A DISTRIBUTED FILE SYSTEM (DFS)

By

PRASHANT JAYARAMAN

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

2006

To my parents and to my brother Vivek.

ACKNOWLEDGMENTS

I thank Dr. Richard Newman for his help and guidance. I would like to express my gratitude to Dr. Randy Chow and Dr. Shigang Chen for serving on my supervisory committee and reviewing my work. I also acknowledge the contributions of all members of the Distributed Conferencing System group. Special thanks are due to my family and friends for their encouragement and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

Causally related: Two events are causally related if the order of their occurrence affects their result.

Causal consistency: A model specifying that operations that are potentially causally related must be seen by everyone in the same order. The order of other operations is unimportant and they are assumed to be concurrent.

Client: Process that accesses a (remote) service on another computer through a network A user uses a client to access the services provided by the servers.

Coherence: A property of distributed systems in which the file is consistent across all its replicas. The consistency is defined by the consistency model that is used.

Concurrency control: The manner in which concurrent access/modification of objects is handled. Time stamping, locking and optimistic concurrency control are forms of concurrency control.

Consistency: A system is consistent if operations on objects follow specific rules. The system guarantees that upon following the rules, the objects will be consistent and the results will be predictable.

Consistency model: The set of rules that specify when an object can be classified as consistent.

DCS aware application: An application that can take advantage of DCS services. It is designed specifically with DCS in mind.

DCS conference: Set of users, with a set of roles, 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. A conference may be spread over several sites. Each site contributes resources to the conference.

DCS file space: File space managed by Distributed File System presented in this thesis. It is only accessible through the services provided by this module.

DCS instance: A collection of sites and conferences.

DCS server: Server modules of the various DCS services. A site may run several servers belonging to different DCS conferences.

DCS site: Collection of servers running DCS services. A site may be a part of several conferences. 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 unaware application: Any application that is not DCS aware. Almost all normal applications fall in this category.

Delta: A file containing the difference between one file and other. It is also called a diff.

Delta compression: A way of storing data in form of differences between sequential data rather than as complete files. A file is stored as the difference between itself and a base file. Using the base file and the difference, the given file can be rebuilt.

Immutable semantics: A model in which all objects are immutable. Any change to an object is stored as a new object.

Location independence: A resource can be moved about the system without changing its name.

Location transparency: Names used to identify resources do not reveal the location of the user or the resource.

Object: Any file that exists within DCS and is managed by DFS.

Role: A named set of capabilities. Roles can also be viewed as a collection of job functions. 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.

Session semantics: A consistency model that dictates that changes to a file are visible only after the process closes the file.

Stateful server: A server that maintains information about its clients. It is more efficient than a stateless server.

Stateless server: A Server that does not store information about its clients. This allows for a very robust design.

Transparency: Refers to hiding irrelevant system dependent details from the user.

Uniform name space: Everybody invokes a resource using the same name. In the context of DFS, this means that every client uses the same path to access the file.

Unix semantics: A model specifying that operations on a file are immediately visible to all users/processes.

User: An entity (usually human) in the system that performs actions on objects through services provide by the servers.

User file space: File space that exists on the user's local system. It is not managed by DCS. However, a DCS client often acts as a go-between.

Version chain: A sequence of file versions in which, the first version is stored as a full file but all others are stored as deltas.

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 (DFS)

By

Prashant Jayaraman

May 2006

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

Distributed Conferencing System (DCS) is a platform that supports collaboration in a distributed environment. Distributed File System (DFS) provides the file system of DCS. Distributed file systems face some peculiar challenges. They must provide a uniform name space for users, along with location independence and location transparency. A file can be edited simultaneously at different locations. If the changes are not coordinated, the file may become inconsistent and the resulting unpredictability can be frustrating to users. An important characteristic of a distributed system is that at almost all times, some node or another will be unavailable. DFS must ensure that the system is reliable and highly fault-tolerant.

Currently available distributed file systems cannot support DCS. They are not designed for an environment where a high level of concurrency is common. They assume that simultaneous writes are unlikely and do not support an adequate level of concurrency control. Most file systems make use of primary copy replication which has a large overhead. Others use read-only replication.

DFS supports a high level of concurrency. It uses an immutable versioning scheme where, every time a file is closed, it is saved as a new version of the file. This allows multiple users to edit and save the file simultaneously, without the danger of interleaving file blocks. At the same time, it makes sure that no changes are lost. DFS provides fault tolerance and ensures high reliability by replicating files across multiple sites. In addition, it supports file migration. The immutable versioning scheme also gets rid of the problem of synchronizing replicas.

CHAPTER 1
INTRODUCTION

Introduction

In the modern world, there is a growing need for collaboration among geographically separated groups. Platforms and services that assist such collaboration are in great demand.

Distributed Conferencing System (DCS) is an effort to provide a platform for users to work cooperatively in real time in a distributed environment. The system is based on a robust, hierarchically distributed architecture and provides access control, file service, notification, secure communication and mechanisms for group decision-making [1]. DCS provides basic distributed collaborative tools (text editors, graphics editors, etc) and can also support third-party applications. The architecture of DCS is designed to be scalable. Another major design goal is fault-tolerance. The system is designed to handle network failures and system crashes by restoring from backups and contacting other sites for the most up-to-date information.

File management is important in any distributed system. File systems in distributed environments have to tackle issues not seen elsewhere. A key question is how are files and directories presented to the user [2]? Another question is what happens when several users modify the same file at the same time [2]. It is also important to ensure that data is not rendered unavailable by the failure of a few systems [3]. File permissions are also significantly more complicated [2]. Finally, performance should be one of the goals of a distributed file system [2].

Distributed File System (DFS) provides file-handling services for DCS users. It is designed to allow file sharing and concurrent access of files. It also provides transparency [2]. In addition, it uses the Access Control Service provided by DCS to enforce file access permissions [4].

Distributed Conferencing System Services

The first version of DCS (DCS v1) provided only limited services for focused tasks. One of its drawbacks was that it had very few roles and only users in the voter role could vote on decisions [1]. Also, it relied on UNIX permissions for security [5].

The second version (DCS v2) provides more services and better communication primitives. In addition, it supports more voting mechanism and limited Role Based Access Control (RBAC) [4]. DCS v2 modules have better support for file/object types. The major services provided by DCS are described below [6].

- **Conference Control Service (CCS):** This module manages the Collaborative Group (CoG). It is the first service to be activated and instantiates all other modules. It uses secure messaging to allow users to login and interact with services. This module also handles operations like splitting and merging instances/CoGs/sites, access control requests and add user requests. It is responsible for creating and deleting CoGs.

- **Database Service (DBS):** DBS maintains all tables in DCS space. It makes use of a Database Managements System (DBMS) as the backend. Tables are stored as partially replicated, distributed databases and group multicast is used to ensure eventual consistency.

- **Notification Service (NTF):** NTF provides asynchronous event notification to registered users. In addition to built-in events, NTF allows users to define new events. NTF maintains a global and local database to match events to registered users, along with the delivery method.

- **Decision Support Service (DSS):** DSS facilitates the resolution of issues by a group of people with the joint responsibility for them. It maintains decision templates for that purpose. It allows creation, modification and execution of templates. If a template requires a vote among a set of users, DSS will contact the users, get their votes and return the result.

- **Access Control Service (ACS):** ACS is responsible for maintaining access rights and protection information for all objects in DCS. Instead of just maintaining attributes, ACS supports the use of decision templates to allow members to decide on actions. Applications can register new access types for more fine grained control. ACS makes use of DBS to maintain state.

- **Secure Communication Service (SCS):** SCS allows DCS and its users to communicate securely and reliably. It ensures authenticity, confidentiality and integrity of all messages. It is responsible for the creation and maintenance of keys and certificates for sites and users.

- **Application Manager (AM):** The Application Manager is responsible for registering and invoking applications that are available for each CoG. The applications could be DCS aware or DCS unaware. AM maintains a list of applications for each CoG and makes them available to the user.

- **Distributed File System (DFS):** This is the focus of the thesis. It manages all files in the CoG. It provides concurrency control using a versioning scheme with immutable files. It also provides replication and migration services.

CHAPTER 2
STUDY OF DISTRIBUTED FILE SYSTEMS

Introduction

This chapter surveys some of the popular distributed file systems. A distributed file system is a file system that supports sharing of files and resources in the form of persistent storage over a network [2]. The first distributed file systems were developed in the 1970's. The most popular distributed file systems are Sun's Network File System (NFS) and the Andrew File System (AFS). In addition to a discussion on these file systems, this chapter will also provide an overview of some less popular systems.

Network File System (NFS)

Introduction

The NFS protocol was developed by Sun Microsystems in 1984 [7]. It was meant to be a distributed file system which allows a computer to access files stored remotely, in the same manner as a file on the local hard drive. NFS is built on the Open Network Computing Remote Procedure Call (ONC RPC), described in RFC 1831 [8]. Version 2 of NFS was stateless and ran over UDP [9]. Version 3 introduced support for TCP [10]. Finally, version 4 introduced a stateful protocol [11, 12]. Though NFS is strongly associated with UNIX, it can be used on Windows, Mac and Novell Netware.

Protocol

NFS provides its services through a client-server relationship. The client and server communicate through RPCs. The client issues an RPC request for information and the server replies with the result. NFS was designed to be machine, operating system,

network architecture and protocol independent. External Data Representation (XDR) is used to translate between machines with different byte representations.

The computers that make their file systems, or directories, and other resources available for remote access are called servers. The act of making file systems available is called exporting. The computers, or the processes they run, that use a server's resources are considered clients. Once a client mounts a file system that a server exports, the client can access the individual server files (access to exported directories can be restricted to specific clients). It must be noted that a computer can be both a server and a client at the same time. An NFS server is stateless. It does not have to remember any transaction information about its clients. In other words, NFS transactions are atomic and a single NFS transaction corresponds to a single, complete file operation. NFS requires the client to remember any information needed for later NFS use. This has meant that NFS has poor performance over Wide Area Networks (WAN). However, Version 4 has introduced stateful protocols and compound procedures to improve this situation [2].

Naming and Transparency

A client can mount the exported directory (from the server) anywhere on its local file system. The purpose of this is to provide clients transparent access to the server’s file system.

