A DISTRIBUTED FILE SYSTEM FOR DISTRIBUTED CONFERENCING SYSTEM

By

PHILIP S. YEAGER

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2003

ACKNOWLEDGMENTS

I would like to thank Dr. Richard Newman for his help and guidance with this project. I would like to express my gratitude to Dr. Jonathan C.L. Liu and Dr. Beverly Sanders for serving on my committee. I would like to thank Dr. Joseph Wilson for serving as a substitute at my defense. I would also like to thank Vijay Manian and the other DCS group members for their advice and contributions. I thank my parents and friends for their encouragement and support. Finally, I would like to thank Candice Williams for everything she has done. Without these people this work would not have been possible.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

A DISTRIBUTED FILE SYSTEM FOR DISTRIBUTED CONFERENCING SYSTEM

By

Philip S. Yeager

December 2003

Chair: Richard E. Newman
Major Department: Computer and Information Science and Engineering

This thesis presents a Distributed File System (DFS) for the Distributed Conferencing System (DCS). DCS is a system that is developed to allow users to do real-time collaborative work in a distributed environment, while providing the tools and applications to make this process more efficient and easier. In distributed file systems, efficient and accurate management of files is vital. Files can be replicated and stored at physically dispersed locations. They can also be concurrently read and written by numerous people. If these actions are not specially controlled, the resulting files and system can be unpredictable. The work in this thesis is designed to provide an efficient and reliable file system for users to access and modify shared files within DCS.

Contemporary distributed file systems such as the Network File System (NFS), Andrew File System (AFS), Coda, and others do not meet the vast needs of DCS. That is because these file systems are not specifically designed to be an environment where users are constantly reading and writing the same files. They do provide file sharing and limited consistency, but mainly assume that write sharing will be rare, and thus do not

provide the necessary mechanisms and concurrency control to make it efficient and ensure it is handled the best way possible.

DFS is specifically designed to meet the needs of DCS, where write file sharing will be at a maximum. It provides better concurrency control between shared files by using a versioning scheme with immutable files. Each time a file is modified by a user, it is saved as a new version. This ensures that concurrent user writes will not interfere with each other. It also allows users to revert back to older versions of a file if they realize the changes are incorrect. Write conflicts can also be identified within the multiple versions of a file and be corrected. DFS also provides better scalability because clients need not be aware of the number of other clients or the number of servers. Each client communicates with only one server at a time. The client's requests are forwarded to other servers if necessary. Finally, it provides fault tolerance and availability because files are replicated across the multiple servers. If one server that contains the file is down, the file can be retrieved from one of its replicas at another server. This is all done transparent, or hidden, to the users so they can concentrate on the project at hand, and not the inner workings of the system.

DFS is part of a larger DCS version 2. There are other modules that are involved with this project and these are only briefly described in this paper. This thesis focuses on the distributed file system portion of DCS and presents ideas that can be used to improve the methods of shared file access.

CHAPTER 1
INTRODUCTION

1.1 Introduction

This thesis describes a Distributed File System (DFS) designed for the Distributed Conferencing System (DCS) version 2. DFS is designed to manage shared files and provide concurrent user access in DCS. This chapter provides an introduction to DCS and presents why there is a need for such a system. It also explains why there is a need for the distributed file system that is presented in this thesis. The first section gives an overview of the work presented in this thesis. The second section gives useful definitions to terms used throughout this thesis. The third section explains DCS in more detail and its evolutions thus far. The fourth section states the motivation behind this work and the final section gives the organization of the thesis.

1.2 Overview

Frederick P. Brooks Jr. in the *The Mythical Man-Month* [1] states that the amount of work one person can do in twelve months is not equal to the amount of work twelve people can do in one month. This is because when more people are working on a project, more time is spent in other tasks such as communication. Software development over the past few decades has proved this to be true. In today's society, many software projects have become so large that it requires thousands of people to work together, even while separate by oceans. In such a situation, efficient communication and synchronization is important. DCS is designed exactly for this purpose.

DCS is a system that provides users an efficient, reliable, and secure way to work collaboratively on a project. It offers users the infrastructure, tools, and applications to share files in a distributed environment. This is a large need in today's computing world where software development is pushed to a new level. Software is becoming larger and is being demanded at a more rigorous pace each year. In order to meet these demands, people from all over the world must work together on single software projects. DCS is designed to meet these needs and allow users to update and share files in a real-time distributed environment.

File management is an important aspect of any distributed system. Files need to be shared in order to facilitate collaborative work. When a large number of people are sharing files and information in a digital world, many issues must be considered and addressed. First, is the information up-to-date? Digital information can be copied and changed easily. The issue of if the copy or replica possessed is accurate or out-dated is important. Managing the replicas and providing a coherent view of the shared file is called coherency control. Second, is the information consistent? When multiple people share and modify the same file, the issue of if they see other users' modifications, and at what point they see it, is important. Also, when many people are concurrently modifying a single file, the writes can interfere with each other in such a way that the file becomes useless. The management of concurrent file access to avoid inconsistent or erroneous results is called concurrency control. Third, is the information reliable? When dealing with computer systems some failures are inevitable. The issue of whether these failures will corrupt the data and render it inaccessible is a large concern [2]. The Distributed

File System (DFS) presented in this thesis addresses these issues and provides distributed file management for DCS.

1.3 Definitions

This section provides the definitions of terms used throughout this work. Some of these terms are associated with DCS as a whole [3] and others are associated with distributed file systems in general.

- *User*: This is an entity that exists within the system. It uses and performs actions on objects through services provide by the servers. It is usually a human being.
- *Client*: This refers to the process in a client/server relationship that requests services from another process. A user uses a client process to interact use the services provided by the servers.
- *Object*: This is any type of file that exists within the system and is managed by the system.
- *Role*: A role is a named set of capabilities. Users are bound to specific roles within the system. Each role has a set of actions it can take on an object. A user that is bound to a role can perform the set of actions available for that role on a specific object.
- *Conference*: This is fundamental building block of DCS. It is a set of users, with a set of roles, with a set of objects, and a set of applications. Typically a conference has one specific purpose or goal and the members of the conference work together using the objects and applications to achieve this goal. A conference can also have one or more sub-conferences that are used to accomplish a specific aspect of the larger conference's goal.
- *Vestibule Conference*: This is the default conference that exists when a site is created. All other conferences are sub-conferences of the vestibule conference. Each site has a vestibule conference and it is where users go before they log into a specific conference on that site. Using a default vestibule conference allows sites and conferences to be managed the same.
- *DCS file space*: This is the file space used within DCS. It is managed by the distributed file system presented in this thesis. It is only accessible through the services provided by this module.
- *User file space*: This is the file space that exists on the user's local system. It is separate from the DCS file space.

- *DCS aware application*: This is an application that is designed specifically for DCS and can take advantages of the services provided by DCS.

- *DCS unaware application*: This is any application that is not DCS aware. It is not designed for DCS and does not take advantage of any DCS services.

- *DCS Instance*: This is a collection of sites that contain a collection of conferences.

- *DCS Site*: This is a location where one or more DCS servers are operating. It is also where the information about conferences is stored. DCS sites are composed of a Local Area Network (LAN) and have a shared file system. The communication within a site is assumed to be reliable and highly efficient. There is also the notion of correlated accessibility, which means that the accessibility of the servers and objects on a site are related. If one is inaccessible, there is a great probability that all will be inaccessible.

- *DCS Server*: This is a general term that is used to refer to a server designed for DCS that provides services to connected clients and other servers. There are many DCS servers developed by the different modules that compose DCS. Each server provides a different service and together they make DCS possible. A DCS server runs on a DCS site. There can be multiple DCS servers running on the same DCS site.

- *Stateful servers*: This is a characteristic of distributed file system servers which indicates that the servers store some information about the connected clients. Which files a client has opened or cached is an example of state information that is kept by some distributed file servers [2].

- *Stateless Servers*: This is a characteristic of distributed file system servers. It indicates that the servers do not store any information about the clients and each operation must be self-contained [2].

- *Delta*: This is a file that is used in versioning schemes. It does not contain an entire version. It only contains the changes needed to construct one version from another.

- *Coherent*: This describes the state of an object in which all of its replicas are identical or up-to-date. Managing the replicas and providing a coherent view of the shared file is called coherency control [2].

- *Concurrency Control:* This specifies how a system handles the concurrent execution of transactions or commands. Concurrency control ensures that concurrent user accesses are handled in a consistent manner. Some examples of consistency control are file locking, which uses mutual exclusion, and time stamp ordering, which enforces serialization using unique timestamps [2].

- *Consistent*: There are two definitions of consistent. The first definition is similar to coherent. It is the state of an object in which the same view is presented to the

users. The second definition deals with behavior. An object is consistent if its behavior is predictable. Overall, a system is consistent if there is uniformity when using the system and its behavior is predicable [2].

- *Consistency Semantics*: This describes how distributed file systems allow multiple users to access files. It specifies when changes made by one user to a file become visible to other users or when the file is consistent [4].

- *Unix Semantics*: This is a particular type of consistency semantics that is followed by the Unix file system. It specifies that writes to one file by a user are immediately visible to other users that also have this file open [2].

- *Session Semantics*: This is another type of consistency semantics. A session is defined as the time between an open and close of a file. It specifies that writes to a file by one user are performed on that user's working copy and not immediately visible to other users with the file open. It is only when the file is closed that the modifications are made permanent and are visible to other users who open the file later [2].

- *Immutable-Shared File Semantics*: This is another type of consistency semantics. It specifies that once a shared file is created and written it becomes read-only. All new writes are performed on a new version [4].

- *Transparency*: This is a goal motivated by the desire to hide all irrelevant system-dependent details from users [2].

- *Location Transparency*: This is a property exhibited by some distributed file systems. It means that file names used by the clients do not contain information about their physical location [2]. NFS is location transparent.

- *Location Independency*: This is also a property exhibited by some distributed file systems. It means that files can be moved from one location to another without changing their names [2]. AFS is location independent.

- *Uniform Name Space*: This means that each client in the distributed file system uses the same path or file name to access a file.

1.4 The Distributed Conferencing System (DCS)

DCS is a research project under the supervision of Dr. Richard E. Newman at the University of Florida. It is designed to provide users a way to work cooperatively in real-time in a distributed environment. DCS is composed of conferences or groups that are typically created for a specific purpose or goal. Figure 1-1 illustrates the concept of

DCS. Users can connect to these conferences and work cooperatively with each other while using the resources provided by the DCS servers. For example, a user connected to Conf 1 on Site 1 can interact with the users and files available in Conf 1 on sites 3 and 4.

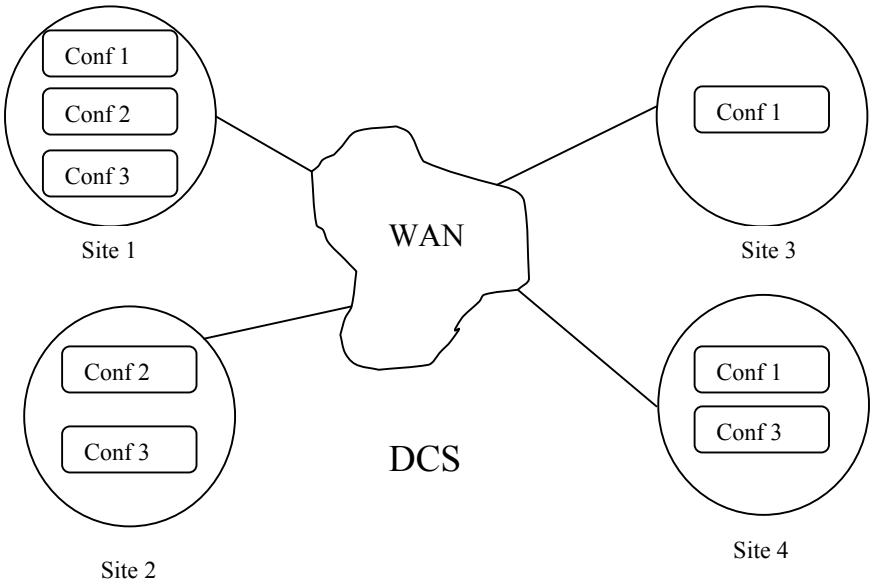

Figure 1-1. An illustration of DCS

There have been two versions of DCS thus far. DCS version 1 was developed in the late 1980s and early 1990s. Its primary goal is to provide users with a set of basic applications and services that enable them to work together and communicate to accomplish a specific task. It provides users with basic applications to co-edit a shared file and communicate with each other. One of the drawbacks of version 1 is that the work a conference could do depends heavily on the applications available. Also, only the users who are bound to a specific role could vote on decisions [5]. Finally, DCS version 1 relies only on UNIX permissions for access control and security [6].

The next version, DCS version 2, started in the early to middle 1990s and is continuing today. It supports a larger variety of voting mechanisms and decision support

services. It also provides notification services for better communication, database services for information storage, and control services to control the cooperating group. The modules that compose DCS version 2 and a brief description of their functions are described below [3].

- *Access Control Services (ACS)*: The ACS provides access control for the objects within DCS. It extends the traditional Access Control Matrix (ACM) to include a decision pointer that points to a decision template in the DSS. This template contains certain conditions that must be processed in order to arrive at a decision. These conditions could be a vote among a set of users or simply determining if this user with this role can access this object.

- *Conference Control Services (CCS)*: The CCS controls the activities of a conference and manages the applications. The CCS starts and initializes the other modules and provides users a way to interact with other services. It provides a Graphical User Interface (GUI) for clients to log into a conference. It also provides operations such as creating a conference, splitting a conference, merging a conference, creating a site, splitting a DCS instance, merging a DCS instance, importing applications, exporting applications, among others.

- *Database Services (DBS)*: The DBS maintains all the tables in DCS. Most of these tables are replicated and kept updated through group multicasting using a causal order protocol.

- *Decision Support Services (DSS)*: The DSS maintains all the decision templates for that conference. It provides mechanisms to add new templates, modify templates, and execute the conditions on the template. For example if a template requires a vote among a set of users, the DSS will contact the users, gather their votes, and return the result back to the calling party.

- *Distributed File System (DFS)*: The DFS module is presented in this thesis. It provides file management for the files within DCS. It also provides users a way to access and modify files. It provides concurrency control using a versioning scheme with immutable shared files. It also provides fault tolerance by providing and managing file replication.

- *Notification Services (NTF)*: The NTF provides users with asynchronous event notification. A user can register to be notified when a specific event occurs. The NTF maintains a global and local database where it stores the information on what users to notify when an event occurs.

- *Secure Communication Services (SCS)*: The SCS provides DCS modules and users a way to communicate securely and reliably with one another. It authenticates

users, uses Public Key Cryptosystem (PKC) to establish session keys, and also maintains these keys.

Some of the above modules are complete while others are still in development. This thesis focuses on the DFS module and presents it development and implementation. However, because of technological advancement, new ideas, and experience these modules are all subject to change in DCS version 3.

It is also important to distinguish the terms used to identify the DFS module in this thesis and generic distributed file systems. In this thesis the term DFS will be used to identify the module from this thesis, not distributed file systems in general. When general distributed file systems are discussed, the term will be written out in its entirety.

1.5 Motivation and Objectives

The need for an efficient and reliable method to manage files within DCS is great. This is an important aspect of all distributed systems. These files may contain extremely important material that could cost a user or organization valuable time and resources if damaged or lost. DCS is designed so users can share and access files in real-time. These users could be in the same room or half way around the globe. The more users that can access a file, the greater the chances are that something undesirable can happen. So in order for DCS to accomplish its goal of providing a way of users to work cooperatively in a distributed environment, it must have a file system that provides efficient, reliable, and controlled file management. That is the goal of the work presented in this thesis.

Traditional distributed file systems, some of which are described in chapter 2, do not adequately meet the needs of DCS. In DCS, in order for users to cooperate, they must share files. Also, for DCS to be as effective as possible, users should even be able access and write shared files concurrently. Traditional distributed file systems are not

designed to be an environment where write accessible file sharing will occur extensively. They assume that most sharing will occur between read-only files. Thus, they provide only limited support for write sharing. They do not provide proper concurrency control to ensure that users' writes are handled the best way possible. They also do not protect against users' writes from interfering or conflicting with each other.

Some traditional distributed file systems recommend using file locks to ensure that the file is protected or verbal communication to ensure that two users do not modify the same portion of a file. However, this reduces the concurrency of the system, which can hinder the progress of a project. This is not acceptable for DCS, which is intended to aid users working together. The need for such system as DCS is imminent and the DFS module presented in this thesis is a large part of what will make it successful.

The overall objectives of DFS are to:

1. Develop a distribute file system layer that implements a shared and uniform name space within DCS,
2. Develop an interface for users and other DCS modules to efficiently interact with the shared files in DCS space,
3. Provide the necessary concurrency control that allows user's to safely and efficiently modify files,
4. Provide reliability and high availability for files within the DCS space,
5. Make the above as transparent as possible to the user.

The following chapters in this thesis describe how these objectives are achieved.

1.6 Organization of Thesis

The first and current chapter introduces the concepts and gives the motivations behind this work. The second chapter of this thesis reviews the past and present work in distributed file systems. The third chapter reviews the past and present work in

versioning control systems that provide protection for shared write accesible files. The fourth chapter presents the requirements of the DFS module of DCS. The fifth chapter presents the design issues and decisions. The sixth chapter discusses the implementation of the DFS module. The seventh and final chapter concludes the thesis, evaluates the work completed, and discusses any future work or ideas discovered while working on this project that were not pursued.

CHAPTER 2
DISTRIBUTED FILE SYSTEMS

2.1 Introduction

This chapter reviews the past and present work in distributed file systems. A distributed file system is a file system that provides users a centralized view even though the actual files are stored at physically dispersed sites [2]. It also allows users to share files and storage. The next sections of this chapter examine three distributed file systems: the Network File System (NFS), the Andrew File System (AFS), and Coda. The main characteristics of each are discussed within each section. Also, a section is included to briefly mention other distributed file systems that are not widespread or still experimental. The next section discusses a system called Redundant Array of Inexpensive Disks (RAID). This system is not a distributed file system. However, it is related to this work because it can enhance the performance and reliability of distributed file systems. The next section briefly discusses the needs of the Distributed Conferencing System (DCS) and why the related work discussed in this chapter does not meet these needs. Finally, this chapter concludes with a section that highlights and summarizes the main details discussed.

