequest> *writeTest;
35  writeTest= new NetObjTest<WriteRequest>
                      ("WriteOutput",objA,objB,objC);
    writeTest->TEST();
```

When these variables have been assigned values, the NetObject test is started. If any of the methods encounters an error the test is terminated. The test consists of the following:

TEST_name This method is used for testing the name of a class, if it has been written correctly.

TEST_clone This method is used to test if the name of the class can be cloned successfully.

TEST_operator= This method is used to test the assignment of the values to the variables.

TEST_operator== This method is used to compare if the contents of object A and B are equal. If A is equal to B and A is not equal C, the test will be successful otherwise it is a failure.

TEST_outputBody This method tests each variable to see if it can be written to the transportation medium.

TEST_inputBody This method tests each variable to see if it can be read from the transportation medium.

For the NetObject test to be successful, each of the tests described here must be successful. Any class that has not completed this test cannot be transported over the network.

8.11.2 Working with log files

In order to debug a program effectively, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To eradicate bugs from the program, a debugger has been used. All the methods implemented did includ the debug file, which is primarily useful for debugging purposes. When the program is running, the print function defined in each file, prints a summary to a log file. The log files created are used to trace the bugs in the Command Manager agents and in the Node Manager agents. This is an example of a debug output. From the log file, one can see the names of the classes and the methods being executed.

Listing 8.7: Log file used for debugging

```
class NetObject * BaseMgrSendEntry::getItem()
INFO: Got a DS3CmdLine object from the queue
class NetObject * BaseMgrSendEntry::getItem()
bool BaseMgrSenderThread::mainloop(NetObject *)
bool IODS3Socket::connect(unsigned long adresse, int port)
DS3Socket& IODS3Socket::write(unsigned char *buf, unsigned int len)
INFO: Writing 1 bytes
INFO: In Buffer 1 bytes
```

The purpose of a debugger is to allow one to see what is going on "inside" another program while it executes–or what another program was doing at the moment it crashed.

8.11.3 Interactive test

In order to test the file service modules described above, I used a single Command Manager agent, three Node Managers and one client. A Command Manager, a Node Manager and a client were started on the node “Kriemhild”. At two other nodes “Hagen” and “Siegfried” only the Node Managers were started.

The client sends the commands to the Command Manager, which forwards the request to the Node Managers at the nodes. Depending on the type of request sent by the client, the Command Manager might send a broadcast or a unicast to a particular node.

When testing with three nodes and employing the majority voting approach, if a broadcast is sent out by the Command Manager, as soon as it receives two replies, it can continue with the transaction. All the file operations implemented by the File Service have been tested and they are functioning properly.

Chapter 9

Conclusions and future work

This thesis presented the design and implementation of a distributed file service for the Dragon Slayer system. Emphasis was laid on the decentralized nature of the services in Dragon Slayer, because all nodes have identical services and these services must communicate with other services resident at other nodes. In such systems, one should be careful that communication overhead between the services should not degrade the performance of the system.

File operations have also been implemented, for the client application programs to use for accessing and manipulating data stored in the system. In order to avoid conflicts among processes, locks must be used to implement mutual exclusion when data is being accessed for modification. The lock mechanism for Dragon Slayer has not yet been realized.

I have also revisited other topics around the file systems like storage technologies, version number management and file replication. The replication process is very important to distributed and decentralized file systems. From the experiments carried out in [YV01] on availability as a function of the number of replicas, quantifies the intuition that additional replicas will not always improve service availability and can in fact reduce it. Thus, one must carefully balance the marginal availability benefit of additional replicas against the cost of maintaining consistency for a given fault load and consistency level. It is also clear that increasing the number of replicas will also increase read availability since a large number of clients will be able to access the data. The file service module has been implemented and tested.

9.1 Quality of Service

The planned allocation and scheduling of resources to meet the needs of multimedia and other applications is referred to as quality of service management [CD01]. The Quality of Service that the File Service intends to offer applications is based on storage management and the bandwidth management. Each application has

its own requirements which it needs to perform a task well. Some applications may demand that the data transfer be processed on a high bandwidth and other applications might prefer to reside on disk arrays. Since disk arrays provide higher availability than single disks, some applications might perform well on such facilities or some might prefer the solid-state storage devices (SSD), which electrically store data in battery powered DRAM memories. These devices have significantly lower latencies and higher transfer rates than disks and disk arrays.