Figure 2-1 illustrates the fact that different clients can mount the exported directories at different locations. In this case, the client Client 1, mounts the directory Dir A (from Server 1) at /mnt and the directory Dir B (from Server 2) at /mnt/remote. By contrast, Client 2 mounts Dir B at /mnt and Dir A at /mnt/remote.

An NFS server can mount directories that are exported by other NFS servers. However, it cannot export these to its clients. The client will have explicitly mount it

from the other server. On the other hand, a file system can be mounted over another file system that is not a local one, but rather is a remotely mounted one.

NFS supports location transparency but not location independence. This is due to its decision not to enforce a common namespace. This makes it harder to share files.

Concurrency Control and Consistency

The NFS protocol does not provide concurrency control mechanisms. A single read or write call may be broken down into several RPC read or writes because each RPC call can contain up to 8KB of data. As a result, two clients writing to the same remote file may get their data intermixed.

It is hard to characterize the consistency semantics of NFS. New files created on a machine may not be visible anywhere for half a minute. It is indeterminate whether writes to a file are visible to other sites that have the file open for reading. New opens of a file observe only the changes that have been flushed to the server. Thus, NFS does not provide either UNIX semantics or session semantics.

Replication and Migration

NFS version 4 provides limited support for file replication. It requires that an entire file system be replicated, including the files, directories and attributes. One of the attributes of a file, contains the list of alternate locations (DNS name or IP address) for the file system containing the file. In order to access files from a server, the client must switch over to that server. It must be noted that the NFS protocol does not specify how replicas are kept up-to-date, each implementation must decide for itself on the best manner to achieve this.

As explained previously, NFS does not provide location independence. This means that migrating a file system from one server to another would invalidate all mounts of that file system.

Caching

NFS version 3 mainly left caching out of the protocol specification. Most implementations never guaranteed consistency. The time lag, between the time a file was written to the server, and the time the file was invalidated on the client cache varied from a few seconds to almost half a minute. NFS version 4 does try to alleviate this problem.

Most clients cache file data and attributes. On a file open, the client contacts the server to check whether the attributes need to be fetched or revalidated. The cached file blocks are used only if the corresponding cached attributes are up-to-date. Cached attributes are generally discarded after 60 seconds. NFS uses both read ahead caching and delayed write, to improve performance.

Security

Security, in distributed file systems has two parts – secure communication between the computers and controlling access to files.

NFS is built on top of RPC's. Hence, secure communication boils down to the establishment of a secure RPC. Older versions of NFS had support for Diffie-Hellman key exchange for establishment of a secure key. Recent versions support Kerberos (version 4) for establishing secure RPC's. NFS version 4 also provides a general framework that can support numerous security mechanisms to ensure authenticity, integrity and confidentiality.

NFS provides a much richer set of access controls than the UNIX file system. Access control is implemented in the form of access control entries, where each entry

specifies the rights for a user or a group of users. Unlike UNIX, it is possible to specify the rights of several users and groups.

Andrew File System (AFS)

Introduction

The Andrew File System (AFS) was initially developed by Carnegie Mellon University (CMU) as part of the Andrew Project [13, 14, 15]. It is named for Andrew Carnegie and Andrew Mellon, the founders of CMU. Its biggest advantages are security and scalability. AFS has expanded to over 1000 servers and 20000 clients worldwide [16]. It is believed that AFS can scale up to support twice as many clients. AFS heavily influenced Version 4 of Sun Microsystem's NFS. Additionally, a variant of AFS was adopted by the Open Software Foundation in 1989 as part of their distributed computing environment. There are three major implementations of AFS - Transarc (IBM), OpenAFS and Arla. AFS is also the predecessor of the Coda file system.

Architecture

Like NFS, AFS uses RPC's as the basis of communication between clients and servers. AFS can be divided into two components. The first is a group of relatively small number of centrally administered file servers, called **Vice** and the second consists of a large number of workstations called **Virtue**. Virtue workstations provide access to the file system. Every client (Virtue workstation) has a process running on it, called Venus. This allows the client to access the shared file system in the same manner as the local one.

AFS defines a volume as a tree of files and subdirectories. Volumes can be created by administrators and linked to specific paths (similar to mount in NFS). Volumes can be

located anywhere and replicated or moved without informing the user. In fact, a volume can be moved while its files are in use.

Naming and Transparency

AFS provides its clients with a global name space. In addition, clients have a local name space to store temporary data. From the client's point of view, the file system appears as one global resource. It is not possible to deduce the location of the file from its name. In addition, all files have the same path from every client. Thus, AFS provides both location transparency and location independence.

Concurrency Control and Consistency

Like NFS, AFS uses the Ostrich approach (it is widely and incorrectly believed that when threatened, the Ostrich buries its head in the sand, hoping that its attacker would go away). It does not make any effort to resolve concurrency, believing that shared writes are extremely rare. When two users close the same file at the same time, the results are unpredictable.

AFS supports session semantics. This means that once client A has saved the file, any client opening the file after A has closed it, will see the modifications made by A. This is weaker than UNIX semantics but much easier to support in a distributed environment.

Replication and Migration

AFS provides replication to read-only backup copies. Even an entire volume can be replicated. A client system looks to retrieve the read-only file from a particular copy. In case the copy is not available, it switches to a different copy. This allows both better fault tolerance and improved performance. AFS allows the creation and migration of copies as

needed. By ensuring that the copies are read-only, replication consistency problems are avoided.

Caching

AFS divides files into chunks of up to 64 KB, for caching. A large file may have several chunks, while a small one may occupy just one. The cache tries to establish a "working set" of files (copied from the server) on the client. In case of server failures, the local copy is still accessible.

Read and write operations are performed on the local, cached copy. When the file is closed, the modified chunks are copied back to the file server. Callback is a mechanism used to maintain cache consistency. When the client caches a file, the server makes a note of it (stateful) and informs the client when a change is made to the file. The cache is believed to be consistent while the callback mechanism is operational. Callbacks must be reestablished after client, server and network failures.

Security

Security is one of the strengths of AFS. It uses a secret key cryptosystem to establish a secure channel between Vice and Virtue. The secret key is used by the two machines to establish a session key, which is used to setup the secure RPC.

The process of authenticating the user is more complex. The protocol used for this purpose, is derived from the Needham-Schroeder protocol [17]. An Authentication Server (AS) provides the user with appropriate tokens. These are used to establish his/her identity.

Access control lists are used to control permissions to files and directories. The Vice file servers associate lists with directories only. Files do not have access control

lists. This makes it simpler and more scalable. AFS supports both users and groups. It is also possible to list negative rights.

Common Internet File System (CIFS)

In its original incarnation as Server Message Block (SMB), it was invented by IBM to provide a genuine networked file system. Microsoft modified it heavily, adding support for symbolic links, hard links and large files and launched an initiative to rename it as CIFS.

CIFS can run on top of both NetBIOS and TCP/IP protocols. In the manner of other distributed file systems, CIFS uses a client-server model for communication. In contrast to other file systems discussed, CIFS also provides Inter-process communication (named pipes). In addition, SMB servers can share other resources besides file systems.

CIFS cannot really be described as a legitimate distributed system. It merely provides a way to access resources over a network. It does not provide either location transparency or location independence. Resources are identified as //server-name/path-to-resource. Unlike other distributed file systems, it makes little effort to remain agnostic of the underlying operating system. There is no concurrency control or caching. It does not support file replication or migration. It does provide UNIX consistency semantics because the file has only one copy.

Coda

Coda is another distributed file system, developed at Carnegie Mellon University [18]. It is based on the Andrew File System. Some of the areas that it improves over AFS are listed below.

In contrast to almost all other distributed file systems, Coda allows the client to work, even when it is unable to contact any server. In this case, the client uses the local

(cached) copy of the file. It is possible to ensure that the client fills its cache with the appropriate files before disconnecting from the server. This is called hoarding.

Replicated volumes do not have to be read-only. Coda uses a replicated-write protocol to ensure that the modified parts are transferred to all servers at the same time. In case server failures, different servers will have different versions of the file. Version vectors (similar to vector timestamps) are used to resolve this inconsistency.

The security mechanisms of Coda are interoperable with Kerberos. Also, it has native support in the Linux 2.6 kernel.

Plan 9

Plan 9 is based on the idea of using a few centralized servers and numerous clients [19]. Plans 9 allows computer to share any resource in the form of files. For example, the network interfaces of a machine can be exported as a bunch of directories. Each TCP connection is represented as a subdirectory of the corresponding network interface.

Plan 9's naming philosophy is similar to that of NFS with some interesting deviations. Multiple name spaces can be mounted at the same mount point. In this case, the contents of the two remote file systems are Boolean ORed. Such a mount point is called a union directory.

Plan 9 implements UNIX file semantics by ensuring that all updates are always sent to the server. Concurrent writes will happen in an order determined by the server but unlike session semantics, the writes will never be lost. However, this reduces performance.

All files have four integers identified with them. Two of these – type and device number, identify the server or device hosting the file. Two others, path and version,

identify the file relative to the server or device. Version is incremented after every modification.

Plan 9 provides some support for replication and caching. Clients can cache files and use their version numbers to ensure that the files have not been modified. Users are authenticated using a protocol similar to the Needham-Schroeder protocol. File permissions are identical to those in UNIX. Plan 9 has the notion of group leaders. Only group leader can modify the group's permissions on the file.

xFS

xFS is an unusual serverless file system [20, 21], designed to operate on a local area network. It distributes storage, cache and control over cooperating workstations. This arrangement is believed to yield higher performance, reliability and scalability.

xFs defines three types of processes. Storage servers stored parts of data. Metadata managers store information on locating data and clients form the interface between users and the file system. The storage servers together, implement a virtual RAID device.

xFS uses system wide identifiers to locate metadata managers. Clients cache blocks of data rather than files and the cache consistency scheme is similar to AFS. Each data block belongs to a manager. When a client wishes to modify a block, it obtains write permission from the manager. The manager invalidates all other cached copies of the file. Write permission is revoked when another client wants to modify it. This also requires that the block be flushed and written.

xFS does not emphasize security. It provides the usual access control lists and requires that clients, managers and storage servers run on trusted machines.