2.2 The Network File System

2.2.1 Introduction

NFS was first introduced by Sun Microsystems in 1985 and has since become the world's most popular distributed file system. It is first important to note that there are several different implementations and versions of NFS. In this work, the main items

discussed do not differ between the versions unless otherwise indicated. NFS version 3 [7] and version 4 [8] are described by Brian Pawlowski et al. Abraham Silberschatz et al. [4] and Mahadev Satyanarayanan [9] also give general details of NFS. The main implementations in operation are Solaris and Linux.

2.2.2 Basic Architecture

NFS follows the client/server paradigm to allow multiple distributed clients shared access to files. Servers export file systems or subdirectories to be mounted by clients. However there is no real distinction made between clients and servers. Machines can be both client and servers to other machines. However, each machine must be aware of the servers from where it is importing and the clients to where it is exporting. The clients are aware of each individual server and servers must be aware of clients. The servers also define a set of Remote Procedure Calls (RPC) that allow the clients remote file access. The RPCs include searching for a file, reading directories, changing links and directories, access file attributes, and reading and writing files. All client system calls go through a Virtual File System (VFS) layer that sits on top of the client's local file system. The VFS intercepts calls on the client that are directed to files in the mounted directory. These calls are then sent to the NFS client and then to the appropriate NFS server. The servers are for the most part stateless servers. NFS version 4, however, introduces some state to the servers by handling file locking.

2.2.3 Naming and Transparency

NFS is location transparent but not location independent. Each server can choose to export its entire file system or only a subtree. The server must also indicate to where it is exporting and with what permissions. Each client that wants to access the shared files must mount the files from the server to a directory on their local machine. This can be

done in a different location on each client's directory tree. Cascading mounts can also occur in most implementations. This is where directories mount onto already mounted directories. This can lead to a highly unstructured hierarchy. Figure 2-1 shows two clients who have mounted shared files from the same two servers onto a different location in their directory tree. Client 1 has dir1 from Server 1 mounted at /usr/local/dir1 and has dir2 from Server 2 mounted at /usr/dir2. Client 2 has dir1 from Server 1 mounted at /usr/dir1 and has dir2 from Server 2 mounted at /usr/local/dir2. For clarity dir1 from Server 1 is represented by a dashed line and dir2 from Server 2 is represented by a dotted line. Client 1 and Client 2 see two different views of the same files. This does not yield a uniform name space as in AFS, which is discussed later. Also, once the files are mounted they cannot be moved to a different server. This would cause the path names to become invalid on the client machines. This characteristic does not allow the NFS servers to be load balanced.

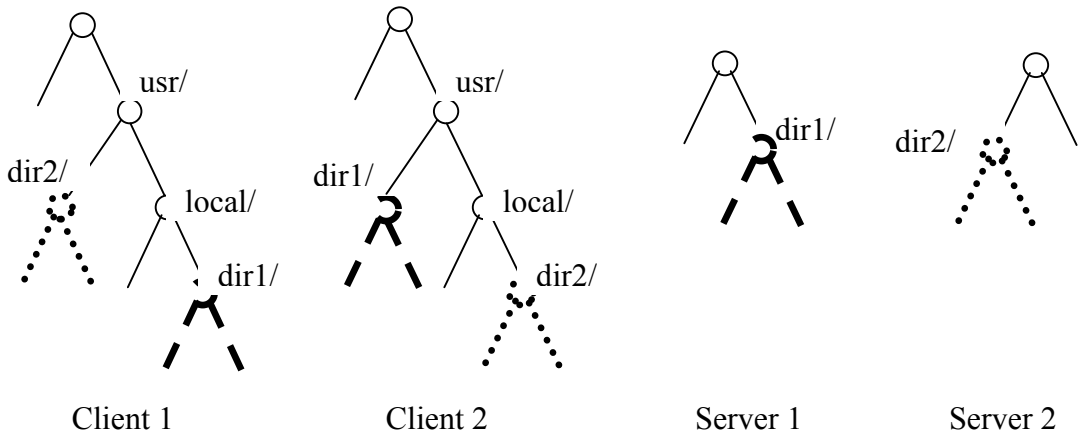

Figure 2-1. An example of two NFS clients that have the same shared files from two servers mounted at different points in their local file system

2.2.4 Availability and Replication

There is no standard replication protocol in most implementations of NFS. However, NFS version 4 does provide very limited file system migration and read-only replication. If the file system is moved to a new server the clients are notified by a special error code. The client can then access the new location by looking at a field in the file's attributes that can contain the other locations. This attribute can also contain the other locations of a read-only file system. However, in order for an NFS client to access the files on another server, it must officially switch to that server. The new server then handles all requests to the mounted files. NFS version 4 does not address atomic server-to-server file system migration or replication coherency. However, since migration and replication are only available in version 4, when a server is unavailable in other implementations, its files will also be unavailable. Also, since cascading mounts can occur, if one server goes down, directories from other still active servers can also become unavailable. The only possibility is to copy entire directories to other servers and explicitly mount them on the client machines.

2.2.5 Caching and Consistency Semantics

NFS clients cache all file attributes and only small blocks of files from the servers. When a user opens a shared file and the file is not in cache, the kernel contacts the server where the file is located and caches the file's attributes and file blocks. NFS uses read-ahead caching so the requested file blocks and a few blocks after are read into the cache. The file attributes cache is discarded after 60 seconds. On subsequent opens to a file, the kernel checks to see if the cached file attributes are up to date. It contacts the server to determine whether to fetch or revalidate the cached attributes. The cached file blocks are only used if the kernel returns that the cached attributes are correct and need to be made

valid again on the client machine. This procedure of continuously handing attribute validation requests places a heavy burden on the file servers. However, it does provide for one important characteristic about NFS. It allows client workstations to be completely diskless. All calls can be sent to the server and only small file-blocks cached in the client workstation's memory.

NFS does not follow any one type of consistency semantics. NFS uses a delayed-write technique where the cache is scanned at regular intervals. Modified blocks are flushed to the server. Once a block is flushed the write is not considered complete until it reaches the server's disk. The delayed write is still completed even if other users have the file open concurrently. However, it cannot be known how long it will take these changes to be visible to other users with the same file open or if they will be visible at all. NFS also does not provide proper concurrency control to ensure that users' writes do not interfere with each other. However, the concurrency control and consistency problem can be eliminated if only binary and read-only files are shared.

2.2.6 Comments

There are several key things to note about NFS. First, it does not provide a uniform name space. Second, all implementations, except for version 4, do not provide any standard file replication. Version 4 only provides rugged support for read-only replication. Third, it does not guarantee consistency because does not specify when user's will see the changes made by other users. Finally, it does not provide adequate concurrency control because it does not protect concurrent user writes from interfering with each other. However, despite these drawbacks, NFS is the most common distributed file system. This is mostly because it is very well implemented, performs well, and is reasonably secure.

2.3 The Andrew File System (AFS)

2.3.1 Introduction

The design of AFS began in 1983 at Carnegie Mellon University (CMU). It was later developed as a commercial product by the Transarc Corporation, who was purchased by IBM. Details of AFS are given by the work of Silberschatz et al. [4], Satyanarayanan [9] and [10], and Satyanarayanan and Mirjana Spasojevic [11] and [12].

AFS is deployed in several hundred organizations and has become the principle file system software at major universities such as CMU, the Massachusetts Institute of Technology (MIT), the University of Michigan, and others. It is continuing to gain more support worldwide. Its original goal is to be a state-of-the-art computing facility for education and research where users can collaborate and share data. The ability to scale well and be secure are fundamental principles in its design and have helped it evolve into a very powerful distributed file system. Recently, AFS became an open source project called OpenAFS.

2.3.2 Basic Architecture

Like most distributed file systems, AFS uses the client/server computing model. Unlike NFS, AFS makes a clear distinction between server machines and client machines. Servers are few and trustworthy while clients are numerous and untrustworthy. In client machines or workstations the local file space and shared file space are kept separate. The shared file space is located under a directory /afs . The shared space under this directory is handled by a collection of dedicated file servers on a Local Area Network (LAN) called Vice. The Vice servers also connect to other Vice servers over a Wide Area Network (WAN). On each client machine there is a process called Venus that controls the shared file access between the client and Vice. Vice and

Venus give each user access to the shared name space as if it was present on their local machine.

AFS divides the disk partitions on the servers into units called volumes. Volumes are small and consist of a set of related directories and files. They are formed similar to the way a mount works in NFS. Volumes are also a key abstraction that allows AFS to be extremely flexible and efficient. Volumes can be replicated and moved without the knowledge of the user. This allows servers to be load balanced which is not possible in NFS.

2.3.3 Naming and Transparency

Unlike NFS, AFS illustrates both location transparency and location independency. It provides a uniform name space and to the users' point of view there appears as if there is one virtual resource. Figure 2-2 illustrates this idea. Client 1 and Client 2 both see the same file system even though the directories dir1 and dir2 are on different servers. The file names do not reveal the physical location of the files and they can be moved without changing their names. Volumes can be moved from one server to another completely transparent to the user. This is different than NFS, which only exhibits location transparency. This is also advantageous because it gives each user a single mount point, which means they can all be configured the same way. This is very important for large organizations with many clients.

2.3.4 Availability and Replication

AFS offers another improvement over NFS in that it provides support for entire read-only volumes to be replicated. The basic design of AFS makes this very straightforward. Since AFS is location independent, volumes can exist on any server and the same view given to the clients. This allows the volumes to be replicated and placed

on different servers. If Vice determines that the server with the requested volume is down, it can direct the request to one of the replicas. This is significant in two ways. First, it makes AFS more fault tolerant. This is because there is a greater chance the desired volume will be available even if one server is inaccessible. Second, it improves performance. Replication allows requests to be directed to the closest copy, which reduces network delay. There also is not any need for replication coherency control because only read-only volumes are replicated.

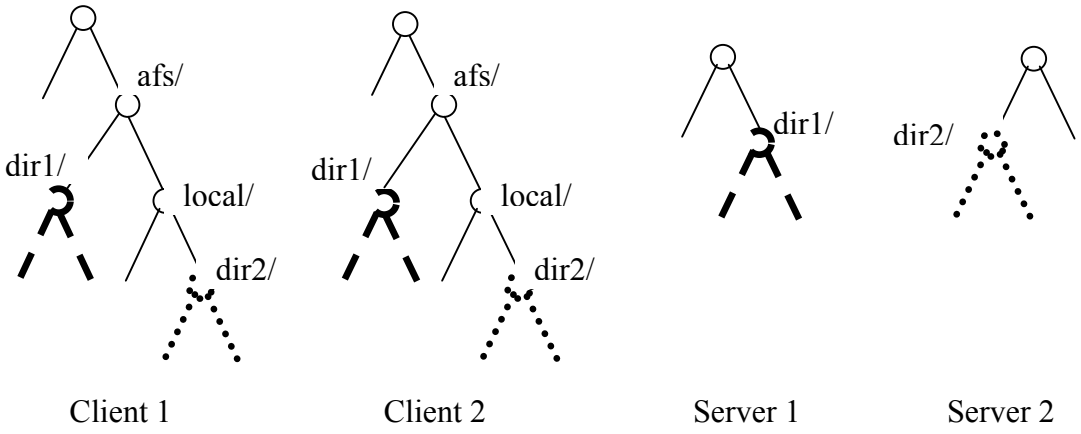

Figure 2-2. An illustration of the uniform name space that is present in AFS

2.3.5 Caching and Consistency Semantics

AFS follows session semantics. When a file is open, chunks of 64k are cached at the client at a time. The AFS servers are stateful in that they record that this client has cached this file. This information is called a callback. The client then performs all writes and reads to their local cached copy. Cached entries are assumed valid unless specified otherwise. Cached entries become invalid when the callback is removed. This is called an optimistic approach. The server removes the callback for all clients with the cached file after another client performs a close after modifying the same file. The clients then

have to open the file and get a new version from the server. This process always guarantees that a file will be consistent after a close is performed. However, it still does not provide adequate concurrency management for DCS purposes. Like NFS, it does not ensure that one user will not overwrite another user's changes. It is possible for two users to close the file simultaneously, yielding inconsistent and unpredictable results. It accepts this fact assuming that writing sharing is rare which makes such conflicts unlikely.

Modifications made to directories and file attributes are handled slightly different then with files. A callback is still used when these are cached at a client. However, the modifications to directories and file attributes are written directly to the server and to the client's cache. This is to ensure security and integrity. The cached data is still used until the callback is broken.

By allowing AFS clients to interact with servers only when a file is opened and closed takes a large burden off the servers. This removes the need for them to respond to constant validation requests from the clients as in NFS. Removing the work load from the servers is a major design goal in AFS.

2.3.6 Comments

There are several appealing characteristics of AFS that should be highlighted. First, AFS provides users with a uniform name space. Second, it allows read-only volumes to be replicated increasing availability, fault tolerance, and performance. Third, it follows session semantics that guarantees shared file consistency when a file is closed. However, AFS does still does not provide adequate concurrency control for DCS. There is still the possibility of conflicts when two users are both writing the same file. This aspect is accepted because it is assumed that write sharing and these conflicts are rare and

easily repairable. All together, these aspects make it easier for AFS to scale to handle numerous clients. This makes AFS a very powerful distributed file system that is gaining popularity.

2.4 Coda

2.4.1 Introduction

Coda is descendent of AFS that is being developed at CMU. Satyanarayanan et al. describe the details of Coda in their work [10] and [13]. Satyanarayanan states that the goal of Coda is to provide maximum availability at the best performance, while maintaining the highest degree of consistency [13]. This is done through two of Coda's main design characteristics: server replication and disconnected operation. This is where Coda differs from other distributed file systems and even its predecessor, AFS. Server replication, disconnected operation, and other aspects of Coda are discussed further in the proceeding sections.

2.4.2 Basic Architecture

The basic architectural design of Coda is very similar to that of AFS. It follows the client/server paradigm with a clear distinction between client and server machines. The shared space on clients is kept in a directory called /coda and is separate from the local space. Also like AFS, servers form clusters called Vice, and clients run cache manager processes called Venus. Servers in Coda also divide disk partitions into units called volumes that can be replicated and moved transparent to the user. A major difference between Coda and AFS is that Coda allows write accessible volumes to also be replicated.

2.4.3 Naming and Transparency

Coda and AFS are identical with regards to naming and transparency. They are both location transparent and location independent. They both also have a uniform name space offering the same file system view to every user.

2.4.4 Availability and Replication

One of the design goals of Coda is constant data availability. This may seem like a lofty goal, but Coda makes many significant advances in this area. Satyanarayanan states that this goal was sparked when the actions of many AFS users with laptops were realized [10]. Users were manually copying relevant files from Vice to their laptops before they disconnected from AFS and left for the weekend. While they were disconnected they would modify the files on their local machines. Then on Monday the users would reconnect to Vice and copy the files back. This led to the idea that an intentional disconnection from Vice and an unintentional disconnection, resulting from the failure of one or more servers, can be viewed and handled the same way. Coda is designed so that this is done automatically and transparent to the user. When the client is connected the opened files are cached on their machine. If and when the client becomes disconnected, a time out or server failure does not occur. The user can simply continue to work on their local cached copies, possibly unaware that he or she is no longer connected. Their changes to local files are stored in a log and when he or she reconnects to the server, these changes are implemented into the actual shared files. This aspect called disconnected operation allows Coda to support mobile clients and provide users with excellent availability even when a server or the network local to a client is down.

Coda and AFS are also similar with regards to server replication. However, Coda allows both read-only and write-accessible volumes to be replicated, while AFS only

supports the replication of read-only volumes. This is accomplished by maintaining a list called a volume storage group (VSG). This list contains the set of servers where a replicated volume resides. In the client on Coda, Venus also keeps a subset of the VSG that is currently accessible from which the client has cached data. This list is called the accessible volume storage group (AVSG). Servers are removed and added depending on their availability. Coda uses a read-one-write-all approach to keep these replicas consistent. Data is read from one server, but after a modified file is closed, it is sent to all servers stored in the AVSG. This increases the chances that every server where a volume is replicated will have the most recent data. Putting the responsibility on the client to make sure that every replica has these changes continues the AFS design goal of unloading the work off the servers.

2.4.5 Caching and Consistency Semantics

Coda, unlike AFS, caches entire files and directories. AFS only caches 64k chunks of a file at a time. When a cache miss occurs in Coda, Venus obtains the data from the preferred server in the AVSG. This can either be chosen at random or by some optimization criteria such as physical location. As mentioned in the previous section, the data is only obtained from the preferred server. However, this server does contact the other servers in the VSG to make sure that that it has the latest copy of the data. Like AFS, a callback is established with the preferred server once the data is cached. The cached copy of a file is always used if it is present and the callback is unbroken.

Coda, like AFS, also follows session semantics. Actually, Coda and AFS will perform exactly the same when no network failures are present. The difference comes in the presence of network failures when Coda still strives for maximum availability. In AFS, an open or close would fail if the server is inaccessible. In Coda, however, an open

fails only if there is a cache miss while the server is inaccessible. An open and close will both fail if a conflict is detected. Conflicts can arise only upon reconnection after disconnected operation. This is when the modification log of the file used in cache is sent to the server. These conflicts need human intervention to resolve. Like AFS, Coda does not provide concurrency control to ensure that one user will not overwrite or another user's modifications during normal operation. Again these types of conflicts are assumed to be rare and easily resolved.

2.4.6 Comments

There are several unique aspects of Coda that should be highlighted. Even though Coda inherits many of its characteristics from AFS, it is unique in the way it handles availability and server replication. Coda strives for maximum availability, so it caches entire files and allows users to work on these files even when the servers are unavailable. Coda also allows write accessible volumes to be replicated. The responsibility of keeping these replicas consistent is left up to the client and not the server. Coda, like NFS and AFS, does not provide adequate concurrency control for concurrent write sharing because it also assumes that this will be rare and conflicts will be easily correctable. Despite this drawback, the other aspects of Coda make it a very unique distributed file system that is likely to gain popularity.

2.5 Other Distributed File Systems

2.5.1 Introduction

There are many other distributed file systems that have been developed or are still in production. For space purposes these all cannot be discussed in detail. However, the sections below give a brief description of a few distributed file systems that have important characteristics that should be noted.