The quality of service addressed here is only applicable to data storage, input and output. The underlying storage facilities and the I/O interfaces are of great importance to us. Traditionally, interfaces fall into two categories: channels and networks. Current trends merge channels and network into single interfaces, for example fiber channel. Channel interfaces such as SCSI predictably transfer data with low-latency and high-bandwidth performance. Network interfaces on the other hand are more flexible, have high connectivity, connect long distances and operate in unpredictable environments.

For optimal performance of the file system, all the storage devices lying underneath the file system should be equally loaded. This enables the file system to derive maximum benefits of parallelizing the data access from the devices. This issue will be dealt with in the thesis of [Her02], work is in progress. The devices can have different data rates and storage capacities. For example, assume that there are two disks in the pool, one having a bandwidth of 2MB/s and capacity of 2GB and the other having a bandwidth of 4MB/s and 10GB capacity, i.e., the file system can support a maximum of 6MB/s data rate.

The File Service will have to maintain a resource table that contains information about the physical characteristics of the storage devices, the I/O interface and the bandwidth. The values and the range of permissible variation of the Quality of Service parameters will be stated explicitly.

In order to negotiate a Quality of Service between an application and the underlying system, the application must specify its Quality of Service requirements to the File Service. This is done by the transmission of a set of parameters.

The File Service evaluates the feasibility of meeting the requirements against the resource table, to find out which resources are available and which resources are currently committed before it gives a positive or negative response. If it is negative, the application may repeat the request after decreasing its demands.

If the response is positive, the resources will be reserved for the application and it will be given a time limit. The application is then free to run and when it terminates, it releases the resources.

Quality of Service management is needed in order to guarantee that applications obtain the necessary quantity of resources at the required time, even when other applications are competing for the resources [CD01].

The Quality of Service does not create more storage but manages what is available so that it can be used more effectively to meet the requirements of applications.

9.2 Parts and fragments

To add more features to file partitioning, the resource manager could be designed with the possibility of locking only a region of a file. In that case, the lock operation will return a lock context, which must be used during subsequent read and write operations to the region. Lock operations indicate the caller's intention to access a portion of a region.

An example of a system that has implemented this is Khazana presented in [CRS98]. Khazana has been designed as a middle-ware layer for arbitrary distributed applications. The concurrency protocol used here ultimately decides the concurrency control policy based on the stated intentions to provide the required consistency semantics. A region's attributes in Khazana currently include: the desired consistency level, consistency protocol, access control information and a minimum number of replicas.

The consistency contract should specify the properties that the resource manager protocol must fulfill e.g. atomicity. A consistency protocol implements a consistency contract.

Working with parts and fragments can only be fully implemented if most of the services to be offered in DRAGON SLAYER have already been implemented.

9.3 Integrating VFS, the File System and LVM

In chapter 3 , we described the Virtual File system (VFS) and in chapter 4 the Logical Volume Manager. Every file in the system has a vnode. For most files, the vnode also contains the inode for the file. The inode contains the owner of the file, the size of the file, the device the file is located on, pointers to the actual data blocks where the file is located on disk, and so on. The directory service stores to each file the vnode number of that file.

A project Group was already scheduled to continue with work in this area [WLS01]. In most systems, all files have inodes. These inodes could be extended to include an entry which indicates if a file has been partitioned and if that is the case, the number of parts present and the names of the various parts should also be included. The names of each part should be accompanied by their unique IDs. The list of pointers to the data blocks of the parts should also be stored with them.

The VFS is an object -oriented interface which defines virtual VFS and vnode operations. Each installed file system provides the kernel with functions associated with VFS and VFS-vnode operations. Vnode operations manipulate individual files. A VFS-vnode is the VFS virtual equivalent of an inode.