Low Bandwidth File System (LBFS)

LBFS is designed for networks with low bandwidth and high latencies [22]. LBFS works on the concept that a version of a file has much in common with its previous version. It also believes that there are similarities between files created by the same application. The techniques used by LBFS can be used in conjunction with the techniques used by other distributed file systems (like Coda) to improve tolerance to network vagaries.

LBFS works by maintaining a large cache at both the client and the server. Files are divided into variable size chunks. The chunk boundaries are determined by Rabin fingerprints [23], subject to lower and upper bounds. A Rabin fingerprint is the polynomial representation of the data modulo a predetermined irreducible polynomial. When the low-order 13 bits of a region's fingerprint equal a chosen value, the region constitutes a breakpoint. Assuming random data, the expected chunk size is $2^{l3} = 8192 =$ 8 KBytes . This scheme ensures that modifications to a chunk affect only that chunk and its neighbours.

LBFS indexes chunks using their SHA-1 hash [24]. In case the chunk is present in both the server and the client, only its hash is sent. Otherwise, the entire block is transmitted. All data sent on the network is compressed using gzip.

LBFS provides the same session semantics as AFS. Files are flushed to the server on close. LBFS uses temporary files to implement atomic updates. All writes to the file happen on a temporary file which is then renamed. This ensures that concurrent writes can never result in inconsistent files. This is in marked contrast to NFS. The client that closed the file last will overwrite the changes of the others.

Every server has a public key, which is specified on the client when it mounts the server. Future versions look to incorporate self authenticating pathnames [25], to facilitate auto-mounting. All communication between the client and the server is encrypted with the help of a session key.

Freenet

Freenet is a decentralized, anonymous, censorship resistant, distributed data store [26]. Unlike most distributed file systems, it is based on the peer to peer communication model. Freenet allows all participating computers to contribute bandwidth and storage space. It allows users to publish and retrieve data anonymously.

Files have a descriptive string associated with them. This yields a public/private key pair. The private key is used to sign the file and the hash of the public key is called the Keyword Signed Key (KSK). Signed Subspace Key (SSK) allows the use of personal namespaces and is created in a manner similar to KSK, using usernames instead of file descriptions. A combination of the username and descriptive string is sufficient to access the file. Content Hashed Keys (CHK) are created by hashing the contents of a file. A Content Hashed Key can also be used to access a file. A CHK points directly to a file, but the KSK and SSK combination uses an indirection mechanism.

Freenet uses a routing table similar to the Internet Protocol. The client retrieves the file by specifying its binary key. It then checks whether it has knowledge of the node that should be contacted to get this file. If no node is found, it looks up the nearest key (lexicographic) in its table and forwards the request to that node. This proceeds until the key is located or the TTL expires. In order to preserve anonymity, nodes along the successful path randomly designate themselves as sources of the file and may cache a copy of the file.

To insert a file, the user computes a binary key for the file and makes a request for a file with that key. In case there is no collision, the file is inserted into the node and all nodes along the path of the previous query are informed of the new key. Again, nodes can randomly re-designate themselves as hosts. Each file in Freenet is encrypted using a random key that is published along with the binary key.

The routing heuristic and insertion algorithm ensure that files with similar keys are clustered. This also ensures better performance of both of them. Freenet is closer to an anonymous bulletin board that a distributed file system. It has been included here because of its unusual design and goals.

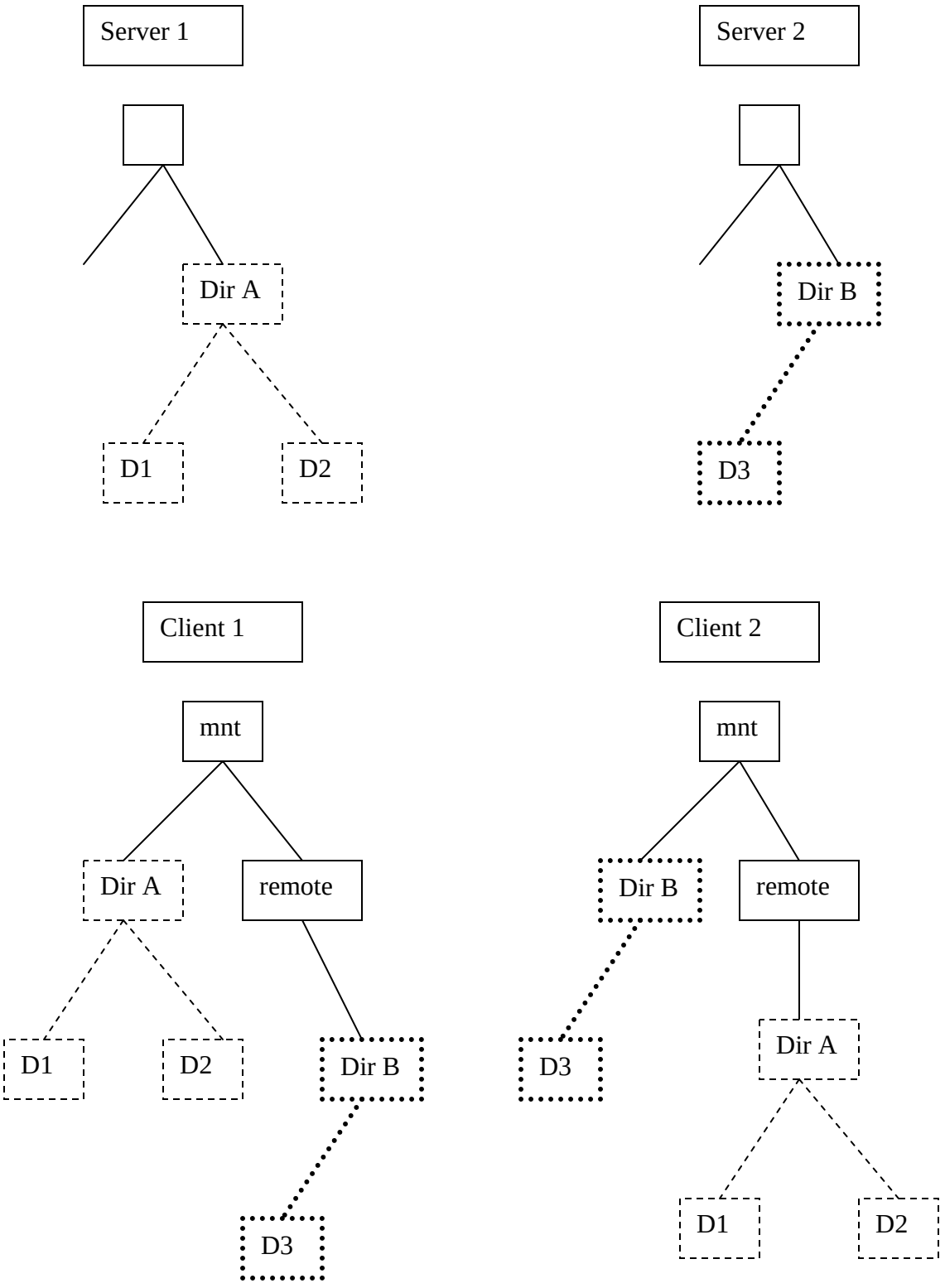

Figure 2-1. Example of NFS clients mounting the same directories at different mount points.

CHAPTER 3
REQUIREMENTS AND DESIGN

This chapter presents the requirements of Distributed File System (DFS) in detail and goes on to describe the design of DFS. In each section, it details the approach taken in DFS along with the reasons for it.

Requirements

DCS is designed to simplify collaboration among geographically separated users. DFS provides the underlying file system for DCS. Other modules of DCS interact with DFS to provide the complete user experience. So, the requirements were formed on the basis of discussion among the members of the DCS Research Group

Naming and Transparency

DCS users can reside in different locations and the file system should allow users to share files. This requires a conference-wide shared name space. In addition, clients and servers will often store data related to them. This requires that there be a local name space, in addition to the conference-wide name space. So, DFS must provide both local and conference-wide name spaces.

Users join DCS conferences using Conference Control Services (CCS) [27]. It is possible for a user to join multiple conferences through CCS. DFS must present the user with a hierarchical view of the file system.

DCS conferences span several sites. It is very likely that files are spread across multiple sites. DFS should ensure that the user does not have to inform it about the location of the file. In other words, filenames should be location transparent.

The name of the file should not depend on the client that is being used to access the file. This ensures that users have the freedom to connect from any client and that they don't have to remember the mount points of that client.

Concurrency Control and Consistency

Concurrency control is very important in a distributed file system. DFS should be designed to allow concurrent access as best as possible. Users must be able to access and write files concurrently. It is up to DFS to ensure that changes made by one user do not interfere with the changes made by others.

Most distributed file systems provide an approximation of session semantics. It would be highly desirable for DFS to provide the same consistency semantics. New files should be seen when the client closes the file.

Replication and Migration

Fault tolerance and high availability are two of the goals of DFS. In order to achieve these goals, DFS must provide file replication. If a site is unavailable, DFS should be able to retrieve the file from its replicas. DFS should also enforce replica transparency [28]. This means that normally, the client should be unaware of the existence of replicas.

DCS is designed to spread across Wide Area Networks (WANs) and different administrative domains. Replicating a file on a server close to the client can decrease access times dramatically. Migration also helps DFS improve performance and availability. DFS should allow migration from far away or unreliable servers, to closer, more reliable servers.

Caching

As mentioned previously, DCS is designed for WANs. This introduces the problem of network latency. DFS must use caching aggressively to reduce access times. DFS should also be aware of the fact that while servers are connected through WANs, the connection between a client and a server is often over a Local Area Network (LAN). If both the server and the client maintain a working set of the files that they use, it is possible to reduce the traffic over WANs.

Security

DCS is spread across multiple administrative domains. DFS must ensure that it makes it as hard as possible for an intruder to play havoc with the system. This requires that all communication between the client and the server (and between the servers themselves) be protected. Authentication, integrity verification and confidentiality must be provided by the communication service.

Another part of security is to regulate the behaviour of legitimate DCS users. DFS must have very flexible and robust permission schemes. In addition, it should allow users to create new permission types. This ensures that DFS is able to handle scenarios that were not envisaged during design and development.

As described in Vijay Manian's work on DCS access control, clients are bound to roles [4]. Each role has specific privileges. One of the desired privileges should be viewing the contents of a directory. Only those files, for which the user has the privileges, should be shown in the directory listing. This can be used to give each a user a different view of the same directory.