2.5.2 Sprite Network File System

Sprite is a distributed operating system developed at the University of California at Berkley described by John Ousterhout et al. [14]. It consists of a distributed file system and other distributed facilities. The Sprite file system has several important characteristics. First, it provides users with a uniform name space in that there is a single file hierarchy that is accessible to all workstations. Second, it provides complete transparency by being location transparent and independent. Third, it provides exact Unix semantics. A change made by one user is immediately seen by others. In doing so, it guarantees consistency at the level of each read and write by disabling caching of a file when one or more users have the file open for writing. This brings the writing sharing of files to the same level as single time-sharing systems. The actual order of the reads and writes is still not synchronized and should be done with locking. This is an important aspect of the Sprite file system. It effectively brings the sharing level of a time-sharing system to that of a distributed system.

2.5.3 The Elephant File System

The design and prototype implementation of the Elephant file system is described by Douglas Santry et al. [15]. The main goal of Elephant is to allow users to easily roll back a file system, directory, or even a file to obtain a previous version. Elephant automatically retains all important versions of a user's files and allows all user operations to be reversible. It does this by providing three policies: Keep One, Keep All, and Keep Landmarks. Keep One is equivalent to standard file systems where no versioning information or log is maintained. This is useful for files in which a complete history is unimportant. Keep All retains all versions of a file. This is done with a copy-on-write technique in which a new version of a file block is created each time it is written. Keep

Landmarks designates certain versions as landmarks that should always be retained. Other versions that are not landmarks can be deleted. There are two ways to designate a file as a landmark version. The user can either do it explicitly or the user can choose a heuristic. An example of a heuristic might be that every version generated within the last week is a landmark, while of versions that are over a week old, only the versions generated a week apart are landmarks. Elephant also permits concurrent sharing of a file by performing a copy-on-write only on the first open and last close. Logs are also maintained to allow the user to specify when to role back the system for a file or group of files. These policies allow Elephant to be a very versatile file system.

2.5.4 xFS

xFS is a prototype system being developed at the University of California at Berkley that is designed to be implemented over Wide Area Networks (WAN). xFS is described by the work of Randolph Wang and Thomas Anderson [16]. The xFS file system has several key features. First, hosts are organized into a hierarchical structure where requests are always handled by the local cluster if possible to avoid network delay. This gives xFS the potential to scale to a very large number of clients over a WAN. Second, xFS uses a invalidation-based write back cache consistency protocol designed to minimize communication. In this protocol, the clients with local storage space can store private data indefinitely. This allows clients to operate without the constant involvement from the server providing for more availability and disconnected operation as in Coda. xFS maintains cache consistency by using a technique designed for multiprocessor computers. It employs a concept where a single user can have write ownership while multiple users can have read ownership. If many writers are sharing the file, no one has ownership. The cached data is only written to the server on a demand basis when other

users are granted ownership or ownership is revoked. This allows writes to occur across the network only when data is being shared and not when there is just a possibility of data being shared. Third, xFS uses the naming structure to reduce the state needed to preserve cache consistency. State information is stored in a hierarchical basis where top level servers keep track of lower level servers and lower level servers only keep track of the information of the clients they serve. These characteristics are intended to make xFS a file system that outperforms other file systems in a WAN.

2.5.5 Comments

The Sprite file system is not as wide spread as other file systems, such as NFS and AFS, mainly due to its performance that suffers in the tradeoff for having guaranteed consistency and exact Unix semantics. Elephant and xFS are still both prototype file systems that are continuingly being implemented and refined. However, these three file systems all have unique characteristics and employ interesting techniques that can be modified and applied to other distributed file systems.

2.6 Redundant Array of Inexpensive Disks

2.6.1 Introduction

RAID was developed in 1987 by Patterson, Gibson and Katz at the University of California at Berkley. RAID is described thoroughly in their paper [17] and the Red Hat Customization Guide [18]. The basic idea is to combine multiple small, inexpensive disk drives into an array of disk drives that gives performance exceeding that of a Single Large Expensive Drive (SLED). Using RAID can enhance speed, increase storage capacity, and provide fault tolerance. This is very useful in distributed file systems.

2.6.2 Description

One concept involved in RAID is data striping. This is done by concatenating multiple drives into one storage unit. Then each drive's storage is divided into stripes that may be as small as one sector (512 bytes) or as large as one megabyte. The strips are then interleaved, round robin, so the combined space is composed alternately of strips from each drive.

Windows NT, Unix, and Netware systems support overlapped disk I/O operations across multiple drives. This allows for the possibility of increasing performance if writes can be mapped to separate drives so they can be done in parallel. In order to maximize throughput for the disk subsystem, the I/O load must be balanced across all the drives so each drive can be kept as busy as possible. In I/O intensive environments, striping the drives in the array with strips large enough so that each record potentially falls entirely on one strip optimizes performance. This allows each drive to work on a different I/O operation maximizing the number of simultaneous I/O operations.

There exist several different options or layers for RAID.

- RAID – 0: This level is not redundant. The data is split across the drives resulting in higher data throughput. This level yields good performance, however, a failure of one of the drives results in data loss.
- RAID – 1: This level provides redundancy by writing data to two or more drives. This level completely mirrors data on every disk. It is faster on reads, but slower on writes compared to a single drive. However, if one drive fails no data is lost. This level can support the failure of every drive except one and still maintain complete data integrity.
- RAID – 4: This level uses parity on a single disk drive to protect data. It is best suited for transaction I/O rather than large file transfers. The dedicated parity disk is a bottleneck.
- RAID – 5: This level distributes parity info evenly among drives, thus eliminating the bottleneck of level 4. This level can only support the failure of one disk at a time. If two disks fail simultaneously, data is lost.

- Linear Mode: This level concatenates the drives to form one larger drive. Data is stored on the drives sequentially and only moves to another drive once one is full. It also offers no redundancy and decreases reliability. If one drive fails, data integrity is lost.

RAID can be implemented in hardware and software. With the increasing speed of today's processors, the performance of the Software RAID can excel that of the Hardware RAID. Software RAID also offers the cheaper of the two solutions. The MD driver in Linux kernel is hardware independent and supports RAID levels 0/1/4/5 + linear mode. However Red Hat Linux does not allow RAID – 4.

2.6.3 Comments

RAID is a very valuable tool that is being implemented into distributed and centralized file systems. It is being used along with NFS to make it more reliable by adding aspects of replication and availability that are not present in its design. It can also be used along with AFS and Coda to improve performance in some areas that suffer to do server replication and other factors of design. RAID is continuing to find its way into the implementation of more systems because of the valuable enhancements it offers.

2.7 Previous Implementation of DCS File Services

In 1996, Ling Li designed and implemented file services for DCS [6]. The goals of his work were to:

1. Organize the provisions of application files distributed in one or more computer administration domains,
2. Identify where an application file may be located and its availability,
3. Control the use of, and access to, application files, according to access control rules,
4. Assist those users wishing to contribute new applications to the community,
5. Control the application and distribution of application files according to their contributor's specification.

Li's work was built upon Sun's NFS and fails to meet all the current needs of DCS version 2 in several areas. First, he provides no distributed file system layer. He assumes that all users of a conference will be at a single site and use the same DCS server. This will allow them to be presented with a single home directory for the conference. However if these uses are at multiple sites then there will be multiple home directories for the conference. This does not provide for group activities or data sharing for the members located at different sites. In DCS version 2 the users of a conference should be presented with a single virtual directory tree. Second, Li's work focuses on application management and only provides import and export of data files to and from the DCS file space.

2.8 The Needs of DCS

This chapter describes several distributed file systems that all have valuable characteristics but don't meet in needs of DCS in several areas. Since DCS is designed to allow users to work cooperatively in a distributed environment, its distributed file system must allow user's to share and access file concurrently. It should provide concurrency control to allow users to concurrently modify the same file without fear of other's writes interfering. It should also provide high availability and reliability so that files can be accessed despite some inevitable network failures.

NFS does not guarantee file consistency. AFS, Coda, Sprint and xFS guarantee file consistency, however do not provide adequate concurrency control. They do not ensure that multiple users' writes do not interfere with each other. One way to ensure that each user's work is protected is to use a versioning scheme where changes made to a file can be kept in separate versions and merged when necessary. Elephant uses a versioning

technique to allow a user to roll back the system or a file to a previous state. However, it does not provide multi-version solutions when multiple users are modifying the same file at the same time. Their modifications would simply go into one version bounded by the first open and last close. Finally, these distributed file systems are not designed to spread across different administrative domains. Administrators at one domain have no control over the actions at other domains.

Chapter 3 discusses several version control systems that successfully implement multi-versioning schemes. These systems and the ones discussed in this chapter are relevant and useful because their techniques can be applied to the DFS module and make DCS an effective environment for collaborative work.

2.9 Summary

Getting a good understanding of how distributed file systems are designed and the issues involved allows the requirements and design of DFS in this thesis to be better formulated. This chapter discusses the main characteristics of distributed file systems using three main examples: NFS, AFS, and Coda.

AFS and Coda, unlike NFS, present all the users with a uniform name space. This is essential in systems that are intended to scale to a large number of clients. It also allows the activities of clients to be easily coordinated because each uses the same path names to reference the same files. Each of the systems discussed also offer a different level of consistency. NFS basically offers little consistency because it assumes that clients will rarely be sharing write accessible files. AFS and Coda only provide session semantics but like the other systems discussed, do not provide concurrency control to protect a user's modifications. Finally, Coda also provides a standard for the replication of write accessible files. AFS only provides for the replication of read-only files, while

most implementations of NFS provide no standard for replication. Replication is important because it allows files to be available to clients. If one server is down the file can still be accessed on other server that contains a replica.

Other file systems are discussed that also have a few interesting characteristics. Sprite provides Unix semantics, Elephant provides multiple user versioning, and xFS provides a unique organization that is intended to boost performance.

From these examples the desirable and undesirable aspects for DFS can be gathered. The next chapter focuses on the requirements of the DFS module and explains how these are not met by the current systems.

CHAPTER 3
VERSIONING CONTROL SYSTEMS

3.1 Introduction

This chapter reviews past and present work in versioning control systems. These systems are relevant and useful in that they aid the implementation and management of files that can be accessed and updated by multiple users, a major characteristic of the Distributed Conferencing System (DCS). The next three sections discuss three major version control systems: Source Code Control System (SCCS), Revisions Control System (RCS), and the Concurrent Versioning System (CVS). The next section briefly presents other version control systems that are either not widespread or still experimental. The following section briefly discusses the needs of DCS and why the related work discussed in this chapter does not meet these needs. Finally, this chapter concludes with a section that highlights and summarizes the main details discussed.

3.2 Source Code Control System (SCCS)

3.2.1 Introduction

SCCS is one of the oldest tools developed for version control. It was designed in the early 1970s and its main goals are the efficient management of memory space when keeping several versions of the same module and to allow the retrieval of any specific version. SCCS is described by Vincenzo Ambriola et al. [19] and briefly by Brian Berliner [20].

3.2.2 Description

Figure 3-1 is an example of a SCCS versioning tree. A versioning tree describes the relation and formulation of a file's versions. Versioning trees can be oriented going from left to right or top to bottom. Branches develop when two different versions are created from the same version. In Figure 3-1, the first box in bold, version 1.0, represents the entire file. The boxes with arrows represent that the only changes needed to get from the previous to this version are stored. SCCS began a technique of storing only the changes or deltas to a version to get to the new version. For example, only version 1.0 in Figure 3-1 has to be saved along with all the deltas to be able to retrieve any version. This is efficient because it does not waste space by storing entire files over again. This technique is still found in most versioning systems used today. A new version is obtained by copying the desired version, making changes to this version, and then saving the modifications or delta. SCCS also automatically saves who made the change, when it was made, and why it was made if the user feels it necessary to give reasons.

There are several downsides to SCCS that should be noted. First, it only allows one user to have a write accessible copy of a file at one time. When a user checks out a file for writing, it locks the file preventing others from checking it out and making modifications. The file is locked until the user decides he or she is done making modifications and checks back in the file. Although this is very safe and simple to implement, it limits the amount of productivity that can occur. Second, there can be only one branch at a given version in the version tree. Figure 3-1 shows a branch at version 1.3. This means that version 1.3 has been modified twice to produce two different versions. One version is 2.1 and the other is 1.4. SCCS does not provide the possibility for another branch or version to start at version 1.3.

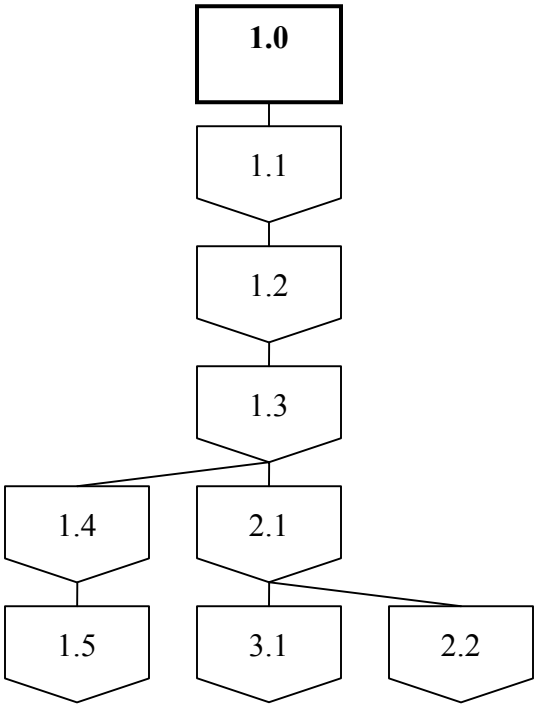

Figure 3-1. An example of an SCCS versioning tree

3.2.3 Comments

SCCS developed the basis for most versioning schemes today. Modifications are made to this scheme in other systems, but the fundamental principles are the same. The Revision Control System (RCS), which is discussed in the next section, is an example of a system that developed from SCCS. Although SCCS was a useful tool for source code management in its time, other more recent version control systems provide significant improvements.

3.3 Revisions Control System (RCS)

3.3.1 Introduction

RCS was designed in the early 1980s and is a descendant of SCCS. RCS is described thoroughly by Walter F. Tichy [21]. The design goal of RCS is to improve on some of the drawbacks of SCCS. RCS accomplishes this goal by improving the structure

of the versioning tree, using forward and reverse deltas, and providing a make interface that allows the user to easily rebuild configurations.

3.3.2 Description

Figure 3-2 shows an example of an RCS versioning tree. RCS improves upon SCCS versioning tree by allowing multiple branches to start from a single version. An example of this can be seen at version 1.1 in Figure 3-2. The first branch yields the new version 1.1.1.1 and the second branch yields the new version 1.1.2.1. The first numbers are the release, 1.1 in this case. The second number is the version in the release, 2 in this case. The last number is the version of the branch, 1 in this case. From the figure it can also be seen that the branches can have subsequent version. The next version after 1.1.2.1 is 1.1.2.2.

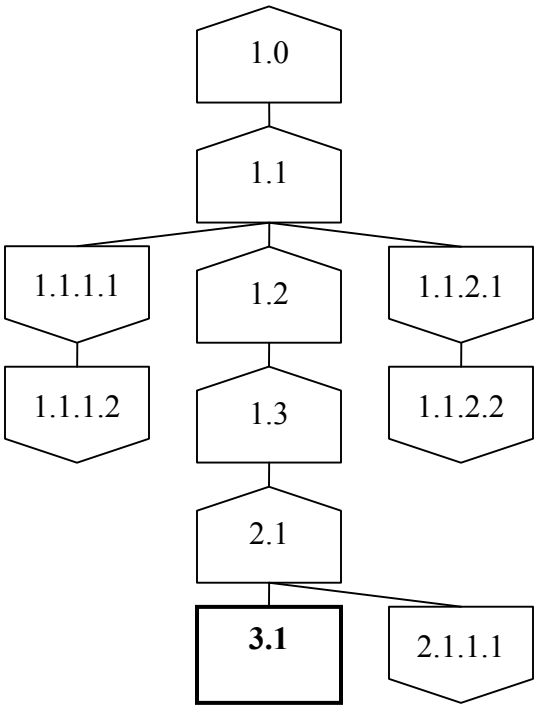

Figure 3-2. An example of the RCS versioning tree

The second improvement RCS made upon SCCS is in the performance of retrieving the latest version. In SCCS, only version 1.0 is saved in its entirety and forward deltas are maintained for the subsequent versions. However, this is inefficient when the latest version of a file is requested, which is the majority of the time. To get the latest version, all the forward deltas must be implemented into the original version. RCS uses both forward and reverse deltas. In RCS, the latest version of the entire file is stored with reverse deltas maintained so previous versions can be retrieved. This can be seen in Figure 3-2. The bold box represents the version of the file stored in its entirety. The boxes with a point facing downwards are forward deltas and the boxes with a point facing upwards are reverse deltas. This allows the latest version to be retrieved quickly. Forward deltas are still used when a branch occurs. This is done so that full versions do not need to be stored at the tip of every branch. From the version where the branch occurs, forward deltas are used to retrieve all versions on that branch. This technique makes RCS both efficient in time and space with regards to retrieving and saving versions.

The third improvement of RCS over SCCS is providing a make interface to the users. This allows users to easily rebuild certain configurations of the software. Since one software project can be spread to multiple files, a configuration is simply the collection of the separate versions that combine together to form a version of the larger project. The make interface provides mechanisms to efficiently compile all the files for a project by automatically checking out files that are needed but not present in the user's working directory. It then deletes the files that were only checked out for compilation purposes when it is finished.

Despite the improvements made to SCCS, RCS still has several drawbacks. First, even though there can be multiple branches from a single version, RCS still employs locks to keep multiple users from edited the same file concurrently. This is done because there is no mechanism to stop one user from overwriting the version made by another user. When users check out files, they must explicitly specify which version they would like to check out. Also, when users check files back in, they must explicitly specify the new version name. For example, the following command sequence below checks out version 1.3 of file example.c and checks it back in as version 1.3.1.

- co –r1.3 example.c
- edit example.c
- ci –r1.3.1 example.c

If two users checked out version 1.3 and check it back in as version 1.3.1, the first user's changes would be incorporated and possibly overwritten by the second user's changes in the same version. To prevent this from happening, RCS uses locks, which can slow the development of the product.

This leads us to the second drawback of RCS. Since users must explicitly name the version they want to check out and the new version to check it back in as, they must be aware of all versions. If a user is careless and not aware that there is already a subsequent version to a file, he or she can overwrite the changes in that version as already mentioned.

3.3.3 Comments

RCS improves upon some of the features of SCCS, but still leaves several drawbacks. Despite these drawbacks, RCS made significant contributions to source code

management and version control. The next system discussed, the Concurrent Versioning System (CVS), uses these contributions and makes further enhancements.