VFS creates and passes VFS-vnodes to the file system vnode operations. VFS-vnode operations include opening, closing, creating, removing, reading, writing and renaming files. To open a file for example, VFS passes the file path name to

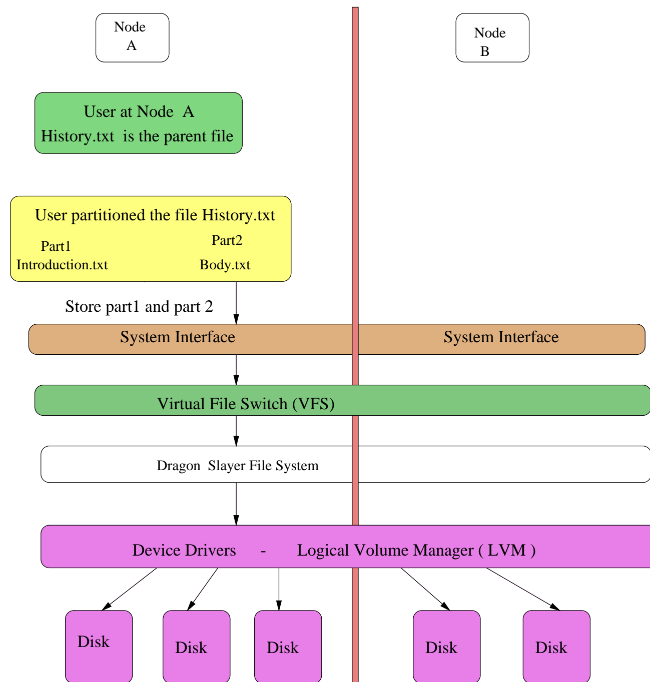


Figure 9.1: The integration of the LVM, the VFS and the File System

the vnode operation, vop _ lookup. Vop _ lookup maps to a file system specific lookup function. This function searches the name-space for each qualifier in the path name. This lookup routine then finds the file and returns the file vnode to the caller. Read and write system calls invoke vop _ read and vop _ write operations. Vop _ read and Vop _ write call vop _ bmap routines which translates file byte offsets to block numbers.

Since Logical Volume Manager (LVM) makes storage management much more flexible by allowing storage to be reallocated easily at any time, storing and accessing of the parts will automatically be done by LVM.

When storing parts, the users requests are received by the VFS interface which then calls the Dragon Slayer III routines to perform the task. Since VFS has direct contact to the disk drivers, it will then forward the data to be stored to the LVM. The LVM then decides on which volumes it deem necessary to store the file.

When retrieving the parts of a file, the part names or the part numbers to be read or modified are passed over to the VFS with the request. Since VFS is in direct contact with the underlying disk drivers, it will then take full control of this process, by forwarding the request to the LVM to retrieve the parts. By combining these two technologies, it could improve the efficiency of the system when working with parts. The system will never grow short of storage because with LVM, more logical volumes can be added as the system grow larger.

9.4 File version management

A primitive file version was implemented here in order to use in detecting and reconciling out-of-date replicas. There are a couple of good version management programs which could be used in DRAGON SLAYER like CVS. On the other hand, an application program could be designed and implemented to satisfy the architecture and design of the DRAGON SLAYER system.

9.5 Replication

Replication of data is an elaborate topic which needs to be taken care of by a dedicated service. Normally, a replication service should consist of a consistency manager which detects and resolves conflicts among concurrent accesses to the replicated object. It should also provide a mechanism to propagate updates to the replicas. Maintaining of replicas in DRAGON SLAYER will be dealt with more elaborately in future works.

Considering the functions of other services in DRAGON SLAYER and due to the systems architecture, I have restricted my work to the maintenance of data consistency among replicas before a read or a write request is processed. This has been achieved by implementing version consistency control.

A project group was already envisaged to deal with the problem of data consistency in the DRAGON SLAYER system as presented in [WLS01]. The aim of the project was to realize an efficient write operation on inconsistent data spread across several sites. Deltas update techniques were supposed to be used when updating data between sites.

The resource manager which is in charge of the allocation of resources in the DRAGON SLAYER system has not yet been implemented, so it is not possible for the file service to implement data consistency in the system without knowing which scheduling algorithm the resource manager will use.

Given the well-known tradeoffs between strong and optimistic consistency models, trends are moving towards a continuous consistency model for improving data availability as stated in [YV01]. The replication service of DRAGON SLAYER could be redesigned to allow applications to dynamically set their consistency level, degree of replication, and placement of replicas based on changing network and service characteristics to achieve a target level of data availability.