Design

Communication Architecture

The design of DCS was made around the idea of separate clients and servers. DFS, as a module of DCS has to maintain that architecture. DFS has dedicated servers for file management and file services. Client processes run on client machines and pass on requests to DFS servers.

It must be noted that DCS does not preclude the client and server being on the same machine and DFS must not impose this requirement. It only requires that the client and server be different processes. DFS servers and clients must be able to run on the same machines as the clients and servers of other DCS modules. The current design of DFS allows a site to run multiple instances of DFS, one for each conference that it participates in. It also allows servers to connect to a cog on the fly.

Stateful/Stateless Server

As previously mentioned, stateless servers are simpler to design and have better fault tolerance as compared to stateful servers. They can be easily plugged in and taken out of a network. Stateful servers are often shutdown in a ‘dirty’ state and must restore themselves to a consistent state. For these reasons, DFS servers do not maintain state information.

File Access Model

File Access Model refers to the manner in which the file system lets the client access the file. Distributed file systems offer one of the following file access models

- **Remote access**: In this model, all operations are performed directly on the remote file. NFS is one file system that offers this type of access.
- **Upload/download**: All files are downloaded by the client when opened and are uploaded back to the server when the client closes the file.

The upload/download model lends itself more easily to caching and is quite popular among distributed file systems. DFS uses the upload/download model. All files are downloaded to the client for use and are uploaded back to the server when they are saved or closed.

Concurrency Control and Consistency

Concurrency is very important in a distributed file system like DFS, which is meant for Wide Area Networks (WANs). NFS does not deal with concurrency at all and other file systems like AFS and Coda have inadequate concurrency control. DFS manages concurrency with the help of an immutable versioned file system.

All files are immutable. Changes made to a file are stored as a new version of the file. This ensures that concurrent writes by multiple clients do not interleave and make the file inconsistent. Another part of concurrency control is to ensure that writes by all clients are incorporated. When multiple clients open a file and write to it, it should not be the case that only the changes made by the user who closed the file last, are incorporated while all others are lost. This is tacked by the versioning system. Every time a change is made to a file and stored, DFS stores the modified file as a new version. Changes made to the file by different users are stored as different versions. Eventually, DFS will offer tools for automatic merging and conflict resolution.

Versioning in DFS is different from versioning systems such as CVS and Perforce. Versioning systems are designed to function as depots (usually code). They do not provide transparent access. All files must be explicitly checked out and check in. They are not optimized for the high number of requests that a file system must handle.

In DFS, for a user to see changes made by others, the user has to specify the version, after it has been saved by the other user. This peculiarity of DFS makes it harder

to evaluate its consistency semantics. Other immutable file systems do not maintain a versioning scheme and so, they seem to follow session semantics. In DFS, if the version is specified properly, it is possible to look at the changes made by the other user. Thus, from the user's point of view, the file system follows session semantics.

Versioning

It has been discovered with versioning systems that consecutive versions of a file have a lot in common. It is more economical to store just the changes between consecutive versions, rather than the versions themselves. All versioning systems maintain their files as deltas. The delta of two files is the set of changes which, when applied to the first file, yield the second file.

The use of a delta scheme will greatly reduce the storage demands of DFS. A delta compression scheme could use forward or reverse deltas. These are discussed next. At the end of the section, the final design decision is presented with reasons why it is chosen.

Forward deltas

If forward deltas are stored, only the original file is stored in its entirety. All other versions are constructed from this version using a string of deltas. The major disadvantage of this scheme is that every time the latest version is needed, it must be reconstructed step by step from all the deltas. This makes the retrieval of new files painfully slow. The SCCS versioning system uses this approach.

Reverse deltas

In a reverse delta, the latest version is stored in its entirety and all other versions are stored as deltas. The older versions can be recreated by applying the deltas on the latest version. This scheme requires that the latest version of each branch be stored as a complete file. There can be multiple ways to generate a file and the paths might be of

different lengths. This scheme uses storage space less efficiently than forward deltas. However, retrieval of the latest version is quick.

Previous design

Previous versions of DFS used a combination of forward and reverse deltas. The latest version on the main branch was kept as a complete file. All other versions could be obtained from it.

Other versions on the main branch were stored as reverse deltas of their subsequent versions. In case of branches in the versioning tree, forward deltas were used.

This scheme allowed the system to quickly retrieve the latest version of the main branch. It also reduced storage space by storing only one full version.

On the other hand, there were a few problems with the scheme. It was hard to define which branch was the main branch. Also, this scheme made file replication and migration very complex. These problems were tackled in the new design.

Current design

The new versioning system is designed with two goals in mind. The first is to reduce server load and the second is to facilitate file migration. All branches are born equal, none more equal than the others. The main branch of a version is simply the first child of the version. The tips of all branches are stored as complete files. At the same time, all deltas are forward ones. It is likely that the latest versions of a branch will be accessed far more often than the older ones. By keeping them as complete files, DFS reduces the load on the server. Also, for this scheme, both forward and reverse deltas will work equally well. However, forward deltas are more intuitive and so, are used in the scheme. Some files may be stored complete, at all times. The versioning system designates such files based on the following rules:

- The version 0 is a full file
- Any version that is not on the main branch, i.e. if it is not the first child of its parent is stored as a complete file
- Any file that not stored in exactly the same servers as its parent is stored as a complete file
- If the version chain exceeds its maximum length with this file, the file is always complete.

The first rule is obvious and needs no explanation. The second rule is to facilitate migration. Let the term last full ancestor of a version denote the last version (in the same branch), older than the given version, that has been stored as a full file. In case this rule is not applied, it would mean that besides the tip of a branch, only the root of the tree is a full file. In this case, to migrate an interior version, from one server to another, it would be necessary to migrate all versions except the tips of branches. This rule makes sure that no two branches have the same last full ancestor.

The third rule helps in keeping the replication procedure simple. Consider the consequences of not having this rule. Now, it is possible that a delta will be stored on a server and its parent will be on another server. In order to retrieve the file, both the delta and the parent are necessary. This means that both servers should be alive and this decreases availability. In order to avoid these situations, this rule is applied.

The last rule ensures that there are no long series of deltas. In the course of operations, it is likely that a complete file will be followed by a long series of deltas. To retrieve the last version, all preceding versions have to be reconstructed. Also, migration will require moving an enormous amount of data because migrating a version involves not only moving that version, but also all versions that it depends on and all versions that depend on it.

Naming and Transparency

DFS supports a conference-wide name space and a local name space. The conference-wide name space allows users to share files and directories on the file system. The local name space is used to store site specific information and temporary files.

DFS provides location transparency and location independence. It is not possible to learn the file's location from its name. In fact, the only way to ascertain the location of the file is to check the metadata. This makes it is possible to migrate a file from one server to another, without changing its name. Location Independence and location transparency allow DFS to provide a centralized view of DCS files, even when they are on totally different sites.

DFS provides hierarchically structured directories. The path of each file starts with /DCS/<conference name>. This allows client side applications to provide a global view of all conferences. It is possible to populate /DCS/ with all the conferences that the user is connected to and make it seem like /DCS/ is one directory on the local file space that provides access to conference files. Thus, client side applications can provide a global name space that merges multiple conferences. These conferences don't have to have any site in common. The files in each conference will be widely dispersed as well.

Storing Metadata

Location transparency and location independence are provided with the help of a file table, which maps a filename to its metadata and stores file metadata. This structure is also essential in order to present a uniform name space. A uniform name space also requires that the metadata on various sites be consistent.

DFS maintains multiple versions of the same file. For the sake of simplicity, let the term physical file refer to each version of a file. This term is used because, on the

operating system of the site, each version appears as a file. Another layer of the file table groups different versions together. This will be referred to, as a virtual file. Virtual files have no existence on the host operating system. A virtual file is just a collection of metadata related to every version of a file. The attributes of a physical file are listed below:

- Location of the file within DCS directory space
- Version of the file
- Type of file
- Owner of the file
- Size of the file
- Time the file was created
- Mode at which the file can be opened
- Location of replicas
- Is this a full file
- Should this file always be a full file
- Has this file been deleted
- Parent version of this file.

Currently, a virtual file maintains only four attributes

- Type of virtual file (File or directory)
- Latest version
- References to all versions (file)
- References to other logical files (directory).

The design of the file table is very important in ensuring performance, scalability and error recovery. Two options will be considered below for this purpose.

Symbolic links

One of the ways of representing this information is analogous to symbolic links. This is illustrated in Figure 3-3 and Figure 3-4. Figure 3-3 shows the physical location of conference files. Figure 3-4 illustrates the manner in which directories and links are used to provide a uniform view of the file system.

In this example, the conference Conf1 is split across two sites – Site 1 and Site 2. Figure 3-3 shows the distribution of the files among the sites. Site 1 has the files File1 and File2 along with the directory Dir1 and its contents. Similarly, Site 2 has File3, File4 and the directory Dir2.

Figure 3-4 presents the uniform namespace, as seen from the site Site 1. Site 1 does not have the files File3 and File4. In order to keep the location of the files transparent, it creates symbolic links File3 and File4. These symbolic links are presented in bold in the figure. File3 and File4 contain information that identifies them as symbolic links and also contain the location of the server that actually stores the file.

In order to access the file, the server opens the file, realizes that it is a symbolic link and obtains the location of the server that stores the file. It then contacts the server to obtain the file.

It can be seen that the directory Dir2 is not shown in bold. This is to indicate that Site 1 does create the directory Dir2, even though all its contents are symbolic links.

File Information Table (FIT)

It is also possible to organize metadata and present a uniform name space using a File Information Table (FIT). This table stores all the attributes of the file, including its path and physical location. Each site has a copy of the FIT. Upon receiving a request for a file, the server searches through the FIT to locate the file and return its physical

location. In case the file is on the same site, it returns the data. If the file is on another server, it requests the other server for the data.

The actual arrangement of files on a site does not bear any relation to the view presented to the clients. It is the task of the FIT to map the logical view to the physical location. It is not necessary to create directories or symbolic links. The FIT can be retained in memory for fast access. A site can be brought up to speed by just sending it the FIT.

Previous design

Previous implementations of DFS chose the FIT approach for its simplicity. The scheme was designed with the Databases Service (DBS) in mind [29]. DBS would keep the FIT consistent by serializing all changes. DBS uses causal order protocol to implement reliable atomic multicast, ensuring that concurrent changes are serialized in some order.