3.4 Concurrent Versioning System (CVS)

3.4.1 Introduction

CVS is developed as a further extension of RCS and is described by Brian Berliner [20]. Its major improvement on RCS is that it allows multiple users to concurrently edit files while providing conflict resolution. Other improvements include a location independent database, detailed modification logging, software release control, and vender source support.

3.4.2 Description

One of the biggest drawbacks of SCCS and RCS is that it requires users to work on files in a serialized manner. They do not allow two or more users to check out and edit a file concurrently. Brian Berliner states that the development pressures of productivity and deadlines of most organizations are too great to only allow one user to edit a file at a time [20]. Thus CVS was developed to more efficiently meet the productivity needs of software development organizations. It allows users to work in parallel on a single file by maintaining the version checked out by each user. It then compares the version checked in with the latest version and tries to merge them if they are different.

Consider the following scenario for example. Say version 1.2 is the latest or head version of a file. User A first checks out version 1.2 first followed by User B. Both are unaware that another user has checked out the same file. Say User A makes changes and checks back in the file. CVS compares this version to the head version and sees that they are both 1.2. It then commits User A's changes as version 1.3. This is now the head version. Now say User B checks back in their modified version. CVS compares this

with the head version, which is now version 1.3. It sees that the version 1.2 checked out is different than the head version 1.3. So it attempts to automatically merge User B's changes with User A's changes. It then reports any conflicts back to User B. CVS leaves it up to the user to resolve the conflicts and resubmit the changes with ready. This process allows multiple users to work concurrently their own copy of the same file, ensuring that their changes will not be lost.

Another improvement in CVS is its location independent module database. This allows users to check out entire directories or files without knowing where they actually exist within the global source distribution. Conflicting names can be resolved by simple name changes. Other improvements made in CVS include the following: tracking and incorporating third party source distributions, modification logging support, tagged releases and dates, and allowing patches to be produced between two software releases.

3.4.3 Comments

CVS is a very useful tool that is used throughout industry today. It provides an efficient and organized way to manage large software projects that include multiple versions. The main aspect of CVS is that it allows multiple users to work concurrently on a file while ensuring that their work is safe and will not be lost. This is a valuable characteristic that aids in the process of software development. CVS can also be used within NFS to allow the sharing of a central repository to multiple remote clients.

3.5 Other Versioning Control Systems

3.5.1 Introduction

There have been several other versioning systems developed. This section presents three distributed versioning systems which allow multiple users in a distributed environment to work on a shared or distributed source code repository.

3.5.2 Distributed RCS (DRCS)

DRCS is built upon RCS to allow clients to connect to remote central repository that stores the files and other information. It is described by the work of Brian O'Donovan and Jane Grimson [22]. There are several important aspects of this system. First, all of the files are still stored in one central location. Second, when files are checked out they are transported to the user's system. Third, all this is done transparent to the user by allowing them to use the same RCS commands.

3.5.3 Distributed CVS (DCVS)

DCVS is built upon CVS and it uses a replicated source code repository. It is described by Brad Miller et al. [23]. In DCVS, each developer has a local copy of the entire repository that he or she uses just like in CVS. However, when changes are ready to be committed, a token must be obtained for each file that is modified. Once the token is obtained, the files can be committed to the local repository in the normal CVS fashion. Once the files are committed, copies of the local repository files are replicated to all machines in the distribution list. This is a list that must be maintained and include all the machines where a local repository for this project is kept.

3.5.4 Distributed Versioning System (DVS)

DVS is a built upon RCS and provides the distribution of files and the collective versioning of files. It is described by the work of Antonio Carzaniga [24]. The key aspect of DVS is that each file can be stored in a different repository located at a different physical location. This can also be done transparent to the user. DVS relies on the Network Unified Configuration Management (NUCM) File System. This file system is specifically developed for the implementation of configuration management systems. It manages the repositories located on the distributed servers.

3.5.5 Comments

The three distributed file systems presented all are extensions of either RCS or CVS. DRCS uses a central repository, DCVS uses repositories distributed on the user's machines, and DVS uses NUCM on multiple servers to manage the repository. While these systems are useful, they don't meet the specific needs of DCS which are briefly discussed in the next section and expanded upon in Chapter 4.

3.6 The Needs of DCS

This chapter presents several versioning control systems that have very useful characteristics, but don't quite meet the specific needs of DCS in several different areas. First, SCCS and RCS only allow one user to access a file at a time. This is not acceptable in distributed environments. CVS, however, does allow multiple users to concurrently access a file. Second, these systems are specifically designed for version control and not file management. Specific commands are needed to check in and check out files and they provide no basic file system interface or management tools. DCS needs to provide users more than mechanisms to check in and out files. It needs to provide users with an entire file system interface that manages files and applications. Third, in order for multiple users to use these systems, they either require the use of a shared file system or their own underlying sharing mechanisms. CVS can be used on NFS, while DRCS, DCVS, and DVS, all use their own underlying mechanisms. For security reasons the objects stored on the DCS servers can only be accessible through DCS client systems running on the client machines. These systems do not support this type of access. Although these systems provide useful tools in version control and source code management, they currently lack certain aspects that are necessary for DCS. What is needed is a system that combines the distributed access characteristics of distributed file systems and the

versioning management characteristics of version control systems. The requirements for the DFS that allow it to meet the needs of DCS are described in detail in Chapter 4.

3.7 Summary

This chapter presents three main version control systems: SCCS, RCS, and CVS, and three less common experimental systems: DRCS, DCVS, and DVS. Version control systems are related to the work presented in this thesis because they provide useful mechanisms for managing files shared by multiple users, which is a major characteristic of DCS. Although these systems are useful tools, they do not meet one or more of the needs of DCS. SCCS and RCS do not provide for multiple user access. SCCS, RCS, and CVS do not provide file management or allow users to access files from outside networks. DRCS, DCVS, and DVS all attempt to extend the common versioning systems to wide area networks, however they still have the limitations of their ancestors. Thus, the development of a system that does meet the needs of DCS is the motivation behind this work.

CHAPTER 4
REQUIREMENTS

4.1 Introduction

This chapter discusses the requirements for the Distributed File System (DFS) within the Distributed Conferencing System (DCS). The intent of this module is to provide the underlying file system for DCS. There are many examples of distributed file systems that have already been developed. A few of these are discussed in Chapter 2. However, these file systems do not meet the specific needs of DCS. A few versioning control systems were also discussed in Chapter 3. These systems provide useful ways to protect users' writes from interfering with each other, however, don't provide any other file management. They require the use of specific commands to "check-in" or "check-out" a file. DFS is designed to be a transparent versioning file system that incorporates the best of both systems. The sections below describe these needs and the requirements for the DFS module. These requirements are developed based on the needs of DCS and the other DCS modules. They are formed through a cyclical process of group discussion and refinement among the DCS members.

4.2 Basic Architecture

4.2.1 Client/Server Architecture

DFS needs to have a client/server architecture. DCS is designed to have dedicated server and client machines. It uses the client/server relationship to allow remote clients to access the services and resources provided by DCS. DFS also needs have dedicated servers that provide the file management and file services to the clients. These servers

can run on the same machines as the other DCS servers developed in the other modules. DFS should also have a client process that runs on the client machine to pass these requests to the servers.

4.2.2 DCS File Space

Since DCS is designed to allow users to work collaboratively together, there needs to be shared DCS file space. The file space is separate from the local file space on each of the client machines. It is managed by the DFS module. Users can join DCS through services provided by the Conference Control Services (CCS) module designed by Ashish Bhalani [25]. When a user connects to a DCS site it joins the default vestibule conference. The user can then request to join a specific conference through the CCS. When a user is connected to DCS, either in a specific conference or the vestibule conference, he or she has access to the DCS file space.

The directories in the DCS file space should be hierarchically structured. The DCS file space should be composed of a virtual DCS directory structure. Each conference has a directory /dcs/conf/<conf-name>. The files for a particular conference reside in that conference's directory. These files do not reside at one particular location. Instead, they are dispersed across the sites of the conference. Each conference can be located at multiple DCS sites. Figure 1-1 in Chapter 1 illustrates the relationship between sites and conferences. The virtual DCS directory structure is composed from the files located at these DCS sites.

4.3 Naming and Transparency

4.3.1 Uniform Name Space

DFS needs to be location transparent and location independent. It should provide users with a uniform name space of the DCS directory structure. In a DCS conference,

DCS files are dispersed across the sites of that conference. Even though the files are distributed across physically separated storage sites, a centralized view of the DCS file space should still be presented to the client. The file names should not represent the site at which the file is located. The DCS clients need to be able to access files from a conference in the same manner regardless the physical location of the files.

NFS is a distributed file system that does not meet these needs. It is location transparent but not location independent. Each NFS server has a set of files that it mounts to a point in the client's local directory structure. If NFS is used within DCS, the files for a particular conference would have to reside on the same server to be mounted in the same conference directory on the user's system. The location of these files in the DCS file space would depend on where the client mounted its directory from the server. Since this is not desired, there should be no mounting that occurs on the client machine.

4.3.2 View Based on Roles

Also, as described in Vijay Manian's work on an Access Control Model for DCS, each client is bound to a specific role that has certain privileges [3]. One of these privileges is viewing the contents of a directory in DCS file space. A goal of DCS is to be able to present each role a certain view of the file system even within a certain directory. This is analogous to the idea of allowing a user to be able to access some files with a directory and not others. Only here there is another level of access that includes seeing the files in the directory. If the role to which a user is bound is not privileged to see certain file names, then when he or she views the contents of a directory, those names would not be present. DFS should aid in presenting a directory view based on the role to the user.

4.4 Availability and Replication

In order to be fault tolerant and offer higher availability to the clients, the DFS module should provide file replication. Files within the DCS space can be replicated at multiple sites. If a site that holds a file is inaccessible, the file can be retrieved from one of its replicas. This is done transparent to users unless the site to which they are connected becomes inaccessible. Then they have to connect to another server at another site manually. DCS is also designed to spread across Wide Area Networks (WANs) and different administrative domains. DFS should allow objects to be available to users across these divisions.

4.5 Caching and Consistency Semantics

Since DCS is designed to be a system where users can collaboratively work on a project, DFS should be designed to provide shared file access in the best way possible. It should be efficient so files should not be locked from other users, which can slow down production. Users should be able to access write to any file concurrently. DFS should also ensure that users' writes do not interfere with each other. To accomplish this, a versioning scheme with immutable files should be used. Entire files can be cached at the user's machine when opened. All subsequent reads and writes are then directed to this cached file. This provides good performance on reads and writes. When a file is closed it can be saved back to the server as a new version. Merging can occur between the versions and conflicts identified. This should all be done transparent to the user. However, in order for a user to see the changes made by other users, he or she must explicitly open the new versions that are created when the other users close the file. This makes it difficult to compare the consistency semantics of DFS with that of other systems. This is discussed further in Chapter 5.

The other distributed file systems discussed do not have these properties. They do not provide adequate concurrency control to ensure each user's modifications are protected. Elephant provides for the transparent multi-versioning of files but not when two users are editing the file at the same time. Versioning systems such as CVS, provide adequate concurrency control, however they do not provide access transparency. The users must explicitly specify that they want to "check-out" or "check-in" a file using the commands provided. They also do not provide full file management.

4.6 File Attributes

There are certain file attributes that need to be kept. The following is a list of file attributes that should be kept for each file within DCS space:

1. Location of the file within DCS directory space,
2. Type of file,
3. Owner of file,
4. Conference to which the file belongs,
5. Size of file,
6. Number of entries in directory if the file is a directory,
7. Last access time,
8. Last time a new version was created,
9. Time file was created,
10. Mode at which the file can be opened,
11. Last version of file created by time,
12. Last version of file lexigraphically or by name,
13. Location of replicas,
14. Number of replicas.

4.7 File System Interface

Along with managing the files in DCS space, DFS should provide an interface for the remote users and other DCS modules to access these files. DCS should provide the commands listed below.

1. Create a file

2. Open a file: Caches file on user's local machine

3. Close a file: Writes back a new version to server

4. Read a file: This is only done between DFS servers or when a file is specified that it cannot be cached

5. Write a file: This is only done between DFS servers. Users always write files as new versions

6. Delete a file: Since multiple versions are kept, this does not necessary permanently free the file's storage

7. Make directory

8. Remove directory

9. Move or rename a file or directory

10. Copy a file or empty directory

11. List contents of a directory

12. Get attributes for a file

4.8 Application Interactions

DFS should allow both DCS aware and DCS unaware applications to interact with the file system. DCS aware applications can simply use the interface provided by the server. DCS unaware applications must be handled a different way. Their system calls must be intercepted and then directed to the DFS Client. This is outside the work presented in this thesis. However, DFS should provide an interface that is specific enough for applications that are DCS aware and unspecific enough for applications that are DCS unaware. This way, DCS unaware applications won't be required to provide unknown information.

4.9 System Requirements

The system requirement for this module is Linux OS. This module is programmed in Java 2 version 1.4.1. This package or greater should also be installed on the system.

4.10 Summary

This chapter presents the requirements for the DFS module described in this thesis. These requirements specify the basic architecture as well as common distributed file system characteristics such as naming and transparency, replication and availability, and caching and consistency semantics. Other requirements outside these characteristics are also presented. DFS should provide a uniform name space to clients, concurrency control by using a versioning scheme with immutable files, and file replication to increase availability. DFS should also cache files to improve performance. DFS is designed to work with DCS version 2 and provide file management and a file interface for users. These requirements are developed to make DFS, and thus DCS, as effective as possible.

CHAPTER 5
DESIGN

5.1 Introduction

This chapter discusses the design of the Distributed File System (DFS) within the Distributed Conferencing System (DCS). The previous chapter discusses the requirements for DFS and this chapter explains how DFS will be designed to meet these requirements. Within each section the design details and options, along with their advantages and disadvantages, are discussed. Also, the final design decision that will be implemented is presented with reasons why it is chosen. It is always important to document all the design options considered even if they were not implemented into the final design. This allows the reader and future designers to know that this possibility was considered and why it was not chosen. The chapter concludes with a summary that highlights the main points discussed.

5.2 Basic Architecture

5.2.1 DFS Client and Server

DFS is also designed using a client/server relationship. The DFS client runs on the user's machine and sends all the file system requests to the DFS servers. It provides the file system interface for the clients. There are several options that need to be considered relating to the design of the DFS servers. These are discussed in the section below. It is followed by a section that describes the decision implemented in DFS.

5.2.2 Design Options

5.2.2.1 One server for entire site

One possibility is to have one DFS server for an entire site. This server would handle all the file system requests for each conference on the site and each client within the conference. The main advantage of this is simplicity. Everything would be handled in one place and the only communication would be between clients and this one server and between servers at different sites. The disadvantage of this is that the server would have to keep track of all conference policies and roles and use the appropriate one to handle each client request.

5.2.2.2 One server for each conference

Another possibility is to have a separate server for each conference. This includes a separate server for the entire site (which is simply the vestibule conference). Each server would only handle the requests of their particular conference and simply forward inter-conference requests to the appropriate server.

The main advantage of this design is that it allows for the possibility for DFS servers to be placed on separate machines within the site. The site would not have to have one central server that handled every request. This places a large burden on one machine that could degrade performance. If separate servers are used the burden and resources, such as space and CPU cycles, are separated among several separate machines or even simply separate processors of the same machine. Clients could connect to the conference server to which they belonged. Of course, this only provides an improvement if the servers are actually placed on separate machines or separate processors. If this is not an option at a site, all the servers could still be run on a single machine or single processor.

The disadvantages of this are complexity and the overhead generated from the testing if this request should be handled by another conference, the lookup of the appropriate server location, and the communication. Having separate servers makes it necessary for the storage and lookup of where these servers are located. This needs to be done on every inter-conference request of the client. This adds complexity to the code. Also, since these requests will have to be sent to another server to be handled and the result has to be sent back to the first server and then back to the client, communication overhead will increase. However, if these servers do happen to reside on the same machine, the overhead will be dominated by the lookup of server. The complexity is somewhat, but not completely, balanced by the fact that a single server only has to handle the requests of one conference with a single set of policies and roles. However, file requests that involve two locations (copy or move), in which one location is handled by one server and another handled by a separate server, become significantly more complicated to handle.

5.2.3 Design Decisions

DFS is implemented using separate servers for each conference. This is done mainly because it is a more flexible design. It allows the possibility of the servers to be physically separate or not. It is also felt that typically most operations from clients will be intra-conference operations. These operations are handled by the single server dedicated to this conference. This allows the server to easily manage the set of policies, roles, files, and other information that is strictly for this conference. Some complexity and overhead is added in the inter-conference operations, but this will likely be rare and tolerable.

5.3 File Management

5.3.1 Linux File System

The Linux File System is used for all underlying file management and disk operations at the clients and servers. This eliminates the need to implement the general file system details that are already successfully and efficiently implemented. This allows more attention to be paid to the issues specific to DCS.

5.3.2 Uniform Name Space

DFS provides a distributed file system layer over the underlying Linux file systems. A single DCS conference can exist on many sites. Each site in the conference maintains some of the files of that conference. A server at a site will contain some of the conference files and not contain conference files that exist at other sites. One of the main characteristics of a DCS conference is that the actual location of the files is transparent to the user. The user should not be aware of which files are kept at which site. A uniform name space and virtual DCS directory structure should be presented to the clients even though the files are physically dispersed. The next section discusses two options considered.

5.3.3 Design Options

5.3.3.1 Symbolic links

One option for presenting the clients with a uniform name space is to use a technique analogous to symbolic links on sites where certain files do not exist. This is best illustrated by two figures. Figure 5-1 illustrates an example of DCS sites and where files for a particular conference actually exist.