Bibliography

[Aga99] Manish M. Agarwal. System calls for automated online file system maintenance tools. Master's thesis, University of Minnesota, 1999.

[Aiv00] Tigran Aivazian. Linux kernel internals - virtual filesystem (vfs). http://www.moses.uklinux.net/patches/lki.html, 2000.

[ARSW92] Farhad Abar, Gary Roedigers, Joseph Stith2, and Matt Wicks. Afs,andrew file system; fermilab final evaluation report and implementation recommendations. http://hepnrc.hep.net/hepnrc/reports/frame _ afs-eval.html, 1992.

[AW99] Thomas Anderson and Randolph Wang. xfs: A wide area mass storage file system. University of California, 1999.

[BB99] Michael Beck and Harald Beohme. *Linux Kernel-programmierung.* Addison Wesely, 1999.

[Ben00] Gnther Bengel. *Verteilte Systeme.* Vieweg Verlag, 2000.

[Bro99] Neil Brown. The linux virtual file-system layer. http://cgi.cse.unsw.edu.au/neilb/oss/linux-commentary/vfs.html, 1999.

[Bur97] Rainer Burkhardt. *UML – Unified Modeling Language, Objektorientierte Modellierung fr die Praxis.* Addison–Wesley–Longman, 1997.

[CD94] George F. Coulouris and Jean Dollimor. *Distributed Systems, Concepts and Design.* Addison-Wesley Publishing Company, 1994.

[CD01] George F. Coulouris and Jean Dollimor. *Distributed Systems, Concepts and Design.* Addison-Wesley Publishing Company, 2001.

[CRS98] John Carter, Anand Ranganathan, and Sai Susarla. Khazana* an infrastructure for building distributed services. http://www.cs.utah.edu/khazana, 1998.

[DKOT91] Fred Douglis, M. Frans Kaashoek, John Outerhout, and Tanenbaum. A comparison of two distributed systems: Amoeba and sprite. http://www.scs.carleton.ca/ csgs/resources/amoeba_papers.html, 1991.

[FR95] Bernd Finger and Michael Richter. Das dragon-slayer-2 system: Konzept und erstellung eines prototyps feur ein verteiltes dateisystem. Master's thesis, University of Dortmund, 1995.

[Fre99] Christian Frey. Entwurf und Implementierung einer verteilten Experimentierumgebung fr Resource-Scheduling Algorithmen. Master's thesis, Informatik III, Universitt Dortmund, Mrz 1999.

[Gos94] Andrej Goscinski. *Distributed Operating Systems*. Addison-Wesley Publishing Company, 1994.

[GRR99] Richard Guy, Peter Reiher, and David Ratner. Rumor: Mobile data access througgh optimistic peer-to-peer replication. http://citeseer.nj.nec.com/guy98rumor.html, 1999.

[Hei98] J. S. Heidemann. Perspectives on optimistically replicated, peer-topeer filing. http://citeseer.nj.nec.com/102426.html, 1998.

[Her02] Thomas Herber. Entwurf und erstellung eines schreib- bzw. leseoptimierten file systems formates fr verteilte anwendung. Master's thesis, University of Dortmund, 2002.

[HHB96] Abdelsalam A. Helal, Abdelsalam A. Heddaya, and Bharat B. Bhargara. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1996.

[Kee98] Matthew T. O Keefe. Shared file systems and fibre channel. http://citeseer.nj.nec.com/184183.html, 1998.

[Lis99] Barbara Liskov. Replication in the harp file system. http://citeseer.nj.nec.com/likov91replication.html, 1999.

[Lor00] Stephen Lord. Porting xfs to linux. http://linux–xfs.sgi.com/projects/xfs/papers/ols2000/olsXfs.htm, 2000.

[Mau99] Heinz Mauelshagen. Beginners guide to lvm. http://linux.msede.com/lvm/howto/en/BeginnersGuideToLVM, 1999.

[Pag90] Thomas W. Page. Replication in ficus distributed file systems. http://citeseer.nj.nec.com/jr91architecture.html, 1990.

[Pre99] Kenneth W. Preslan. Implementing journaling in a linux shared disk file system. http://citeseer.nj.nec.com/330434.html, 1999.

[Rat96] Rational Software Coporation. *Rational Rose – Using Rational Rose 4.0*, November 1996.