After a file is created, few of its attributes would change. The static and dynamic attributes would be maintained in different tables. Either the site that created the file or the home site of the conference can own the rows in DBS and serialize changes.

There are several advantages in using DBS to maintain FIT. First, it takes a great load off the hands of DFS. Besides, DBS already maintains tables for other modules. Unfortunately, DBS was not fully functional at that point.

This required DFS designers to implement their own methods for FIT consistency and a FIFO scheme was finally agreed upon. Here, every entry in the FIT has an owner and the owner is responsible for communicating changes to all servers. Servers must ensure that they process messages from a site, in the same order in which they were sent.

Instead of a causal order protocol, only FIFO consistency is required. The current implementation of FIT is substantially similar.

Proposed design

It can be easily seen that both DBS and FIFO have their disadvantages. Their biggest problem is that neither is particularly scalable. DBS is too restrictive and in the case of the FIFO scheme, the number of messages required to maintain consistency increases nearly exponentially. A more scalable FIT scheme is presented next.

In this scheme, every server maintains the root directory contents. The contents of other directories are maintained only if they lie in the path of a file located at this server. This means that if a server has version 2a.0 of the file /abc/def/xyz/pqr.txt, it has the contents of /abc, /abc/def, and /abc/def/xyz in its local FIT along with information about the different versions of /abc/def/xyz/pqr.txt. In order to get the listings of other directories, the server contacts other servers.

In this scheme, the virtual files (collection of versions) will have to maintain these properties.

- Type of virtual file (file or directory)
- Contents of the virtual file
- Is the virtual file maintained locally
- List of servers maintaining this virtual file.

If a directory is maintained on this server, then the attributes of all its virtual files are valid. Any change in the directory must be propagated to the list of servers. If the directory is not maintained locally, the attributes of the directory must be obtained from the list of servers maintaining this directory.

This leads to a situation in which, the directories whose contents are more widely distributed are maintained on many servers. Normally, this means that directories with more files are more heavily replicated. This does make sense because the FIT should be at least as reliable as the file replication system. Another characteristic of this scheme is that higher level directories have higher replication factors. The directory /games/pc is on at least as many servers as /games/pc/Microsoft.

Operations on the FIT are complicated by the fact that the FIT must be fetched from other servers in some cases. The various operations supported by the FIT are described below.

Getting the attributes of a directory/file. If the given directory is maintained locally, it is trivial to obtain the information from the local FIT. Obtaining the attributes from another server is more challenging.

Consider a directory /books/fiction/humour/20thCentury/English/Wodehouse. Let the server (Server A) have the directory listing up to /books/fiction. On receiving a query for the entire path, the local FIT realizes that it can resolve up to /books/fiction. As described previously, it will also have the list of servers that maintain the directory /books/fiction/humour. It passes on the path to one of them and asks that server to resolve it as much as it can. This server (Server B) might resolve it till /books/fiction/humour/20thCentury and ask Server C to resolve the rest and pass the result to Server A.

Server C will be one of the servers on the list of servers maintaining /books/fiction/humour. It will be much faster if Server A could directly contact Server C for the information. However, Server A cannot know that one server is better than

another for resolving this path. In theory, this information could be cached, so that from now on, Server C is contacted for /books/fiction/humour/20thCentury/English/Wodehouse.

The interaction between servers could be iterative or recursive. In the iterative case, Server B gives Server A the address of Server C and asks it to contact Server C for resolution. In the recursive implementation, Server B queries Server C on behalf of Server A and passes on the result to Server A. The recursive version has a cleaner interface than the iterative version. However Server A might timeout on long paths.

Creating a directory. The first step in creating a directory is to obtain the attributes of the parent directory. Then, all servers having the parent are informed that a new directory is being created and the list of servers maintaining the new directory is left empty (to signify that this is an empty directory).

Creating a new DFS file. Creating a file is much more complicated than creating a directory. The server must first check whether it has the entire path on its local FIT. If it does not have it locally, it has to obtain information on all directories in the path and then inform other servers that it will also maintain all the directories in the path of the new file. Next, it needs to store the file (version 0 of the DFS file) and inform all servers maintaining that directory, of the new file.

Replication is one of the services provided by DFS. The server (Server A), which is connected to the client creating the file, is responsible for the file. In this case, after creating and storing the file on all servers, Server A informs other servers about the existence of the file and the location of its replicas. All files with replicas must update their FIT, to include information about all directories in the path of the file. It must be

noted that Server A may not be one of the servers which store the file. This makes it significantly complicated.

Creating a new version. Normally, DFS stores a new version as a delta relative to the previous version. However, a version might be stored as a complete file if the version chain length is reached or if it is the first version of a new branch.

In case a version is stored as a delta, it has to be stored on the same servers as its parent. It is fairly simple to do this because FIT of these servers already has information on the file. On the other hand, creating a version that is to be stored as a full file is just as complicated as creating a new file.

Deleting a file. All servers that maintain the directory (the file is in) are informed to remove the file from the FIT. Next, each server that stores the file is asked to delete it. Now, these servers determine whether they need to maintain any of the directories in the path of the file. This involves determining whether the server has replicated a file in any of the directories (or their children) in the path of the deleted file. Once a server determines the directories it doesn't need to maintain, it informs the other servers maintaining those directories to remove it from the list.

Cog Policy

All DCS services operate at the cog level. It is imperative that they provide means to set policies that affect the entire cog. This gives administrators a way to broadly define policy. DFS allows users to set CoG-wide policies, which allow a coarse level control over its operations. Cog policies can also be used to set conference specific environment variables. It is possible for clients to set and query for the values of specific strings. This is analogous to environment variables in Windows and UNIX. Interestingly enough, its

initial purpose was to help store replication policies and was simply adapted to this new role. The default cog policy strings are

- **VersionChainLength**: The maximum length of a version chain in the cog. Version chains are a sequence of file versions, where only the oldest version is stored in file system as a full file. All other versions can be obtained by applying deltas.
- **WriteInterval**: This specifies how often the FIT and cog policy are written to disk.

It is possible to specify replication policy for the cog. This policy defines the number of replicas that a version should have. The number of replicas depends on the type of the version. For example, it might be specified that all ASCII files have 5 replicas. The cog policy manager ensures that cog policy changes are propagated to all sites.

User Policy

DFS also allows the user to set policies that apply only to him/her. In a manner similar to cog wide environment variables, applications can also take advantage of user specific environment variables. User policies are a fine grained way of controlling DFS operations.

As described previously, DFS uses a versioning scheme with immutable files. One of the personal policies that a user can set is, to decide which version he/she sees when a file is opened. The user can specify either of the two options listed below.

- Get the latest version that was seen by the user (default)
- Get the latest version created.

It is possible for the user to override the policy by specifying the version while asking for the file. For the sake of clarity, consider the following example. Let user A and user B, be working on the same version of a file. After a while, user A closes the file. This file is now saved on the server. User B is not affected right now and proceeds to

finish his/her work and then save the file. When user A opens the file again, one of the three things might happen

- The user policy can be set to get the last file that the user opened. In this case, the user sees the file that he/she saved
- The user could be given a copy of the latest version of the file. Here, the user sees the version created by B
- The user could specify the version number to avoid any ambiguity.

DFS also has a personal history manager which keeps track of the latest version of every file that was accessed by the user. The server can consult the history manager to obtain the version number, in case the user wishes to see the last accessed version.

User policies also allow users to specify replication policies for files created by them. In a manner similar to cog replication policies, the user can specify the number of replicas for every type of file.

Migration

DFS is designed to operate over Wide Area Networks (WANs). WANs are characterized by long latencies and low bandwidth. In addition, one characteristic of a distributed system is continuous partial operation. The entire system is almost never operational simultaneously [28].

File migration is one of the approaches taken by DFS to mitigate these problems. DFS can move files from one server to another, to increase availability and decrease latency. Migration is complicated by the fact that files are often stored as deltas. Moving just the delta does not accomplish anything because DFS will need to access the earlier server to reconstruct the file. So, DFS migrates a version along with all versions that depend on it and all versions that this version requires. This list of versions is called a version chain. The first version in a version chain is a full file.

Migration is also simplified by versioning rules that place limits on the length of a version chain. This reduces the amount of data that must be transmitted to migrate a version. Another rule ensures that no two branches have the same last full ancestor.

Replication

File replication is one of the ways of improving performance and availability of the system. If a server that maintains a file is unavailable, DFS looks to get the file from another replica. As long as at least one replica is available, DFS will retrieve the file. DFS hides replication detail from clients and maintains replica transparency. Since files are stored as immutable versions, no replica consistency issues have to be addressed. Some design options are considered below.

Replicate entire file

In this option, the file (a delta or a full file) is considered as the unit of replication. The entire file is replicated at a site. The advantage of this scheme is its simplicity.

Replicate portions of file

In this case, a file (a delta or a full file) is split into blocks. The size of the blocks can be fixed or variable. Each block is a unit of replication and is spread among the sites. The file is reconstructed by obtained the blocks from the replicas. This is analogous to RAID devices.

RAID allows the use of error correction codes. It is not necessary to obtain all pieces to reconstruct the file. It is also far more complex and requires the system to maintain a lot of metadata.

Replicate on create

Replication can be performed as soon as a file is created. The server queries user and cog policies to obtain the number of replicas. This is determined by the file type. An

important file can be replicated at a lot of sites to increase availability. A less important file can have fewer replicas, to save space.

Replicate on open

It is also possible to replicate a file on a site, when the site needs it. In this case, when the site obtains the file from another site, it checks whether the file has as many replicas as required for a file of that type. In case it has fewer replicas, a replica is created at this site and the FIT is updated.

Design decision

In DFS, the entire file is replicated when the file is created. Replication of entire files was chosen because of the simplicity of the scheme. It also makes other operations (like migration) easier. Splitting the file in several locations could force the server to connect to more remote sites.

Simplicity is also one of the reasons for choosing to replicate the file on creation. In the replicate on demand scheme, it does not make sense to replicate the delta. All replicas should be complete files. So, there would be only one delta and all other replicas would be full files. This has two side effects. The first is that it increases storage requirements and the second is that it creates a situation where all replicas are not the same. All this can be avoided by replicating a file while creation. It also ensures that a file is not lost, even if it has not been accessed and the site that created it has crashed. Replication on creation ensures that the replication policy at the time of creation is followed. In case of replication on access, the decision to create a file will depend on the policy at the time of access. This is more unpredictable.