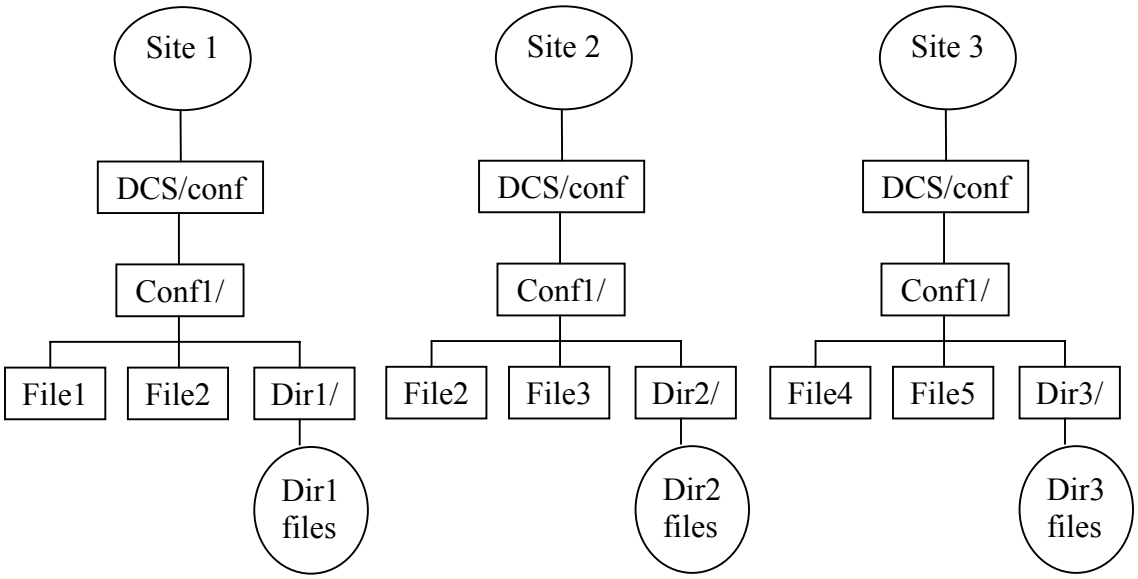

Figure 5-1. An example of the actual location of files within a DCS conference

The conference name is Conf1 and a directory for this conference is inside the general DCS/conf directory. Each site contains a DCS/conf/ directory and a directory of the conference name. Inside the conference Conf1 directory, the actual files that exist on the site are shown.

To keep the location of the files transparent, symbolic links can be used for files that do not exist at a site. A symbolic link is when there is an actual object that exists which directs the requests to the actual location of the desired object. Figure 5-2 illustrates the virtual directory structure of Conf1 for Site 1. The objects in bold represent the files that do not actually exist on Site 1 as can be seen from Figure 5-1. These files are now represented with symbolic links. These bold files now actually exist on Site 1. However, the contents of the file is not the contents of the actual file, it is the actual location of the file. A file attribute indicates that this file is a symbolic link. When a request is made for a given file, the DCS server first checks to see if the file is a symbolic

link. If it is, the server opens the file, gets the actual location of the file, and sends the request to the appropriate server. The new server executes the request, sends the result back the server, which then sends the result back to the client. This is all done transparent to the user. Note that the Dir2 and Dir3 directories are not in bold. This is to indicate that the site must still contain the entire directory structure of each conference, even if the directory contents are all symbolic links.

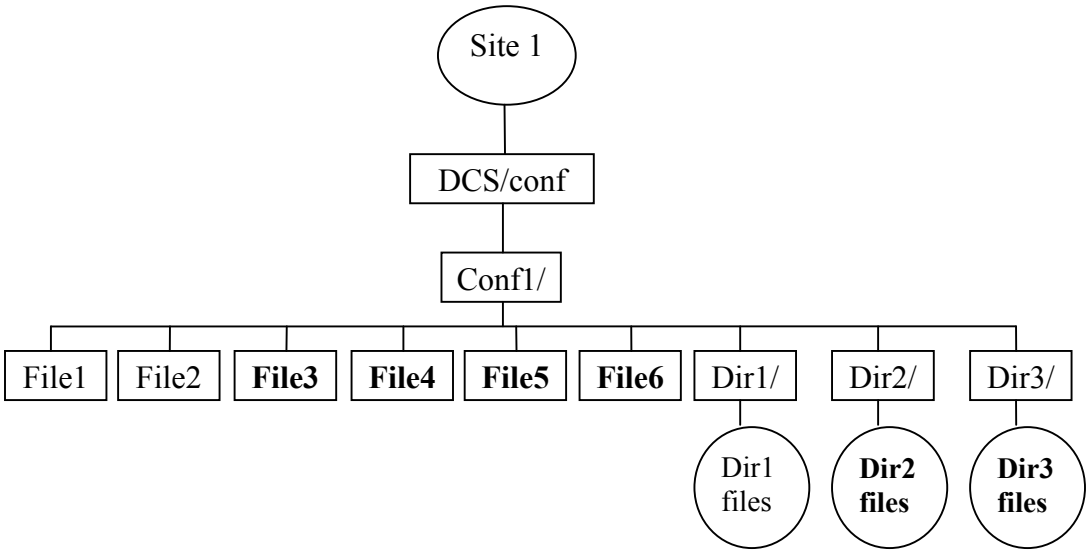

Figure 5-2. Actual virtual directory of conference Conf1 with symbolic links in bold

5.3.3.2 File Information Table (FIT)

The next option for presenting the clients with a uniform name space is to use a File Information Table (FIT). This table can contain the location of file within virtual directory, the actual location of file, and the other file attributes. It contains information of all the files available to the conference. This table, along with the files actually located at that site, is stored by the server. When a user makes a request for a file, the DFS server checks the FIT table to get the actual location of the file instead of using symbolic links.

If it is located at the current server, the request is continued. If the file is located on another server, the request is then forwarded to that server. This DCS server executes the request on the actual file, and sends the response to the original server. The original server then replies to the user that made the request. When a new site joins a conference, the FIT is the only thing that needs to be sent to this new site.

The advantage to this option is its simplicity. The server does not need to construct any symbolic links. It simply needs a copy of the FIT. This table can also be kept in memory for faster access.

5.3.4 Design Decisions

DFS is implemented using an FIT. This is done for several reasons. First, the file attributes already need to be stored in some sort of table or data structure. This can also be done in the FIT. Second, only the FIT needs to be sent to the new sites for a conference. Third, the new sites do not need to construct symbolic links for the files.

5.4 File Information Table (FIT)

The previous section states that an FIT should be used by the conferences to store all the available files and their attributes. Each conference on a site has its own FIT that it uses to keep track of the files available to that conference. The FIT is sent to other conferences so that all conferences are aware of the same files and a virtual directory structure can be presented to the clients. The FIT is stored in a file at the site and is also kept in memory for quick access.

The FIT is modified by the users connected to every conference. Files can be created, deleted, copied, and moved. Also, the file's attributes can be modified. All these actions change the FIT in some way. In order for the uniform name space to be preserved, these changes need to be implemented in the FITs at other sites. This must be

done in a serialized manner so that the FIT is consistent on each site. This is similar to updating a file that has many replicas. There are several ways this can be done. These are discussed in the next section.

5.4.1 Design Options

5.4.1.1 Database Services (DBS)

DBS is a DCS module designed by Amit Vinayak Date [26]. The DBS module implements a distributed database to maintain conference tables and uses a causal order protocol with a message sequence number to keep these tables consistent. In DCS, tables are associated with each conference and are replicated at all participating sites in a conference. Replicating tables provides high availability, however, it also introduces consistency issues. Each modification of a table, must also take place at every replica. The modifications should be serialized so that each table is kept consistent.

DBS maintains consistency between the table replicas by having a coordinator or owner that serializes the updates. When a site inserts a row into a table, the site owns the row. All updates to that row are done first at this owner site. When multiple sites want to update the same set of rows, the owner site serializes the updates avoiding possible race conditions. The owner multicasts this message to the other sites of the conference in an effort to keep each table consistent. However, in order for consistency to be maintained, all messages executed in some order at the owner site, must be executed in the same order at all participating sites. This is done using a sequence number at each site with all other sites only executing a command if they have received all previous messages from that site. Each site maintains a vector in which it keeps the highest in order sequence number from all sites in the conference. A site only executes a message from a site when it has received all messages that the other sites also received or sent

before the current message. This is known as a causal order protocol and DBS uses this to implement an atomic reliable multicast (i.e., the message is either received by all the servers, or none of them).

In order for DFS to use DBS, the FIT would have to be all or partly maintained by the DBS database. After a file is created, only a few of its attributes will ever change. Where its replicas are located and the latest versioning information are two file attributes that will change. These can be kept in a separate table from the other static file attributes. Either the site that created the file or the home site of the conference can own these rows. All updates to these attributes can be then be coordinated and serialized by the owner.

There are several advantages of using DBS to maintain the FIT and provide consistency. First, it takes a large responsibility off the DFS Servers. This allows the DFS Servers to focus on providing efficient user access rather than keeping the FIT updated. Second, DBS already maintains the conference tables and provides table consistency. It only makes sense for it to also maintain the FIT and ensure that it is kept consistent. Unfortunately, DBS is not fully functional at the moment. So in the current system, FIT consistency must be maintained by other means.

5.4.1.2 Implement a method for FIT consistency in DFS

In this option, DFS implements its own methods to maintain FIT consistency. This can be done using several different multicast orderings. First-In-First-Out (FIFO) order is when messages are multicast from a single source and are received in the order they are sent. Causal order, as used in DBS, is when causally related messages from multiple sources are delivered in their order. Finally, total order multicast is when all messages multicast to a group are received by all members of the group in the same order. Total order multicast is expensive and difficult to implement. It is also more than is required

for DFS. Causal order multicast is appealing because provides consistency when messages are sent from multiple sources. This occurs in DBS because different sites can own rows in the database. However, with regards to the FIT, there needs to be one overall owner. This is because in this option, since DBS is not used, the FIT is simply stored in a file at each site. There can also be a situation in which two sites try to create the same file. These actions must be serialized in some manner to ensure consistency among the sites. Since there must be one owner of the FIT, multicast message need only come from a single source. So although causal order is appealing, FIFO order is all that is needed.

FIT consistency is achieved in DFS using through the following methods. The entire FIT can be owned by one site (usually the first site or home site of a conference). All modifications to the FIT are sent to this owner site first. This site serializes all updates and ensures that they can be executed successful. After it performs the necessary error checking, the owner site then multicasts the updates to every site. This is done using a point-to-point unicast. This ensures that messages sent from the same site are received in the order they are sent. However, this does not guarantee that each update will be sent to the same site in the same order. One possibility is to user mutual exclusion to ensure that the owner sends an update to all sites, before it sends any other updates. However, this limits the concurrency of the system. Not all update messages will modify the same information so it is not reasonable to make all update messages wait for each other to complete entirely. Instead, a message sequence number is used with each message so that the recipients can ensure that they all execute the same updates in the same order. Mutual exclusion is used to guarantee each message gets a unique

message number and this number indicates the order in which it is received and error checked by the server. After a unique message number is obtained for a message, the server is free to accept and check other messages. Since it is possible for multiple threads to concurrently be sending update messages, the message sequence number ensures that each site executes the messages in the same order. This provides each site with a consistent FIT.

5.4.2 Design Decisions

FIT consistency is achieved through the implementation of DFS's own method for reliable multicast. This is done using a message sequence number and FIFO order protocol where messages from a single source are delivered in the order they are sent.

There are several issues that are raised with this option. First, what happens when the FIT owner becomes unavailable to some or all sites? Similarly, what happens when site becomes unavailable to the FIT owner? These are serious issues, however, they are not addressed in this option because it is felt that DBS should eventually be used to maintain FIT consistency. The implementation of FIT consistency within DFS is done only because DBS is not fully functional. This allows DFS to be fully utilized and tested.

5.5 Version Control

5.5.1 Basic Design

DFS uses a versioning scheme with immutable files. This provides DFS with the concurrency control needed to ensure that a user's writes are protected. When a file is modified the changes are stored in a new version and not in the original file. The basic design of this scheme is simple.

1. When a user opens a file, the entire file is cached at the user's machine.
2. All reads and writes are then done to this local copy.
3. When a user closes the file, it is saved back on the server as a new version.

This scheme allows multiple users to open and modify the same file. It guarantees that their modifications will not overwrite each other or become interwoven. It also allows the possibility of retrieving an older version if the changes made are not desired. Finally, it also allows the possibility for versions to be merged and a new file created.

5.5.2 Versioning Policies

A user can select a policy that determines what version is returned when a file is opened. When a multiple version scheme is used, a versioning tree develops. Some versioning trees for popular version control systems can be seen in Chapter 3. When multiple users are all interacting with the same versioning tree, they call can create their own versions and branches. When this happens, there are three specific versions within the tree that are of particular importance. These three versions are:

1. The last version created by time,
2. The last version created by name,
3. The last version created by user.

The last version created by time is simply the last version to be added to the tree. This could be on any of the branches, but it will always be at a tip. The last version created by name is simply the version with the highest version number or name. This usually corresponds to the last version on the main branch or trunk of the tree. The last version created by user is the last version created by a particular user. Each user interacting with the version tree will have a version that he or she last created. DFS is designed to allow a user to specify their individual policy. The user can choose which version they would like returned when they open the file.

There are still several other design options that need to be considered. These are presented in the next section.

5.5.3 Design Options

5.5.3.1 Client makes delta file

In this option the client computes the delta file for the new version and sends only the delta back to the server. This option is possible because the client will already have the entire file cached on their machine. When this occurs a copy of this file should be made and preserved so that the modified file can be compared and the differences saved. There are two advantages of this scheme. First, it takes some responsibility off the server, which can increase performance. Second, a smaller file is sent back from the client to the server. This reduces network congestion.

5.5.3.2 Server makes delta file

In this option the server computes the delta file for the new version. The client performs all writes on the original cached copy and then sends the entire file back to the server when it is closed. The server then compares this file with the original and saves only the changes. This option has one main advantage. It allows the server to determine if a forward or reverse delta should be used. This is discussed further in the next two options. If the delta is only sent back to the server, then the server has less control and no choice but to use the delta sent. This could lead to an inefficient use of deltas when retrieving various versions. The disadvantage to this scheme is that it requires more work from the server and more data is being sent over the network.

5.5.3.3 Forward deltas

In a forward delta the entire original file and only the changes made to retrieve the new versions are saved. If only forward deltas are used, the original file, no matter how small or big, would be the only file kept in its entirety. All other versions are simply constructed, as needed, from the string of deltas. The disadvantage of this is that every

time the most recent version of a file is request, the entire file must be constructed from the versions. This causes the performance of opens to suffer. An example of this can be seen in the SCCS versioning tree in Figure 3-1.

5.5.3.4 Reverse deltas

In a reverse delta the entire latest file and only the changes needed to retrieve the original are saved. If only reverse deltas are used the most recent version of a file will always be saved in its entirety. Reverse deltas that allow older versions to be constructed will also be saved. In this scheme the latest version of each branch in the version tree must be stored. This increases the amount of storage space needed. However, the advantage of this scheme is that the most recent version of a file can be retrieved quicker. This allows the open call to perform better.

5.5.3.5 Combination of forward and reverse deltas

In this scheme a combination of forward and reverse deltas are used. The newest version of the file on the main branch of the version tree can be saved in its entirety so that it can be returned quickly on an open call. Reverse deltas can be kept so that past versions can be retrieved. However, if there is a branch in the versioning tree, then forward deltas can be used along the branch. This is equivalent to how the Revision Control System (RCS) and the Concurrent Versioning System (CVS) function. An example of this versioning tree can be seen in Figure 3-2. The disadvantage of this scheme is that the performance of open suffers with retrieving the latest version of a file that is not on the main branch.

5.5.4 Design Decisions

The DFS versioning scheme is designed so that servers can create either forward or reverse deltas. This option is selected because it is felt that most of the time, opens will

be done on the latest version or versions only a few increments from the latest. By saving the latest version of a file in its entirety we ensure that opens on it and versions close by will be handled quickly. Opens on older versions are slower because of the time to construct the file from the series of deltas. This scheme is only possible if the servers have control and can choose to make either the forward or reverse deltas. Therefore, the creation of the delta is done by the server.

5.6 Replication Control

In order to be fault tolerant and provide clients with better file availability, DFS replicates a file at multiple sites. If server that holds a file is unavailable at one site, then the file can be retrieved from one of its replicas. This process is transparent to the user unless the site that the user is currently connected to becomes unavailable. However, if this happens the user can use the services provided by the Conference Control Services (CCS) to connect to the server at another site. Since file changes are stored in new versions and files are immutable, no replica updating, consistency, or coherency issues need to be addressed. When a file is accessed, it can simply be replicated for retrieval. Several design options of this scheme are discussed in the next section.

5.6.1 Design Options

5.6.1.1 Entire file is replicated

In this option the entire file is replicated at various sites. An entire file does not mean the entire file constructed from deltas. A delta is considered a file by itself and it is also replicated. The advantage of this scheme is its simplicity.

5.6.1.2 File is split and portions replicated

In this option a file is split and its portions are replicated at various sites. This option is illustrated in Figure 5-6. Assume a single file, example.txt, is split into three

sections: A, B, and C, and that there are also three sites: S1, S2, and S3. Figure 5-6 shows how these sections can be replicated at theses sites to increase availability. Now any one site can become unavailable without the file becoming unavailable. It can always be constructed from the other two sites. This is similar to a concept done in RAID called data stripping and mirroring. RAID is discussed in Chapter 2.

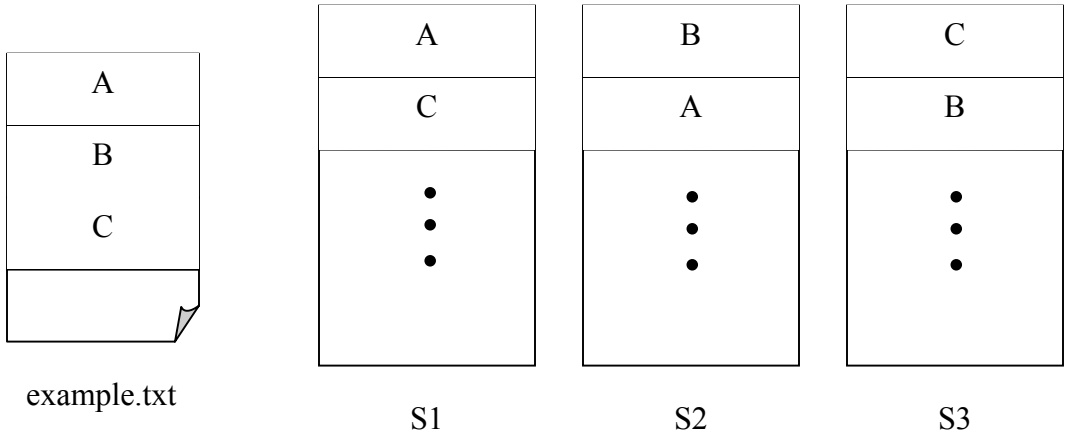

Figure 5-3. An illustration of how a file can be split and replicated to increase availability

5.6.1.3 Replication occurs when file is created

In this option files are replicated as soon as then are created. A replication factor can be used to determine the number of sites at which a file is replicated. This replication factor has a default value and can also be set by the user. This allows the user to decide how much a file should be replicated. If the file is very important, it should be replicated at more sites. This will use storage at these sites but ensure that the file is availability a larger percentage of the time. If a file is not very important, it should only be replicated at a few sites to save storage space. The advantage of this option is that it allows the user to decide how important it is that a file is available.