[Ren99] Harry Renshall. A new unix hierarchical storage management service – the hsm command. http://consult.cern.ch/cnls/236/unix_storage.html, 1999.

[RSK96] Thomas M. Ruwart, Steven R. Soltis, and Mathew T. O Keeffe. The global file sysytem. http://citeseer.nj.nec.com/25335.html, 1996.

[Rub97] Alessandro Rubini. The virtual file system in linux. http://gnu.systemy.it/docs/vfs/, 1997.

[SDH99] Adam Sweeney, Doug Doucette, and Wei Hu. Scalability in the xfs file system. http://linux-xfs.sgi.com/projects/xfs/papers/xfs-usenix/index.html, 1999.

[SG94] Abraham Silberschatz and Peter B. Galvi. *Operating System Concept*. Addison-Wesley Publishing Company, 1994.

[Sie98] Jens-Oliver P. Siepmann. Entwurf und implementierung der kommunikation des dragon slayer iii-systems. Master's thesis, University of Dortmund, 1998.

[Sol96] Steven R. Soltis. The design and implementation of a distributed file system based on shared network storage. http://www.sistina.com, 1996.

[SS94] Keith Smith and Margo Seltzer. File layout and file system performance. http://citeseer.nj.nec.com/smith94file.html, 1994.

[Sun95] Sun. Nfs white paper preserving file integrity. http://www.sun.com/software/white-Papers/wp-Nfs, 1995.

[Sun96] Sun. Nfs white paper preserving file integrity. http://www.sun.com/software/white-Papers/wp-Nfs, 1996.

[Tak99] TakeFive Software Inc., Austria. *Sniff+, User's Guide*, 1999. Release 3.2.

[Tei99] David C. Teigland. The pool driver: A volume driver for sans. http:/www.sistina.com, 1999.

[TM99] David Teigland and Heinz Mauelshagen. Volume managers in linux. http:/www.sistina.com, 1999.

[Ven00] Wietse Venema. File recovery techniques. http:/3w.ddj.com/, 2000.

[VRsTK90] Robbert Van Renesse, Andrew s. Tanenbaum, and M. Frans Kaashoek. The amoeba distributed operating system - a status report. http://www.scs.carleton.ca/ csgs/resources/amoeba_papers.html, 1990.

[VRVST88] Robbert Van Renesse, Hans Van Staveren, and Andrew S. Tanenbaum. The performance of the amoeba operating system. http://www.scs.carleton.ca/ csgs/resources/amoeba_papers.html, 1988.

[WKBC89] Horst F. Wedde, Bodgan Korel, W. G. Brown, and S. Chen. Transparent Distributed Object Management under Completely Decentralized Control. In *Proc. of the 9th International IEEE Conference on Distributed Computing Systems*, pages 335–342, Newport Beach, June 1989. IEEE Computer Society Press.

[WKC$^+$94] Horst F. Wedde, Bogdan Korel, Shengdong Chen, Douglas C. Daniels, Srinivasan Nagaraj, and Babu Santhanam. Transparent Access to Large Files That Are Stored across Sites. In *Readings in Distributed Computing Systems Theory*. IEEE Computer Society Press, 1994.

[WL98] Horst F. Wedde and Mario Lischka. New dimensions in distributed journalism through dragon slayer iii. IEEE, 1998.

[WLS01] Horst Wedde, Mario Lischka, and Oliver Siepmann. Effiziente Scheiboperationen auf verteilte Filekopien für multimediale Anwendungen. Intern Projektgruppenantrag, Lehrstuhl Informatik 3, 2001.

[YV01] Haifeng Yu and Armin Vahdat. The costs and limits of availability for replicated services. http://issg.cs.duke.edu/pubs.html, 2001.

[Zas99] Dr. A. Zaslavsky. Distributed file systems. Http://broncho.ct.monash.edu.au/ azaslavs, 1999.

[Zay91] Edward R. Zayas. Afs-3 programmer's reference, architectural overview. http://consult.cern.ch/service/afs/, 1991.

[Zyn99] Marc Zyngier. Md - multiple devices driver for linux. http://www.penguin.cz/ mhi/fs/Filesystems-HOWTO, 1999.