Caching

DFS is designed to use caching aggressively to improve performance. This is very significant over WANs. DFS supports caching on both the client and the server.

Client caching

DCS client process maintains a cache on the client machine. When asked for a file, the client obtains the file from the server and stores it in the cache directory. All operations are performed on the file. When the file is closed, the new file is pushed to the server, which stores it.

The advantage of an immutable versioned file system is that a version of the file will never change. It can be cached on the client as long as necessary, with hard disk space as the only constraining factor.

Clients do not cache attributes because they might change over time. It is preferable to get the attributes from the server.

Server caching

In contrast to most distributed file systems, DFS allows both replication and server caching. Server caching is similar to replicate on demand. When a file is obtained from another site, the server caches the full file for future use. The biggest difference between caching and replication is that, when a file is replicated on a server, the server must have the means to retrieve the full file without having to contact any other server. The server cannot remove the file from its local storage unless the file has migrated to another server. On the other hand, the contents of server cache do not concern any other server. A server can clear its cache when it needs space. Server and client caching are used to establish a working set of files on the respective machines.

Security

DFS security has two components. The first is, to provide secure communication between the client and the server. Secure Communication Service (SCS) is a DCS component designed to ensure authenticity, integrity and confidentiality of all messages. It also provides ways of authenticating users. Using SCS for this purpose will greatly simplify DFS design.

The second component is to ensure that users are not allowed to perform unauthorized actions. Modern distributed file systems use access control lists for this purpose. In DCS, Access Control Service (ACS) is responsible for maintaining rights and protection information. ACS is extremely flexible and can even allow voting. DFS uses ACS to maintain the access control lists. ACS also allows users to create new attributes.

File System Interface

DFS provides an interface to the file system located at the server. This interface is used by clients and other servers. The server-client interface is described below:

- **create**: If the function is provided with data, it stores the data as a new file. In the event that no data is provided, a blank file is created.
- **readVersion**: This command is used by clients to obtain the contents of a DCS file. The version is determined by the versioning policy. The user can also specify a version explicitly.
- **save**: It saves the given data as a new version of the file. The parent version must be specified.
- **deleteVersion**: This marks a given version of a file as deleted. The file is not physically deleted. It is marked in the FIT as unavailable.
- **undeleteVersion**: The function reverses the action of DeleteVersion. It marks the version as available.
- **deleteFile**: It physically removes all versions of a file from all sites. It is not possible to retrieve the file.

- **copy**: This command asks the server to copy a version of a file as another file. The new file is not just a delta. It is the full contents of that version.

- **move**: This command renames a file or directory. All versions are renamed.

- **makeDirectory**: It creates a new directory in the file system. This operation merely updates the FIT. No directory is created on disk.

- **removeDirectory**: This command removes an empty directory. Again, this operation modifies only the FIT.

- **listFile**: This command lists the attributes of all versions of a file.

- **listDirectory**: It lists the contents of a directory. A directory can contain files or other directories.

- **migrate**: This command migrates the given version to a server. Along with the version, its entire version chain is moved. It is possible to specify the server from which the file is to be moved.

- **getCogPolicy**: This command allows the user to view cog policy.

- **setCogPolicy**: This command allows the user to modify cog policy.

- **getCogReplicationPolicy**: This command allows the user to view cog replication policy.

- **setCogReplicationPolicy**: This command allows the user to modify cog replication policy.

- **getPersonalReplicationPolicy**: This command allows the user to view personal replication policy.

- **setPersonalReplicationPolicy**: This command allows the user to modify personal replication policy.

- **logout**: This command flushes user related information to disk.

- **isDirectory**: It returns true if a given path is a directory.

- **isPresent**: This command returns true if a given path exists.

The server: server interface is described next.

- **getSiteList**: This command returns the list of sites in the cog.

- **readFile**: This command reads a version of a file.

- **writeFile**: This command writes a version of a file.
- **deleteFile**: This command deletes a version of a file.

Conclusion

The chapter presents the requirements and the design of DFS. All options have been documented, even those which have not been used. This allows future designers to modify the design if they are not applicable any longer.

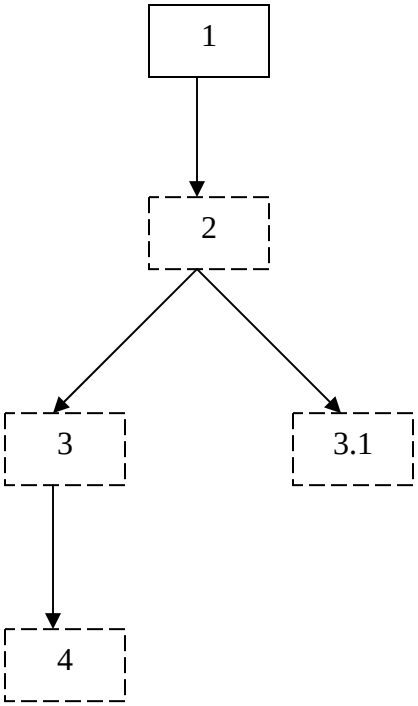

Figure 3-1. Forward deltas. The dashed boxes represent delta files and the solid box represents a complete file.

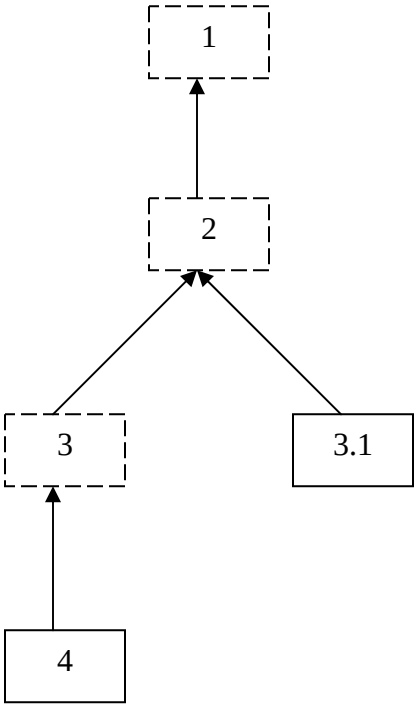

Figure 3-2. Reverse deltas. The dashed boxes represent delta files and the solid boxes represent complete files.

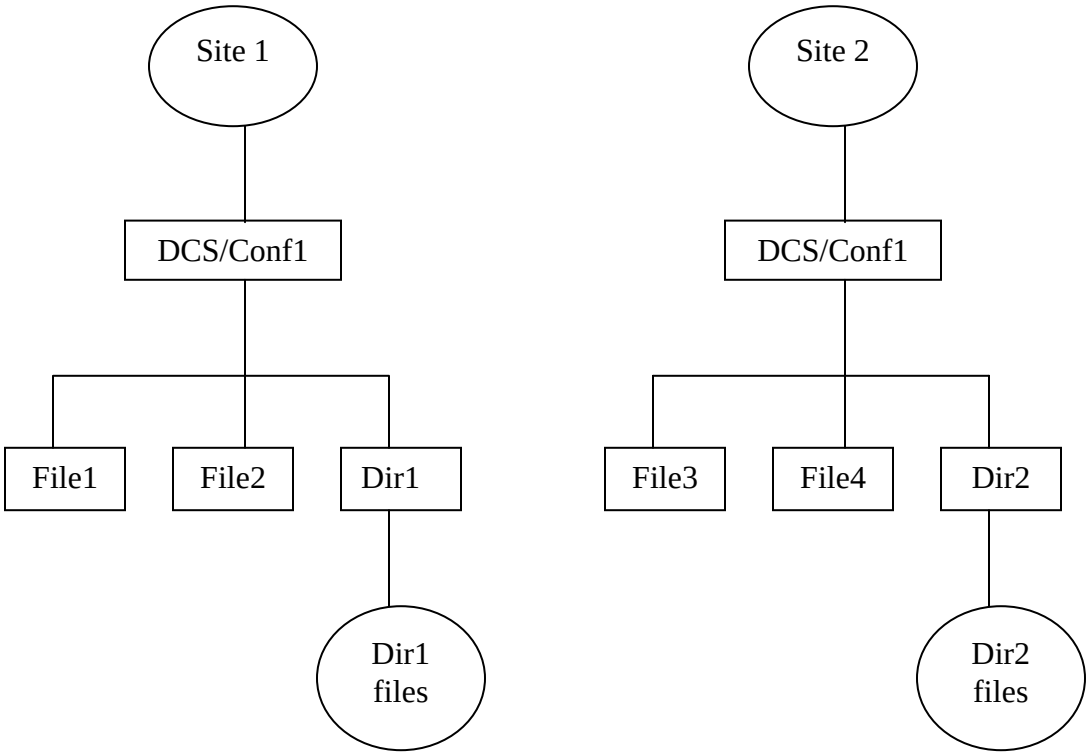

Figure 3-3. Actual location of files in a conference

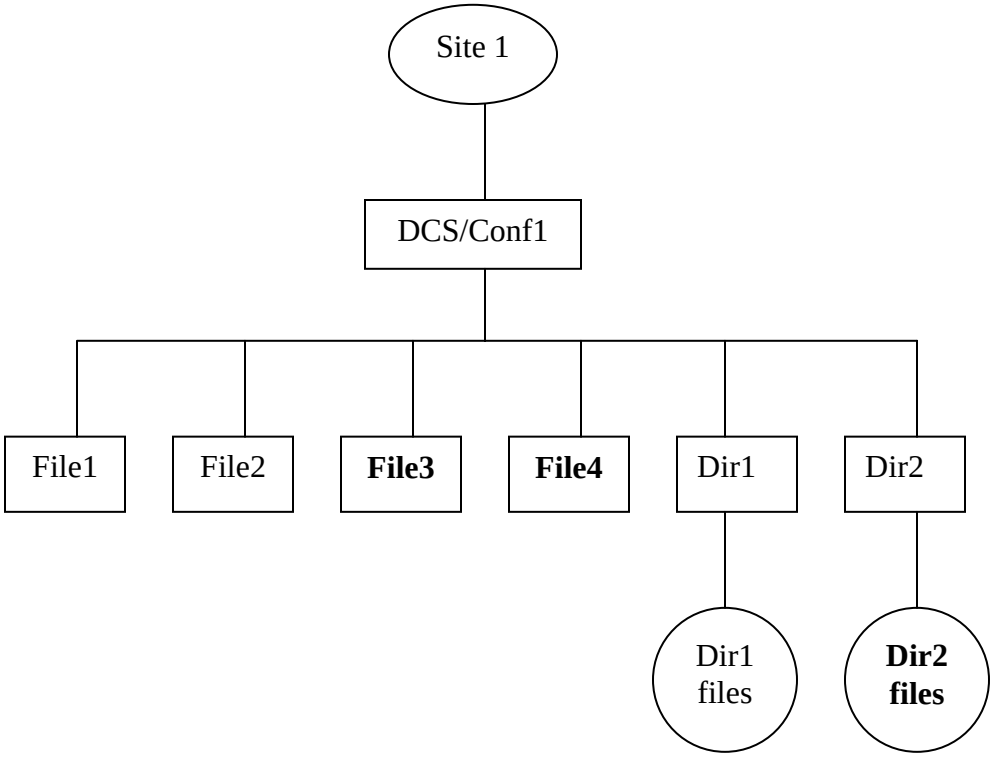

Figure 3-4. Virtual view of conference from site 1.