5.6.1.4 Replication occurs when user at a remote site accesses file

In this option replication only occurs when a user a site where a file doesn't exist requests to access the file. The file is then replicated at this site. The advantage of this scheme is that a replica of a file will always exist at the site where the user accesses the file. This ensures that the later user requests from the same site will occur quickly. The disadvantage of this scheme is that if a file is not replicated before a site goes down, the file could become unavailable.

5.6.2 Design Decisions

In DFS, the entire file is replicated when it is accessed on another site. Replication of entire files and not portions is selected because of the simplicity. Also, file splitting is most effective when done at several failure-independent storage systems at the same site. This way writes can be optimized by performing in parallel. Replicating a file at another site when it is requested is selected because it will yield better performance for sites that are constantly accessing certain files. This technique ensures that the site where a request occurs will have a replica of this file. Later requests will use the copy located at this site. If an older version of a file is requested, the entire rebuilt file for that version will be replicated at the site. It does not make since to replicate only the delta because the other deltas and the last version are need to reconstruct the version. This will have to be done when the file is read and cached on a user's machine. It is more efficient to rebuild the file and replicate it on the site during the first reference.

The downside of this technique is that it does not provide any default replication for files. A file is not replicated unless it is access on another site. If a file is not accessed on another site, and the site that holds the file becomes unavailable, then the file will also be unavailable. Currently, it is felt that this tradeoff is acceptable because of the

simplicity and efficiency of the option chosen. However, future designers should still consider replicating a file when it is created.

5.7 Caching

Each user has a DCS cache directory on their local machine. This is where the files that are opened and copied from the server are placed. After a file has been opened, all reads and writes take place on this local file. When a file is closed, the modified file is sent back to the server. Once a file is closed it is expelled from the cache. File attributes and directory contents should not be cached as they are in some distributed file systems. This is because with a multi-user multi-versioning scheme, new versions of a file are likely to develop. If cached directory contents and file attributes were used, the user would not be able to access the new versions until their cache is updated. Therefore, it is decided that the server will always be contacted on all directory content and file attribute requests.

5.8 Consistency Semantics

The exact consistency semantics of DFS compared to other distributed file systems is not straightforward. NFS does not provide strict consistency semantics because it is not guaranteed when the writes made by one user will become visible to other users. AFS and Coda implement session semantics that guarantees a file will be consistent after it is closed. In other words, the changes made by each user will not be visible to other users until the file is closed. When this happens the callback is broken signifying that the cached data is invalid. The users must then reopen the file to view the changes made by the other users. Finally, even though DFS uses immutable shared files, it does not exactly follow the immutable shared file consistency semantics because a versioning scheme is used.

In the immutable shared file consistency semantics, each file is considered a completely separate entity. This is not completely true for DFS. DFS allows new versions to be created from immutable files. Also, since the versions are stored as a combination of deltas, a single delta can be used to construct multiple versions. Since the versioning scheme is kept transparent from the user, from the user's standpoint it may appear that other user's changes are simply visible when the file is closed. This is similar to session semantics. However, since the user is allowed to specify their versioning policy or select a specific version to open, he or she has the possibility to see or not see the new versions resulting from other user modifications.

For clarity consider the following scenario where User A and User B are working on the same file. They both open the file and it is cached locally at each user's machine. They proceed to read and write their local copy, without interference or even knowledge about the existence of the other user. Now say User A closes the file creating a new version on the server. This has no immediate effect on User B. User B can continue to modify their local cached copy. Now say User B closes their file creating a new version on the server. Now there exists two separate versions on the server: one that contains the modifications made by User A and one that contains the modifications made by User B. Say User A logs off and goes to lunch while User B keeps working. When User A returns and opens the same file as before, there are several possibilities of what could occur.

1. If User A's versioning policy is set to open the latest version by user, User A will see the exact same file as before.

2. If User A's versioning policy is set to open the latest version by time, User A will see a completely different file than before. User A will see the last version of this that was created by User B.

3. Finally, if User A's versioning policy is set to open the latest version by name, User A will also see a completely different file than before. User A will see the version with the highest version name created by User B. Many times this will correspond to the last version created by time, but not always.

From the user's standpoint, one can see that the exact consistency semantics for DFS are not clear and depend on the user's policy and what version he or she opens. This creates a situation where an ignorant user might think system is behaving inconsistently. Consider the following scenario where User C is simply reading a file. Assume that he or she is unaware that their versioning policy set to open the latest version by time. When he or she opens the file, the current latest version by time is cached on their machine. Now at the same time User C is reading the file, other users are creating new versions. Say User C closes the file without making any modifications, then immediately reopens the file. Since their policy is set to open the last version by time, User C will now see a different file than before. He or she will see the last version that was created while he or she was reading the file. This could be a version that is derived from the version User C was previous reading or it could be branch from an earlier version. Thus, the file could possibly have very little in common with the file User C just closed. Now while User C is looking at the file puzzled, another version is created off some other branch in the versioning tree. User C decides to try again and closes and reopens the file. Now he or she may see a file with a completely different set of modifications. This could be very frustrating to users and lead them to believe the system is performing inconsistently. However, in order for users to see the version they previously opened, they simply need to set their versioning policy to open the latest version by user.

From the preceding scenarios and discussion it can be seen that the consistency semantics of DFS are unlike that of any other distributed file system. Also, even though the system is behaving normally, it can appear that it is inconsistent to an ignorant user. However, this does not mean the system is inconsistent. This makes it difficult to exactly compare the consistency semantics of DFS with that of other systems. However, this design provides DFS with concurrency control that allows users to concurrently modify a single file, while ensuring that their writes do not interfere with each other. DFS is the only distributed file system with this characteristic.

5.9 File Attributes

As mentioned earlier, an FIT is used to store the actual locations of files. This table is also used to store all file attributes for every file available to the conference. Chapter 4 lists all the file attributes required. This list is reproduced below for convenience.

1. Location of the file within DCS space
2. Type of file
3. Owner of file
4. Conference to which the file belongs
5. Size of file
6. Number of entries in directory if the file is a directory
7. Last access time
8. Last time a new version was created
9. Time file was created
10. Mode at which the file can be opened
11. Last version of file created by time
12. Last version of file lexigraphically or by name
13. Location of replicas
14. Number of replicas

5.10 File System Interface

DFS provides an interface to the file system located at the servers. This interface is used by clients, DFS Servers, and other DCS modules. DCS aware and unaware

applications should be able to use this interface. Below is a list of commands that compose the file system interface. A brief description of the functionality is also given.

1. Create: This creates a blank file on the server's machine.

2. Open: This command copies a file from DCS space to the user's cache directory. This can be done with or without specifying a specific version. DCS aware applications can specify a specific version. Both DCS aware and unaware applications can open a file without selecting a specific version. If a specific version is specified, that version is returned. However, if no version is specified, the version that is returned depends on the user's versioning policy, discussed earlier.

3. Close: This sends the modified file in the cache back to the server.

4. Read: This command is to read a file on a server. The entire file or only a certain size can be read. This command is called only from other servers and not from clients. Other servers use this command when a client from server requests to read a file. Clients always use open to copy the file to client space and use the read system call to read that file.

5. Write: This command is to write to a file on a server. The write can be appended to the end or overwrite the file starting at the beginning. This command is called only from other servers and not from clients. Other servers use this command during inter-conference operations where a file is moved or copied from one server to another. Clients always write files as new versions when the file is closed.

6. Delete: This command deletes or unlinks a file. If this is called by a DCS aware application, it has the option to specify if this is an absolute delete. Absolute deletes actually remove the file or version and free its storage. However, DCS unaware applications are not able to specify an absolute delete. When an absolute delete is not specified, due to this being a shared multi-versioning environment, the file or any of its deltas is not actually deleted. This is done because other users still might want to access this particular file or version. When a non-absolute delete is performed it simply removes any information that the user actually viewed the file. So an open to the last version by user would not return this file anymore. It would return the next default version specified by the user.

7 Make directory: This command creates a directory on the server. This is called by clients, other DFS servers, and other DCS modules. The CCS needs to call this method to create the DCS and conference directories on a new site.

8 Remove directory: This command removes a directory. This can also be called by clients and other servers.

9. Move: This command moves or renames a file from one location to another. This can occur within a conference or from one conference to another (with the correct permissions of course).

10. Copy: This command copies a file from one location to another. Like move, this can occur within a conference or from one conference to another.

5.11 Summary

This chapter presents the design options and decisions for DFS. All options considered are documented even if they are not used. This allows the reader and future designers to understand the reasons why some choices are made instead of others. DFS provides a design that meets the current needs of DCS, yet is flexible enough to be adapted to meet the future needs. It uses an FIT to handle the management of files. It is kept consistent on each site by serializing each update through one owner and using a FIFO order protocol to ensure that messages and received by each site in the same order as they are sent. Even though the consistency semantics is difficult to compare with other distributed file systems, DFS provides better concurrency control by using a versioning scheme with immutable files. Availability is raised by replicating files. These are all done in ways that are flexible enough to give the DCS users control over their interaction with DCS file space.

CHAPTER 6
IMPLEMENTATION

6.1 Introduction

This chapter discusses the implementation details of the Distributed File System (DFS) module for the Distributed Conferencing System (DCS). The specific components that are implemented and how they interact with each other are discussed. First, the overall architecture of DFS and how it interacts with other DCS modules is presented. In the following sections the major components of DFS are discussed: the DFS Client and Server, the FIT Manager, the Version Control Manager (VCM), and the Replication Control Manager (RCM). Next, the implementation of the DCS Shell and DCS Text Editor is discussed. These two tools provide the user with methods to interact with the DCS file space. Finally, the chapter concludes with a summary of the main implementation details.

6.2 DFS Architecture

DFS is implemented using the model shown in Figure 6-1. It should be noted that in the implementation of the DFS server there is no difference between the regular conference servers and the vestibule conference or site server. The only difference lies in what set of directories the servers manage. The box represents the vestibule conference or site server and the other conference servers are represented by ovals. Each DFS Server consists of four components: the actual DFS Server, the Version Control Manager (VCM), the Replication Control Manager (RCM), and the FIT Manager. These

components are discussed further in the following sections. Figure 6-2 illustrates the architecture of the DFS Servers.

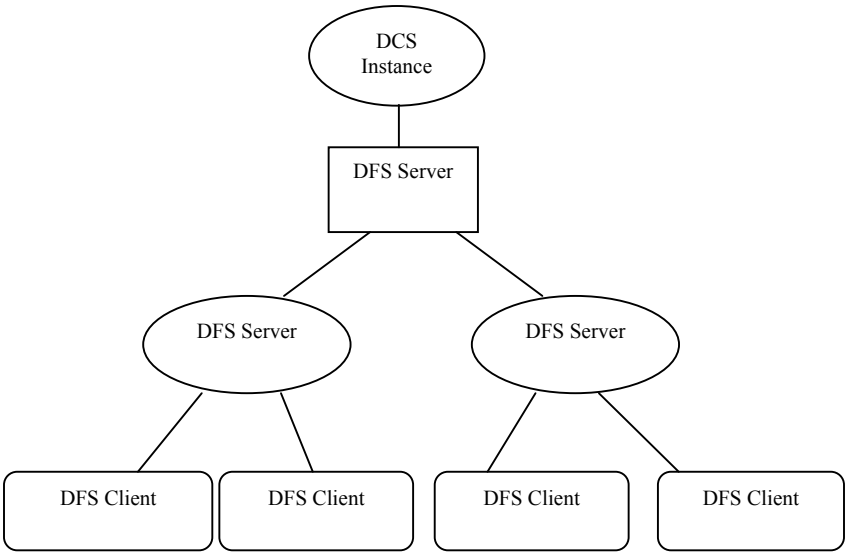

Figure 6-1. The architecture model of DFS

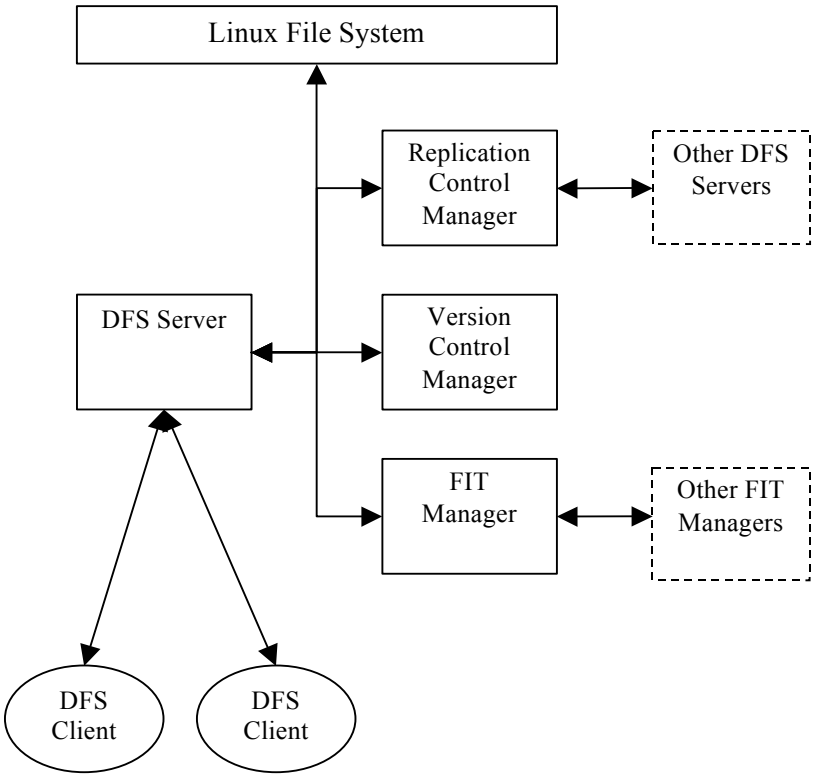

Figure 6-2. The architecture model of the DFS Servers

6.2.1 Conference Control Services (CCS) Interaction

The Conference Control Services (CCS) is designed using a client/server architecture [25]. The CCS provides a User Manager (UM) and two types of servers: a Conference Server (CS) and a Site Servers (SS). The UM is the user interface that is the client to the CCS servers. It provides the Graphical User Interface (GUI) that allows users to log in to DCS and perform the conference operations provided by the CCS servers. There exists a CS for every conference on a site and one SS per site. The CS receives the requests from the UM and handles all conference related requests. The SS is the server to the CS and it handles their interaction and communication. Figure 6-3 illustrates the architecture model for CCS. The DFS Client and DFS Servers in Figure 6-1 figuratively sit underneath the UM, CS, and SS respectively. Figure 6-4 illustrates the interaction between the DFS and CCS components.

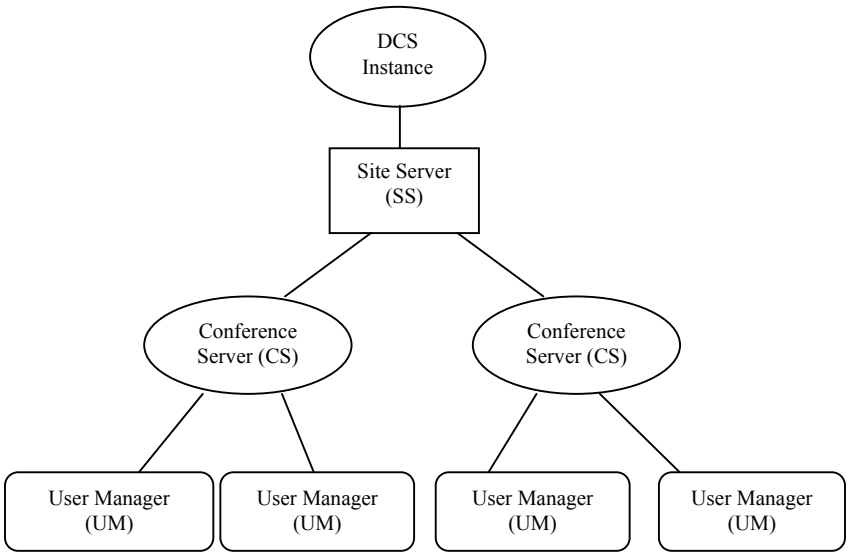

Figure 6-3. The architecture model of CCS

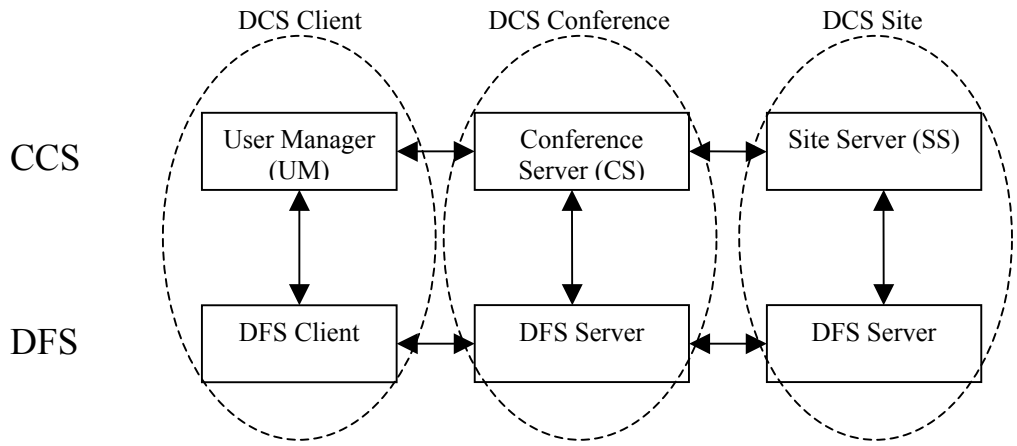

Figure 6-4. Communication between DFS and CSS components

6.2.2 Database Service (DBS) Interaction

DBS uses a causal order protocol with a message sequence number to reliably communicate information and keep the conference tables consistent. Currently, DFS does not directly interact with DBS. DFS implements its own scheme to ensure FIT consistency. However, in future designs, DBS should also maintain the FIT. DFS should interact with DBS when the FIT needs to be updated at other conferences. When a change is made to the FIT at a conference, the DFS Server contacts the FIT Manager, which contacts DBS. DBS then contacts the DBS at other sites, which contacts the conference FIT Managers to notify them about the change.

6.2.3 Access Control Service (ACS) Interaction

DFS interacts with ACS in several ways. First, when a file is created it is added to the ACS tables, identifying which roles have what permissions. Second, when a file is accessed the ACS is checked to verify that the user has permission to perform the action requested.

6.2.4 Communication

Communication between the DFS components and the other DCS modules is done in two ways: normal method calls and Java Remote Method Invocation (RMI). Inter-communication between the components of a DFS Server is done using normal method calls. The DFS Server creates the FIT Manager, VCM, and RCM objects and uses a reference to these objects to invoke their methods. The DFS Server also has a reference to the CCS Server object and uses this to invoke its methods.

Intra-communication from DFS Client to DFS Server and DFS Server to DFS Server is done using Java RMI. Java RMI allows the calling of methods on remote objects as if they were local. Each DFS Server binds itself to the rmiregistry using its conference name. There is only one rmiregistry per site that runs on a well-known port. The DFS Clients can then call methods on the DFS Servers if they know the address and port of the rmiregistry and the name of the conference. The RCM also calls methods on other DFS Servers to read a replica of a file. The FIT Manager also binds itself to the rmiregistry. This allows FIT Managers to contact other FIT Managers and provide FIT consistency.

6.3 DFS Client and Server

6.3.1 DFS Client

The DFS Client provides the DCS user an interface to the DFS Servers or file system. It uses Java RMI to call methods on the DFS Servers as if they were local methods. The DFS Client is created when the user logs onto the conference. It binds itself to a single DFS Server and creates a dcs/ directory in the user's home directory. There should only be one DFS Client per user and all calls to the DFS Server go through this client. A reference to the DFS Client is used by applications running on the user's

machine to also interface with the server. These applications call methods on the DFS client, which then call methods on the DFS Server.

Most of the methods in the DFS Client simply call methods on the DFS Server and return the result. However, the open and close commands perform slightly more work. The open command reads an entire file from the DFS Server and stores a copy in a local cache directory, /dcs/.dcs_cache. When a file is read from the server, it is stored in the cache directory in another directory of the current time in milliseconds plus the DCS path of the file. For example, say the client calls an open on /dcs/file1.txt. The open command reads a copy of this file from the server and stores it in /dcs/.dcs_cache/1059085690157/dcs/file1.txt, where 1059085690157 is the current time in milliseconds. The entire path is then returned to the calling application and can be used to access this local copy. Also the version of the file read is stored in /dcs/.dcs_cache/1059085690157/dcs/file1.txt.ver.

The close command is used to close a file and write it back to the DFS Server. It should be called with the local cache path of a file returned from the open as an argument. It reads this file and the stored version number and sends it back to the DFS Server. The DFS Server then uses this information to create a new version for this file.

6.3.2 DFS Server

The DFS Server is the main component of DFS. It handles all the client requests and implements the file system commands. It also interacts with the FIT Manager, VCM, and RCM, invoking them as needed.

The DFS Server also manages the DFS Hashtable. The DFS Hashtable is a hashtable used to store information about all the available files in memory. This provides efficient management and retrieval file information. When a DFS Server starts, it

invokes the FIT Manager to get a copy of the FIT file from another conference. The FIT Manager and FIT file are discussed in detail in the next section. Once the DFS Server has an FIT file, it reads it storing each file and its information in a DFS_SysFile (DFS System File) object. A DFS_SysFile object is created for each file and stored in the DFS Hashtable. The file's path name is used as its key or identifier to the hashtable to return its DFS_SysFile object. This allows a particular file and all its information to be quickly retrieved.

When a user makes a file request the pathname is used to retrieve the DFS_SysFile object from the DFS Hashtable. If the object is not present in the hashtable it is assumed to not exist within the conference and an error is returned. If the object does exist, the other necessary error checking occurs. If there are no errors and the client has permission, the desired action takes place. This occurs first on the local file system if necessary, to ensure that the request can be completed successfully. Before the DFS Hashtable is modified, the FIT Manager contacts the FIT owner for this conference to assure that this request can take place. Finally the DFS Hashtable is updated and the local FIT file is modified.

6.4 FIT Manager

The FIT Manager handles the management of the FIT file and the updating of the file information at other DFS Servers. It contacts the other FIT Managers when a server begins and needs to get a copy of the FIT file. It also communicates to the particular FIT Manager that owns the FIT file. This FIT Manager communicates to other FIT Managers informing them of the changes that occur to the FIT. This ensures that all the DFS Servers for a conference will have a consistent and up-to-date list of files.

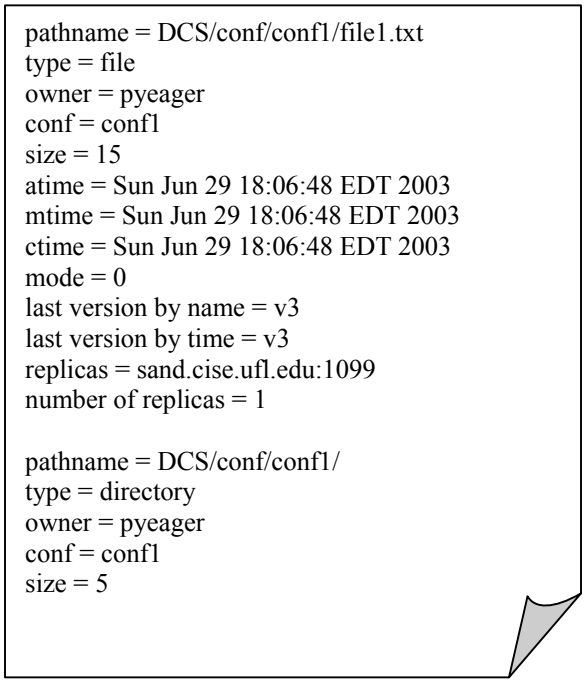

Figure 6-5. Format of FIT file

6.4.1 File Information Table (FIT)

Each conference has its own FIT for the files it manages. The FIT is stored in a file called FIT.dcs. For the vestibule conference, the path is /dcs/sys/FIT.dcs and for other conferences the path is /dcs/conf/<conf-name>/sys/FIT.dcs. A sample FIT file is shown in Figure 6.5 to give the required format. The first field or pathname is the DCS path of the file. This is also the key or identifier used to retrieve the file from the hashtable. The next field is the file type. This can either be a file or directory. The owner and conf fields represent who created the file and at what conference. The size is the size of the file in bytes or the number of items in the directory. The atime and mtime are the last time the file was accessed and a new version was created respectively, while ctime is the time the original file was created. The mode field is currently not implemented but is included to represent the mode in which files can be opened (e.g., read-only or read-write). The last version by name field is used to store the highest version of this file that

exists, while the last version by time field is used to store the last version of this file that was created. The version numbering is discussed further in a proceeding section. The replicas field stores the location of this files replicas. This is in the format: <host name>:<host port>. Finally the number of replicas field simply stores the number of replicas of this file that exist.

6.4.2 FIT Consistency

There are two times when the FIT at DFS Servers can become inconsistent: when a server starts and when a change occurs. These two situations and the procedures used to keep the servers consistent are discussed below.

6.4.2.1 When a DFS Server starts

When a conference is created at a site, the DFS Server invokes the FIT Manager to get the latest copy of the FIT file. The FIT Manager contacts the CCS Server to see if the same conference exists at other sites. The CCS Server contacts DBS to check the site databases and get the conference locations. If the conference exists at other sites, the CCS Servers respond with the location of these conferences. The FIT Manager then contacts the FIT Manager at one of these other locations to read a copy of the FIT file. If another conference doesn't exist, the DFS server continues booting using the FIT file that is already present or creating a new one. When this happens this site becomes the owner of the FIT file. If an FIT file that is already present is used, it is possible that some of these files are actually located at other, now unavailable servers. If this occurs these files will be identified as unavailable when they are accessed. Every time the DFS server boots up, whether when a conference is created or after a crash, it should always try to get the latest copy of the FIT from another conference. This is to ensure that this server has the latest conference information since there is no way to know how long the server

has been offline. After an FIT file has been retrieved or created the DFS Server then loads this file into the DFS Hashtable in memory and waits for client requests.

6.4.2.2 When a change occurs

When the FIT is modified at one server, this change also needs to be reflected at the other servers in order for a consistent and uniform view to be presented to the clients. The FIT is changed at one server when a file or directory is created, removed, or when a file's information is changed. When this occurs the DFS Server invokes the FIT Manager to notify and update the other DFS Servers. Consistency is achieved using a message sequence number and FIFO ordering for the messages. This is done with the following procedure.

1. The FIT Manager is first notified about a change.
2. The FIT Manager then contacts the FIT owner to try and execute the update.
3. If the owner can successfully complete the update, it sends this update to the other sites of the conference. It uses a message sequence number to ensure that each site receives the messages in the same order.
4. When the FIT Manager receives an update message, it checks to make sure the attached message number is the next message number in sequence. If so, it executes the update in the message. If not, it delays executing the message until the previous messages are received.
5. When a FIT Manager executes an update, it makes the necessary changes to the FIT that exists in the server's memory. For example, if a file is created, the FIT Managers will simply add the file and its information to their DFS Hashtable.

The above steps ensure that each conference has the latest information about the files available. This provides clients a uniform name space regardless of where files are located. Note that the updated servers do not write the modified FIT that exists in memory immediately to the FIT file. This does not occur because the clients at that site already have access to the changes once the FIT is in the server's memory. The server

only writes this to its local file when a change occurs from a client at this site. If the server happens to crash before it writes it to file, it will still retrieve the latest FIT file because it always request this from other servers when it restarts. Refer to Chapter 5 for further details of how FIT consistency is maintained.

6.5 Version Control Manager (VCM)

The VCM handles the versioning aspects of DFS. It implements the versioning naming as well as the creating of new versions and rebuilding of old versions. When a client closes a file and the new version is sent back to the server, the VCM handles the creation of the forward or reverse delta file. When a version is read, the VCM also handles the rebuilding of this file using the complete file and the sequence of deltas.

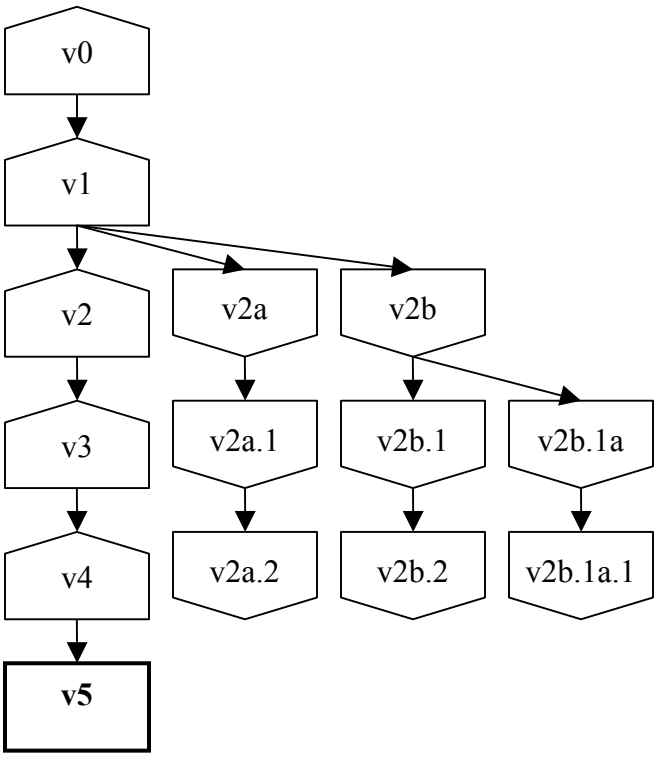

Figure 6-6. An example a version tree in DFS

6.5.1 Version Naming

The implementation of how versions are named in DFS is illustrated in Figure 6-6. The box in bold represents the complete file. The boxes with points represent deltas. If the point is facing upwards, then it represents a reverse delta from the next version. If the point is facing downwards, then it represents a forward delta from the previous version. The line arrow pointing from one version to another indicates that the later version is derived from the former version. Reverse deltas are used along the main branch or trunk of the tree. The entire file is the latest version on the tip. Forward deltas are used along the other branches of the tree. All deltas are stored in a vers/ directory. This directory is present in every directory that contains a file will multiple versions. The vers/ directory stores all the deltas for all files in the directory.

The version naming policy that is implemented in DFS can also be described with the following characteristics.

1. If the last character in the version name is a number, then subtracting one from this number can retrieve the parent version.

2. If the last character is a letter, then removing the substring from the last period to the end can retrieve the parent version. For example, the parent of v2b.1a is v2b. The exception is when there is no last period in the version name. When this occurs removing the letter and subtracting 1 can retrieve the parent. For example the parent of v2b is v1. Version v0 is the original and has no parent.

3. A version has a sibling version if two or more versions are derived from the same parent version. Siblings can be identified as having the same version name except for the last letter, which increases alphabetically. However, the first sibling created does not have a letter and the second sibling starts with the letter ‘a’.

4. If the last character is a letter, the child version can be retrieved by adding a “.1”. If the last character is a digit, adding 1 to the digit can obtain the child version.

5. If all the letters of the alphabet are used in sibling versions, the alphabet repeats using two letters. For example, if the last sibling version is v2z, the next version sibling would be v2aa, then v2ab, and so on.

For each file with multiple versions, there also exists the file with no version extension. For example, say there exists multiple versions of the file example.txt. Each version of the file would have the version number as an extension, including version 0. When the file with no version extension is selected, the version indicated by the selected versioning policy is returned.

6.5.2 Creating a delta

When a cached file is written back to the DFS server as a new version, the DFS Server invokes the VCM. The VCM then tries to compare this new version with its previous version. If its previous version is a delta, then this file is first reconstructed using the process below. When the previous version is a whole file, a delta is created of the differences. This allows the old version to be reconstructed from the new version. The old version is then removed leaving only the delta. The deltas are saved in files named using the following format: file2-file1.delta. This indicates that this delta is used when constructing file1 from file2. For example, say Figure 6-6 is the version tree for a file named example.txt. When version v5 is added, a delta file is created for version v4 called “example.txt.v5_example.txt.v4.delta”. This indicates that this delta is used to change version 5 into version 4.

6.5.3 Reconstructing a version

In the vers/ directory, each delta has a file called <version_name>.prev. For example the file example.txt.v4 would have a file called example.txt.v4.prev. This file contains the name of the version from which this delta is derived. For example, in Figure 6-6 version v4 is derived from v5, and v3 is derived from v4. This is because reverse deltas are used. In the branches, v2a.2 is derived from v2a.1, and v2a.1 is derived from v2a. This is because forward deltas are used. Saving the version name from which a

delta derived allows us to reconstruct the version following the correct path. This is easier than using only the delta file names such as "example.txt.v5_example.txt.v4.delta". If only the delta file names were used, the entire directory would have to be searched repeatedly to find the correct delta. Then the file names would need to be parsed to find the correct path. Saving the derived delta in a file allows this to be avoided. Saving the derived delta cannot be done in the file's attribute because when a version is requested at another site, the entire version is replicated at that site. Therefore, a version can be a delta at one site and an entire file at other sites. Reconstructing a version is done recursively by tracing the path found in the .prev files until the entire file is found. An entire file has "none" in its .prev file. Once it is found, each version along the path is temporarily reconstructed by implementing the changes in the delta files. This is done until the desired version is finally reconstructed. The path of this temporarily reconstructed file is returned so that it can be read.

6.6. Replication Control Manager (RCM)

The RCM has only one function: to create a replica of a file. When a user reads a file, the server looks at the file's attribute that specifies where the replicas exist. If a replica exists on the current site, then the server continues to read the file. If a replica does not exist on the current site, the server contacts the RCM. The RCM then reads the file from one of its available replicas and stores it on the local site. It also updates the location of replicas in the file's attributes to include the current site. The RCM then returns to the server, which then continues to read the file from the local site.

6.7 DFS Shell

In order for the users to take advantage of the interface and commands provided to them, they must have an application to which they can interact. A DCS Shell is designed

that allows the users to perform the directory operations described above. The DCS Shell is a DCS aware application that functions like a normal shell. It starts directory in the dcs/ directory for the server. The user can move back a directory with the command "cd ../". This allows them to enter their home directory on their local machine. Then the command "cd dcs" will move them back into DCS space. Below is a list of command available in the DFS shell. For commands that take a file path as an argument, absolute paths or relative paths with "../" and "./" can be used.

1. ls: This lists the contents of the current directory. Without any options, just file names in current directory are printed. With "-l" the file names along with the owner, size, and modification times are listed. With "-a" hidden files beginning with '.' are also listed. "-la" or "-al" can be used to list both options.
2. cd: This changes the directory to the argument.
3. cp: This copies the file from the first location to the second location.
4. mv: Similar to copy except original is removed.
5. rm: This performs an absolute delete on a file.
6. pwd: This prints the current or working directory.
7. cat: This displays the contents of a file to the screen.
8. mkdir: This creates a directory.

9 rmdir: This removes a directory.

6.8 DCS Text Editor

The DCS Text Editor is a DCS aware editor. It is implemented to demonstrate the functionality of DFS. Figure 6-7 is a snapshot of the DCS Text Editor Graphical User Interface (GUI). The DCS Text Editor contains a main text area where files can be read and modified. It also contains another text area that displays important messages to the user.