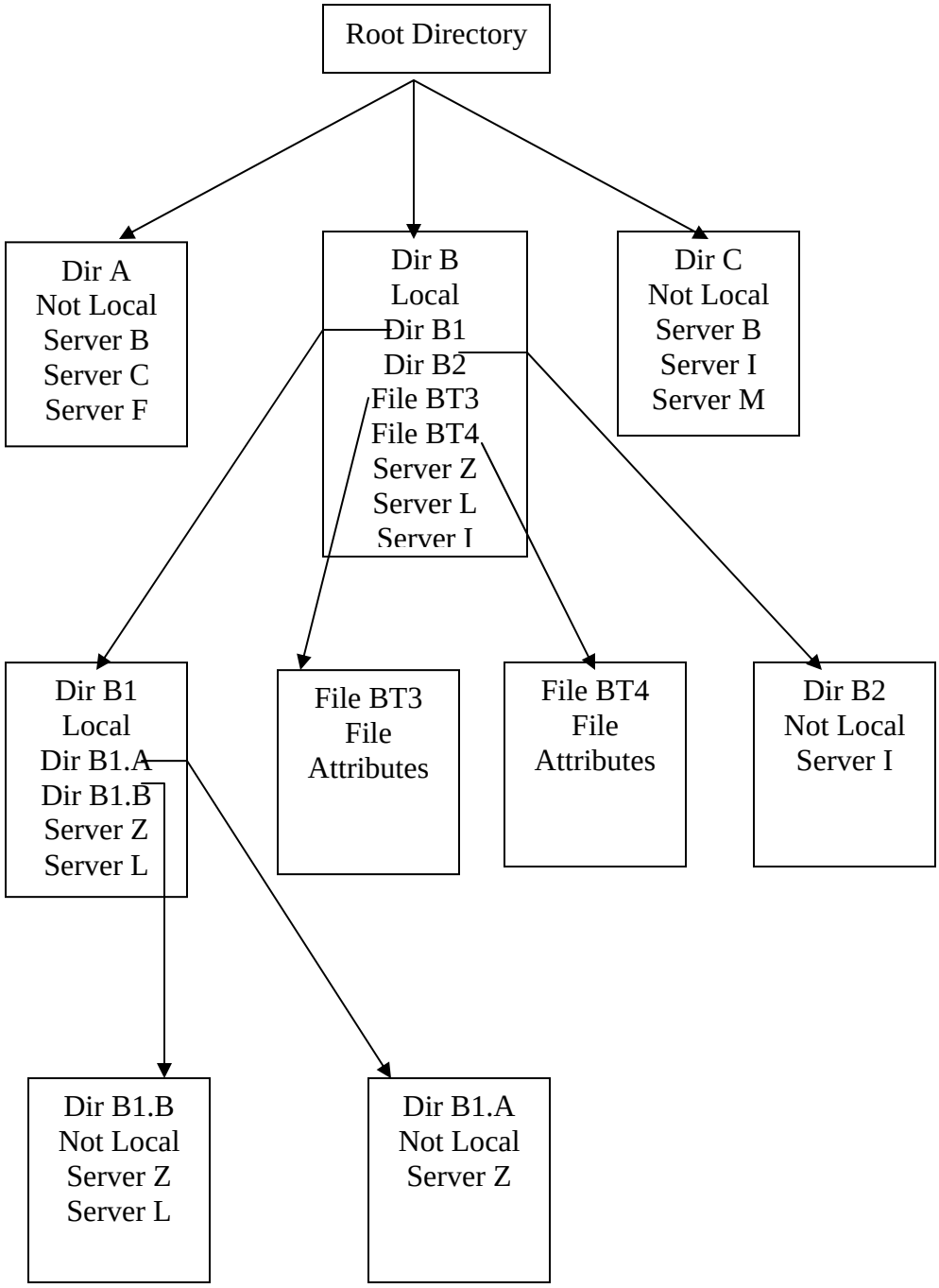

Figure 3-5. Sample FIT

CHAPTER 4
IMPLEMENTATION

This chapter describes the implementation of DFS. It starts with a discussion of the minimum requirements and the interaction of DFS with other DCS services. Details of DFS architecture, with a description of the server, client and DFS applications conclude the chapter.

Requirements

DFS has been implemented in Java. J2SE 5.0 or greater should be installed on the system. DFS code is in pure java and can be run on any operating system and architecture without changes to the code. However, the versioning system uses BSDiff, BSPatch and Bzip for delta compression. These tools are available for Windows and all flavours of UNIX.

Interaction with Distributed Conferencing System Services

Interaction with Conference Control Service

Conference Control Service is the lynchpin of DCS. Like most DCS services, CCS has been designed using client-server architecture [27]. CCS provides a graphical tool – the User Manager (UM) which functions as a CCS client. User Manager allows the user to log in to the conference and use the services provided by the CCS server. The DFS client cannot connect to the server until the user's credentials have been verified by CCS. On the server side, requests to the DFS server are often routed through CCS. Current work on DCS is focusing on integrating DFS with CCS.

Interaction with Access Control Service

ACS is used by DFS to ensure that users do not perform any unauthorized action. When a file is created, an entry is made in ACS tables, identifying the file and specifying the permissions of different roles. Also, when an operation on a file or its metadata is to be performed, ACS is consulted to determine whether the user has the specific permissions.

Interaction with Secure Communication Service

SCS is a DCS service that allows other DCS modules to communicate safely. It uses cryptographic techniques to guarantee message authenticity, integrity and confidentiality. SCS also maintains client and server certificates and changes keys periodically. DFS uses SCS to provide secure communication between DFS servers and between DFS clients and servers.

Communication

DFS has not been completely integrated with SCS. This has forced DFS to implement its own communication module. All communication between the DFS client and the DFS server takes place as a Java Remote Method Invocation (RMI). DFS Server object is registered with RMI registry and the client binds to it. The client invokes the appropriate methods for various operations.

Communication between servers also takes place using RMI. Distributed File System Server helps in initializing new servers that wish to join the conference. The FIT Manager also binds to the registry. This helps it contact other sites and maintain consistency. The Replication/Migration Manager registers with the RMI registry so that it can deposit and retrieve replicas on other sites. Both User Policy Manager and Cog Policy Manager bind to the registry to ensure policy consistency throughout the cog.

Distributed File System Architecture

The implementation architecture of DFS is shown in Figure 4-1. The components in bold communicate with the corresponding modules on other servers to maintain synchronization. The details of the modules are given next.

File Information Table Manager (FITM)

File Information Table Manager maintains and updates the FIT. It is also responsible for synchronizing the FIT with other servers. It ensures that the site is consistent and has an up-to-date list of files. When the server is started, it contacts other servers to get the latest FIT and when the server is shutdown, it writes the FIT back to the disk.

The FIT is stored in a file on the directory DFS_ROOT/system. The filename is specified in the property file. For the sake of convenience, the list of attributes that it maintains is reproduced here.

- Location of the file within DCS directory space
- Type of virtual file (file or directory)
- Version of the file
- Type of file
- Owner of the file
- Size of the file
- Time the file was created
- Mode at which the file can be opened
- Location of replicas
- Is this a full file
- Should this file always be a full file

- Has this file been deleted
- Parent version of this file.

The first attribute stores the DCS path of the file. The /DCS/<conference name> prefix is stripped out of the path name. They next attribute identifies whether the virtual file is a file or a directory. Directories have only a few attributes associated with them. Files have all attributes and can have multiple versions. There is a similar attribute, type of version. Most DFS policies are set according to the type of version. Different versions of the same file can be of different types. For example, a document with the format of a letter might be of template type but after it has been filled up, its type can become letter. This is set by application at the time the new version is written. The FIT does not maintain file permissions. The last accessed time is not maintained because this places an unnecessary synchronization load on the servers. The attribute mode is not currently implemented. It can specify the mode in which the file should be opened – read-only, binary, can cache, etc.

One copy of the FIT is maintained in memory for fast access. Hash tables are used because they have an order of O(1) access time. The FIT is periodically written to disk. This interval is configurable.

Version Naming Manager (VNM)

Version Naming Manager generates version names for new versions of a file. The generated version name depends on the parent version and the number of its siblings. Version naming in DFS has the following characteristics.

- The version zero is the first version of a file.
- Siblings are versions that are derived from the same parent. Siblings can be identified by the fact that they have everything in common until the last sequence of alphabets. In Figure 4-2, V2, V2a.0 and V2b.0 are siblings.

- Once all alphabets are used up, the next sibling is V2aa.0 and so on.
- To obtain the parent of a version, decrement number after the last period.
- However, if the number is 0, then remove the 0, period and the alphabetic portion and then decrement the number, to obtain the parent. The parent of V2b.1a.0 is V2b.0.

Figure 4-2 shows a version tree in DFS. Versions marked in bold are the first versions of their sub tree. DFS stores all these file in full. Version Control Manager (VCM) obtains these file names from VNM and then stores the file. Version Naming Manager also has routines to determine whether a child is on the main branch of its parent. Using this information, VCM decides whether the file should be stored as a delta or as a complete file.