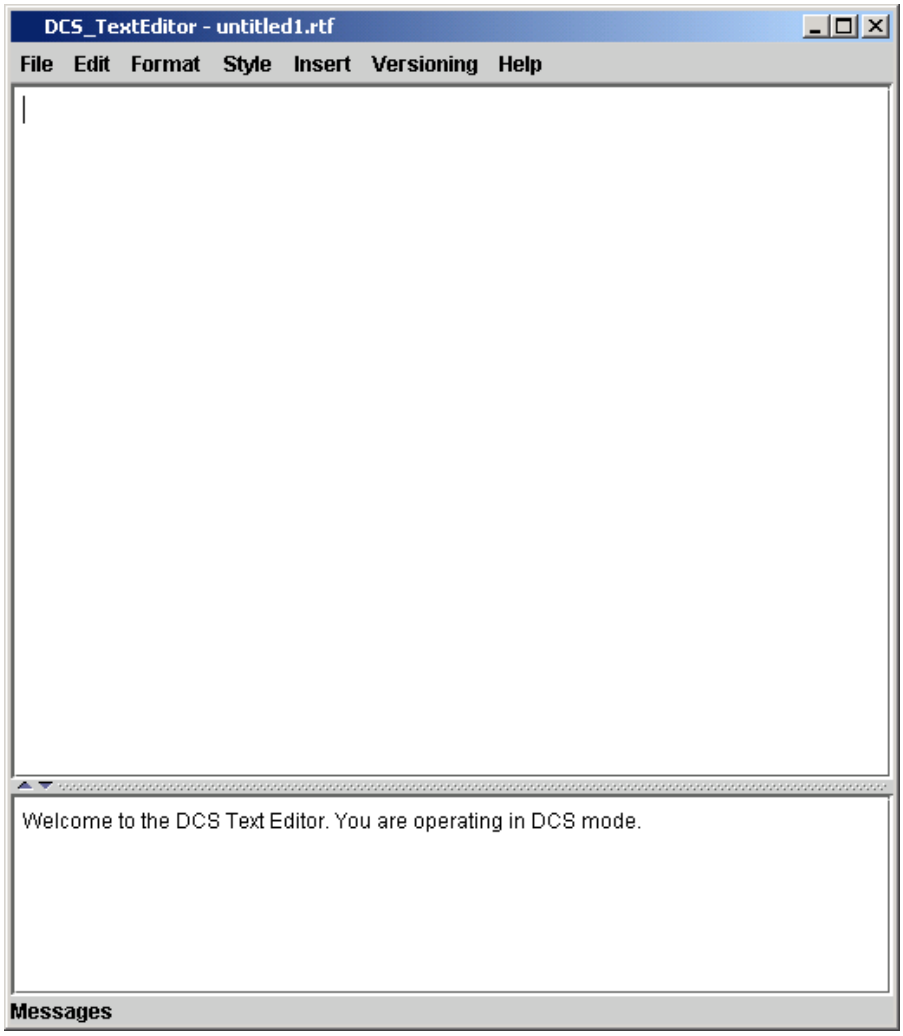

Figure 6-7. The DCS Text Editor

The DCS Text Editor can operate in two modes: DCS mode and Normal Mode. DCS mode is when a valid connection is established when a DFS Server. In this mode everything in the dcs/ directory goes to the DFS Client and to the DFS Server. For example, if a file is modified and saved as dcs/example.txt, the blank file example.txt and version 0 would be created in the dcs/ directory at the server. When operating in Normal mode, the DCS Text Editor is not using a DFS Client that is connected to a DFS Server. All files accessed and saved in this mode are located on the local machine.

DCS Text Editor also allows the user to select their versioning policy. This can be done using the toolbar shown in Figure 6-8. By default, the version policy is to select the latest version by user. When the policy is selected, the choice made is displayed in the message area. Then whenever the user opens a file from the DFS Servers that does not have a version extension, the version indicated by this policy is returned.

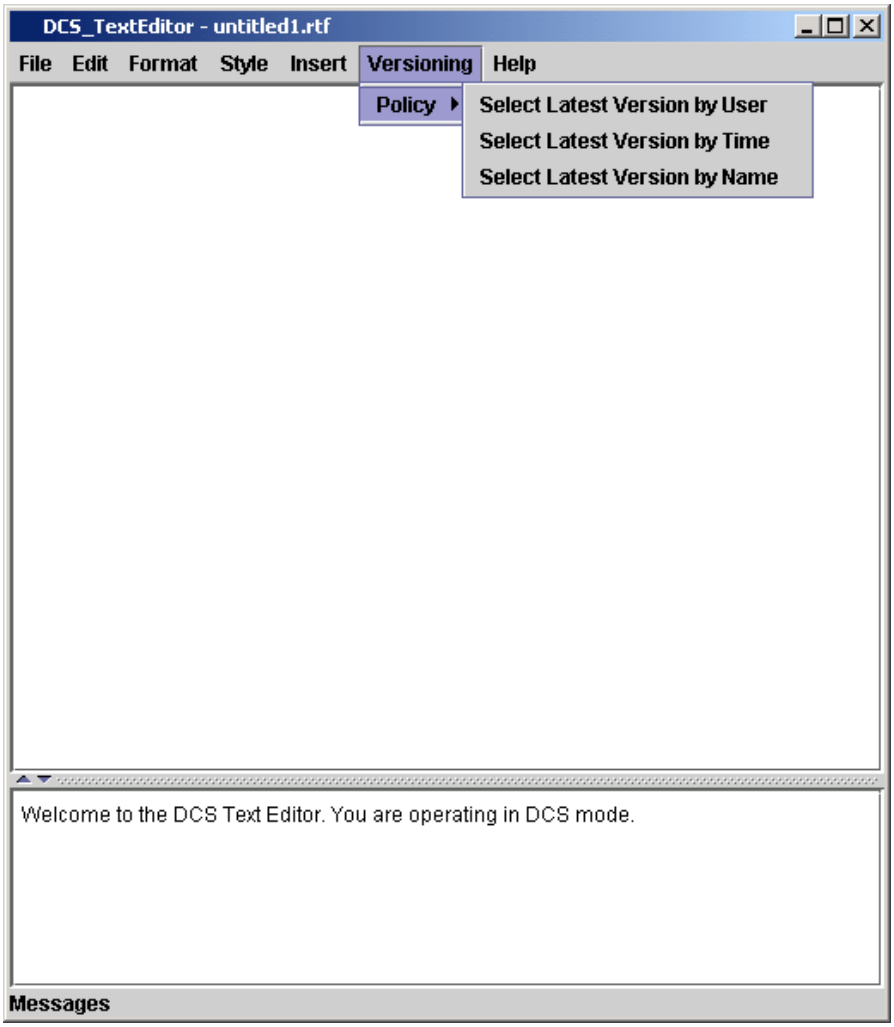

Figure 6-8. The selection of the user's versioning policy.

The DCS Text Editor demonstrates how multiple users connected to remote DFS Servers can use DFS. Through the open and save dialog boxes the available files of this

conference can be seen. When one user creates a new file or new version, it can be immediately seen by the other users when they attempt to open or save a file.

6.9 Testing

In order to fully test DFS, some specific testing classes are created. Unfortunately, at the time this thesis is written the CCS and the DBS module are not completely functional. In order to press forward, a separate testing class is created that imitates the CCS functionality. This class implements the retrieval of other conference locations used by the FIT Manager using some default values. The actual method declarations that should be used in CCS are imitated as closely as possible to allow the future integration to be as easy as possible. Also, since the CCS is needed for ACS initialization, a testing class for ACS is created as well. This class implements basic access control using a set of default values.

In order to simulate and test DFS on multiple sites, the FIT Manager is used to communicate FIT changes to other servers. This allows other servers to see the changes made at one server. This is done by one site owning the FIT file and coordinating all updates. When an update is requested, the site first sends the update to the owner. The owner performs the necessary error checking and then sends this update to the other sites of the conference. This is done using a FIFO order protocol with a message sequence number to ensure that each site receives and executes the updates in the same order. When DBS is fully functional, it should be used to communicate the FIT changes. This is because it already maintains other conference tables and uses a causal order protocol to provide consistency. The current implementation should be used on a limited basis until the DBS code is functional.

6.10 Summary

This chapter presents a detailed description of the implementation of DFS. It describes the four main components of DFS: the DFS Client and Servers, the FIT Manager, the VCM, and the RCM. It describes how these components are implemented and how they interact. The DFS Client and Servers provide the file system interface and implementation. The FIT Manager manages the FIT file and ensures that it is updated on other servers by using a message sequence number with FIFO order protocol. The VCM and RCM handle the versioning details and replication details respectively. Formats are also given for the FIT and the version naming scheme. DFS is implemented following the design presented in chapter 5. It is written in Java and uses the RMI protocol for communication.

CHAPTER 7
CONCLUSION

7.1 Conclusion

The work in this thesis presents the design and implementation of a Distributed File System (DFS) for a Distributed Conferencing System (DCS) version 2. DCS is designed to allow users to efficiently and securely work together in a distributed environment. In such an environment it is common for users to concurrently access and write the same shared files. Other contemporary distributed file systems are not designed with the idea that this will be a common occurrence. They assume that write sharing will be rare, if at all, and thus do not provide the necessary mechanisms to assure that it is done in the best way possible. DFS is designed specifically for such a purpose. It uses a versioning scheme to protect all users' modifications. Caching is done at the user's local machine to provide good performance. Also, files are replicated to make DFS a highly available fault tolerant system. Even though DFS was specifically designed for DCS, the concepts and design can be used in any distributed file system where write sharing is prevalent and protecting user modifications is important. The rest of this chapter evaluates the work done with respect to the overall objectives. Also any future work or ideas not implemented are presented. Finally, this thesis concludes with a brief summary.

7.2 Evaluation

In Chapter 1 the overall objectives of DFS were stated. These are reproduced below for convenience.

1. Develop a distribute file system layer that implements a shared and uniform name space within DCS,

2. Develop an interface for users and other DCS modules to efficiently interact with the shared files in DCS space,

3. Provide the necessary concurrency control that allows user's to safely and efficiently modify files,

4. Provide reliability and high availability for files within the DCS space,

5. Make the above as transparent as possible to the user.

These goals are now evaluated in light of the work accomplished in DFS. Objective 1 is achieved in DFS by use of the File Information Table (FIT). This table keeps track of all the files of a conference and allows servers and different sites to know the files that are present on each other's system. Objective 2 is achieved by providing an interface of file system commands that allows users and other servers to interact with the DCS file space. Files are also cached locally to improve performance. Objective 3 is achieved by use of a versioning scheme. Each user can access and modify the same file by caching a copy of the file at their local system and performing the modifications on that file. When the user is done making changes the file is saved back to the server as a new version that is immutable. This protects each user's modifications from interfering with others and has the added benefit in that past versions can be accessed if it is discovered that an undesirable change is made. Objective 4 is achieved by replicating files to multiple servers. This makes the system highly available and fault tolerant because it allows a user to retrieve the file on a number of other servers if one is unavailable. Finally, objective 5 is achieved hiding the details of versioning and file replication to the user. Users do not have to explicitly "check-out" or "check-in" a file. They can simply perform the normal operations of open and close. System consistency is

maintained even though this transparency may cause some user's to believe the system is inconsistent. The versioning policies can be used to control the actual version opened. Also, a user can always specify a specific version to open. The users are also unaware of where the actual file is located and which replica is being retrieved.

From the work presented in this thesis, which is briefly summarized in this section, it can be seen that the main objectives of DFS are achieved. However, the next section describes any future work or unexplored ideas that should be considered when developing the next distributed file system for DCS.

7.3 Future Work

Since designing and implementing a distributed file system is a large endeavor, some important aspects could not be implemented in this version of DFS. The work in this thesis designs and implements the foundation for most future designs. The following is a list of any unimplemented aspects or ideas to improve performance that are not yet explored.

1. Provide a command for users to merge the versions of a file. This should be done using the Decision Support Services (DSS) template and Notification Services (NTF) to inform the involved parties and allow them to vote if the merge should take place.

2. Provide different modes in which files can be opened. Currently, each file has an attribute that can indicate the modes in which it can be opened. However, this attribute is not used in the current implementation. For example, the owner can declare that certain file be read-only. This would prevent other users from creating multiple versions of this file. Also, if a file is opened in read-only mode, the close command would not have to attempt to write back the file as a new version.

3. The open command can possibly be redesigned to look in the user's cache for their last modified version of a file, if the last modified version by user policy is selected. This is possible because when a user opens a file it is always cached at their machine. When the file is closed the modified version is sent back to the server. Currently the next open call to the file would go to the server to get the file. However, if the last modified by user policy is selected, then this version already exists in the user's cache (if it is not deleted). The open could use this file instead

of going to the server. This would improve performance and allow disconnected operation. If the file is not in the user's cache, nothing is lost because the open can still go to the server.

4. The copying and moving of files between conferences is inefficient if the two conference servers are actually located on the same machine of use a shared file system. Currently this is not assumed and every file that is copied or moved is done by writing these files to the other servers by calling their remote methods. The efficiently suffers greatest when a directory and all its contents must be copied or moved.

5. Provide some default file replication. Currently a file is only replicated when it is read at a site on which it does not exist. However, if a server becomes unavailable before a file is access by another site, that file will be unavailable to all sites. Future designs should consider replicating all files to a default number of sites.

6. Completely integrate DFS with Conference Control Services (CCS), Database Services (DBS), Access Control Services (ACS), and other DCS modules. Due to the delays encountered because of bugs in the CCS and DBS code, this could not be fully accomplished.

7. Another possible design of DFS that was not explored is to make use of the Linux Virtual File System layer. In file systems such as the Network File System (NFS), Andrew File System (AFS), and Coda, this layer intercepts the system calls made by users and applications. If these calls are made in remote directories, it routes them to the file system client, which can then sends it to the server.

7.4 Summary

This thesis describes the design and implementation DFS within DCS. This is an important module of DCS because it provides the mechanisms for users to share and access files. Even though some improvements can be made, it successfully implements a distributed file system for DCS, where users can be assured that their work and files are safe and protected.

LIST OF REFERENCES

[1] Frederick P. Brooks, Jr., *The Mythical Man-Month*, Anniversary Edition, Addison-Wesley Publishing Co., Reading, Massachusetts, 1995.

[2] Randy Chow and Theodore Johnson, *Distributed Operating Systems & Algorithms*, Addison-Wesley, Reading, MA, 1997.

[3] Vijay Manian, "Access Control Model for Distributed Conferencing System," Master's Thesis (M.S.), University of Florida, 2002.

[4] Abraham Silberschatz, Peter Galvin, and Greg Gagne, *Operating System Concepts*, Sixth Edition, John Wiley & Sons, Inc., NY, 2002.

[5] R. E. Newman, C. L. Ramire, and H. Pelimuhandiram, M. Montes, M. Webb, and D. L Wilson, "A Brief Overview of the DCS Distributed Conferencing System," Proceedings of the Summer USENX Conference, USENIX, 1991, pp. 437-452.

[6] Ling Li, "File Services for a Distributed Conferencing System (DCS)," Master's Thesis (M.S.), University of Florida, 1995.

[7] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz, "NFS Version 3 Design and Implementation," Proceedings of the Summer USENIX Conference, USENIX, 1994, pp. 137-151.

[8] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Thurlow, "The NFS Version 4 Protocol," Proceedings of the 2nd International System Administration and Networking Conference (SANE2000), Maastricht, The Netherlands, May 2000.

[9] Mahadev Satyanarayanan, "A Survey of Distributed File Systems," Annual Review of Computer Science, Vol. 4, 1990, pp. 73-104.

[10] Mahadev Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," IEEE Computer, Vol. 23, No. 5, May 1990, pp. 9-21.

[11] Mirjana Spasojevic and Mahadev Satyanarayanan, "An Empirical Study of a Wide Area Distributed File System," ACM Transactions on Computer Science, Vol. 14, No. 2, May 1996. pp. 200-222.

[12] Mahadev Satyanarayanan, and Mirjana Spasojevic, “AFS and the Web: Competitors or Collaborators,” Proceedings of the Seventh ACM SIGOPS European Workshop, Connemara, Ireland, 1996.

[13] Mahadev Satyanarayanan, James Kistler, Puneet Kumar, Maria Okasaki, Ellen Siegel, and David Steere, “Coda: A Highly Available File System for a Distributed Workstation Environment,” IEEE Transactions on Computers, Vol. 39, No. 4, April 1990, pp. 447-459.

[14] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, “The Sprite Network Operating System,” IEEE Computer, Vol. 21, No. 2, Feb. 1988, pp. 23-36.

[15] D.J. Santry, M.J. Feeley, N.C. Hutchinson, and A.C. Veitch, “Elephant: The File System that Never Forgets,” Proceedings of the Seventh Workshop on Hot Topics in Operating Systems, Rio Rico, AZ, Mar. 1999, pp. 2-7.

[16] R.Y. Wang and T.E. Anderson, “xfs: A Wide Area Mass Storage File System,” Proceedings on the Fourth Workshop on Workstation Operating Systems, Oct.1993, pp. 71 -78.

[17] David Patterson, Garth Gibson, and Randy Katz, “A Case for Redundant Arrays of Inexpensive Disks (RAID),” ACM SIGMOD Conference Proceedings, Chicago, IL, June 1-3, 1988, pp. 109-116.

[18] “Red Hat Linux 7.3: The Official Red Hat Linux Customization Guide,” Red Hat Linux Inc., NC, 2002.

[19] V. Ambriola, L. Bendix, and P. Ciancarini, “The Evolution of Configuration Management and Version Control,” Software Engineering Journal, Nov. 1990, pp 303-310.

[20] B. Berliner, “CVS II: Parallelizing Software Development,” Proceeding of the Winter 1990 USENIX Conference, Jan. 1990, pp. 341-352.

[21] Walter F. Tichy, “RCS-A System for Version Control,” Software-Practice & Experience, Vol. 15, No. 7, July 1985, pp. 637-654.

[22] B. O’Donovan and J. Grimson, “A Distributed Version Control System for Wide Area Networks,” Software Engineering Journal, Vol. 5, Issue 5, Sept. 1990, pp. 255-262.

[23] Brad Miller, Curtis Olsen, and Dave Truckenmiller, “Distributed Concurrent Version System,” CSci 8101 Distributed Systems, Nov. 1993.

[24] A. Carzaniga, “Design and Implementation of a Distributed Versioning System,” Technical Report, Politecnico di Milano, Italy, Oct. 1998.

[25] Ashish Bhalani, "Implementation of Conference Control Service in Distributed Conferencing System," Master's Thesis (M.S.), University of Florida, 2002.

[26] Amit Vinayak Date, "Implementation of Distributed Database and Reliable Multicast for Distributed Conferencing System Version 2," Master's Thesis (M.S.), University of Florida, 2001.

BIOGRAPHICAL SKETCH

Philip Yeager was born in Columbus, Ohio, in 1980. He is the son of Eddie and Linda Yeager and the younger brother of Andrea Lenzer. He completed high school at Rutherford High School in Panama City, Florida, in 1998. He received a Bachelor of Science in computer engineering from the University of Florida (UF) in 2002. He will receive a Master of Science in computer and information science and engineering from UF in 2003. Upon graduation, he plans to pursue a career in industry. However, he hopes to eventually receive a Ph.D. and teach at a university or community college.