Cog Policy Manager (CPM)

The Cog Policy Manager is responsible for maintaining general cog policy. In addition, it also maintains specific policies like the cog replication policy. Cog policy values can be queried by applications. It is possible for applications to modify the policy as well. Replication policy determines the number of replicas of a file. This information is used by the Replication/Migration Manager (RMM).

Policy values can be modified using the setPolicy interface which is exposed to the client. The getPolicy interface is used to obtain a particular policy. Cog Policy Manager is also responsible for maintaining synchronization with other sites. When cog policy is modified by a user, CPM communicates this to other sites so that cog policy is uniform across all sites.

User Policy Manager (UPM)

User Policy Manager (UPM) is quite similar to CPM. It maintains policies regarding specific users. This includes the general user policy and user replication policy.

They are quite analogous to their cog wide counterparts. They can be queried and modified by applications. In case of conflicts between cog and user policies, the application can choose to follow either.

One user policy is to specify which version of a file is seen by default. This can either be the latest version of the file or the last version seen by the user. User Policy Manager needs to keep track of the last version of every file seen by the user. This component of UPM is called Personal History Manager (PHM). Whenever the user opens a file, PHM enters the file name and the version. This can be retrieved by querying UPM.

Cog policies must be immediately communicated to other sites in the cog. By contrast, it is unlikely that a user will be connected to more that one site in a conference. That makes it unnecessary to update every site when a change is made to user policy. When the user logs out of the system, the server contacts all other sites and updates their user profile.

Cogs have a large number of users. It is prohibitively expensive for sites to load the profile of every user. Instead sites load profiles into memory on demand. Once the user logs out, the profile is dumped to disk and that memory is reclaimed.

Version Control Manager (VCM)

Version Control Manager (VCM) is responsible for the versioning aspects of DFS. It gets the version name from VNM, along with the number of replicas from CPM and UPM. It then hands the file over to Replication/Migration Manager (RMM) which replicates the file on different sites. VCM then checks whether a delta of the parent version needs to be created and replicates it as necessary.

Its other task is to rebuild files from deltas. For this purpose, it gets the list of versions that are required to reconstruct the file. Then, it applies the deltas repeatedly and rebuilds the file.

Creating a Delta

The new version is always stored as a complete file. It is the parent version that must now be converted into a delta. In some cases however, the parent file remains as a complete file. These cases are recapitulated below.

- The version 0 is a full file.
- Any version that is not on the main branch, i.e. if it is not the first child of its parent is stored as a complete file.
- Any file that not stored in exactly the same servers as its parent is stored as a complete file.
- If the version chain exceeds its maximum length with this file, the file is always complete.

If the parent version does not fall under any of these categories and is a complete file, it must be converted to a delta. First, the grandparent version number is obtained from FIT. Version Control Manager reconstructs the grandparent file. This may not be necessary if the grandparent is a full file. Now, a delta is created between the grandparent and the parent files. Due to the lack of a delta compression package is java (current language of implementation), DFS uses the utility BSDiff to create deltas. BSDiff was chosen because it is available for a large number of platforms and can handle both text and binary files. The delta file is replicated on the same sites and the FIT is modified to show that the parent version is no longer a full file.

Recreating a Version

The FIT stores the version number of the parent of every version. When VCM asks for the list of versions that are required to recreate this version, the FIT traverses up the versioning tree to find the latest ancestor that was stored as a full file.

This full file is copied to a temporary file. The delta of its child version is applied to recreate the child version. The next delta is applied to get the next version and so on. The deltas are applied till the required version is recreated. The data in the temporary file is then returned. BSPatch is a utility which can recreate file that were diffed using BSDiff.

Replication/Migration Manager (RMM)

File replication and migration are handled by Replication/Migration Manager (RMM). When a request is made to migrate a file, RMM checks whether a copy of the file already exists on the destination site. If there is a replica of the file on the destination server, no action is taken. If the file is not present on the site, RMM contacts its corresponding modules in the source and destination sites. In the first step, a copy of the file is transferred from the source to the destination server. The FIT is then updated to reflect this change. In the final step, the file is deleted from the source server. Replication/Migration Manager uses Local Filestore Manager (LFM) to handle the actual storage and retrieval.

When a file is created, it is replicated on the required number of sites. Replication/Migration Manager identifies the servers which can host the file. It connects to RMM on these servers and passes the data to them. FIT is then updated to include the location of these servers.

Local Filestore Manager (LFM)

Local Filestore Manager (LFM) is the component of DFS that interacts directly with the OS file system. When VCM stores a file, it passes the data and asks LFM to store the data as a file on disk.

It handles the retrieval of files too (both delta and complete files). As far as LFM is concerned, both are identical. It just reads the data and passes it on. It is up to the Version Control Manager to rebuild the file.

Local Filestore Manager uses the underlying OS file system for file and disk operations. This eliminates the need to implement file system manipulation functions, which have already been correctly and efficiently implemented.

Distributed File System Server

Distributed File System Server glues all the components together, and coordinates their activities. It also interacts with the client and with other servers when necessary. When the server is started, it loads the DCS property files and if necessary, creates the DCS directory. It then reads the list of servers that belong to the conference. At this point, one of the servers is contacted and a fresh list of cog sites is obtained. Now, all servers in the cog are asked to add the location of this new server.

The next step is to synchronize the server with other sites in the conference. It tries to obtain the up-to-date FIT table from another server. In case it is not successful, it looks for the FIT file on disk and loads the FIT. If there is no FIT file, the server starts with a clean FIT. Cog policies are requested from other servers in the cog. Again, if it is unsuccessful, the server tries to read it from disk. If there is no policy file, a blank policy is loaded.

The final step is to register the necessary server objects on the local RMI registry. All modules that require synchronization with other sites are registered. These include DFS Server, FITM, RMM and both UPM and CPM.

The interfaces presented by the server to the client and to other servers are reproduced below for convenience.

Client-Server Interface

- **create**: If the function is provided with data, it stores the data as a new file. In the event that no data is provided, a blank file is created.
- **readVersion**: This command is used by clients to obtain the contents of a DCS file. The version is determined by the versioning policy. The user can also specify a version explicitly.
- **save**: It saves the given data as a new version of the file. The parent version must be specified.
- **deleteVersion**: This marks a given version of a file as deleted. The file is not physically deleted. It is marked in the FIT as unavailable.
- **undeleteVersion**: The function reverses the action of DeleteVersion. It marks the version as available.
- **deleteFile**: It physically removes all versions of a file from all sites. It is not possible to retrieve the file.
- **copy**: This command asks the server to copy a version of a file as another file. The new file is not just a delta. It is the full contents of that version.
- **move**: This command renames a file or directory. All versions are renamed.
- **makeDirectory**: It creates a new directory in the file system. This operation merely updates the FIT. No directory is created on disk.
- **removeDirectory**: This command removes an empty directory. Again, this operation modifies only the FIT.
- **listFile**: This command lists the attributes of all versions of a file.
- **listDirectory**: It lists the contents of a directory. A directory can contain files or other directories.

- **migrate**: This command migrates the given version to a server. Along with the version, its entire version chain is moved. It is possible to specify the server from which the file is to be moved.
- **getCogPolicy**: This command allows the user to view cog policy.
- **setCogPolicy**: This command allows the user to modify cog policy.
- **getCogReplicationPolicy**: This command allows the user to view cog replication policy.
- **setCogReplicationPolicy**: This command allows the user to modify cog replication policy.
- **getPersonalReplicationPolicy**: This command allows the user to view personal replication policy.
- **setPersonalReplicationPolicy**: This command allows the user to modify personal replication policy.
- **logout**: This command flushes user related information to disk.
- **isDirectory**: It returns true if a given path is a directory.
- **isPresent**: This command returns true if a given path exists.

Server-Server Interface

- **getSiteList**: This command returns the list of sites in the cog.
- **readFile**: This command reads a version of a file.
- **writeFile**: This command writes a version of a file.
- **deleteFile**: This command deletes a version of a file.

Distributed File System Client

The DFS Client exposes the DFS server interface to DFS aware applications. Any application that seeks to exploit the features of DFS must include a reference to the DFS client. Prior implementations mandated that each user have only one DCS Client, which has been done away with, in this implementation. Applications can use multiple DCS Clients at the same time. It is also possible for multiple applications to use the same client

and interleave their calls to the DFS server. Most methods on the client side, merely call the same methods on the server.

Applications

Shell

In order to take advantage of the capabilities of DFS, it is necessary to implement applications. The shell is an application designed to perform operations similar to UNIX shells. It connects to the server using the DFS Client and manipulates DCS files and directories. The shell can accept both absolute and relative paths (starting with ./ or ../). It implements the following commands.

- **ls**: If the command is applied on a directory, all its contents are listed. These might be files or other directories. If the command is applied on a file, information is displayed on all versions of the file.
- **cd**: This command changes the current directory.
- **cp**: This command copies a given version of a file as another file.
- **mv**: This command renames / moves a file or directory.
- **delfile**: This command deletes a file permanently.
- **delver**: This command marks a version of a file as deleted.
- **undelver**: This command restores a deleted version.
- **mkdir**: This command creates a new directory.
- **rmdir**: This command removes an empty directory. If a directory is not empty, an error message is displayed.
- **migrate**: This command migrates a file version to a server. The location of the source site can be optionally specified.
- **logout**: This command closes the connection to the server.
- **pwd**: This command displays the current working directory.

- **chpol**: This command changes a specific policy. The next argument must specify the policy which is to be changed. Current options are:
    - **repl**: User replication policy.
    - **gen**: General user policy.
    - **coggen**: Cog general policy.
    - **cogrepl**: Cog replication policy.
- **getpol**: This command gets a specific policy. It requires one of the options listed above.

Distributed Conferencing System Text Editor

The DCS Text Editor is an application which is specifically designed to demonstrate the functionality of DFS. The text editor uses the DCS Client to connect to the server. When the text editor is asked to open a file, it checks to see if the file exists. If the file does not exist, it asks the user whether a new file should be created and then creates the file if needed. In case of an existing file, the file is retrieved and its contents are displayed in the editor. The editor allows the user to modify and then save the file as a new version.

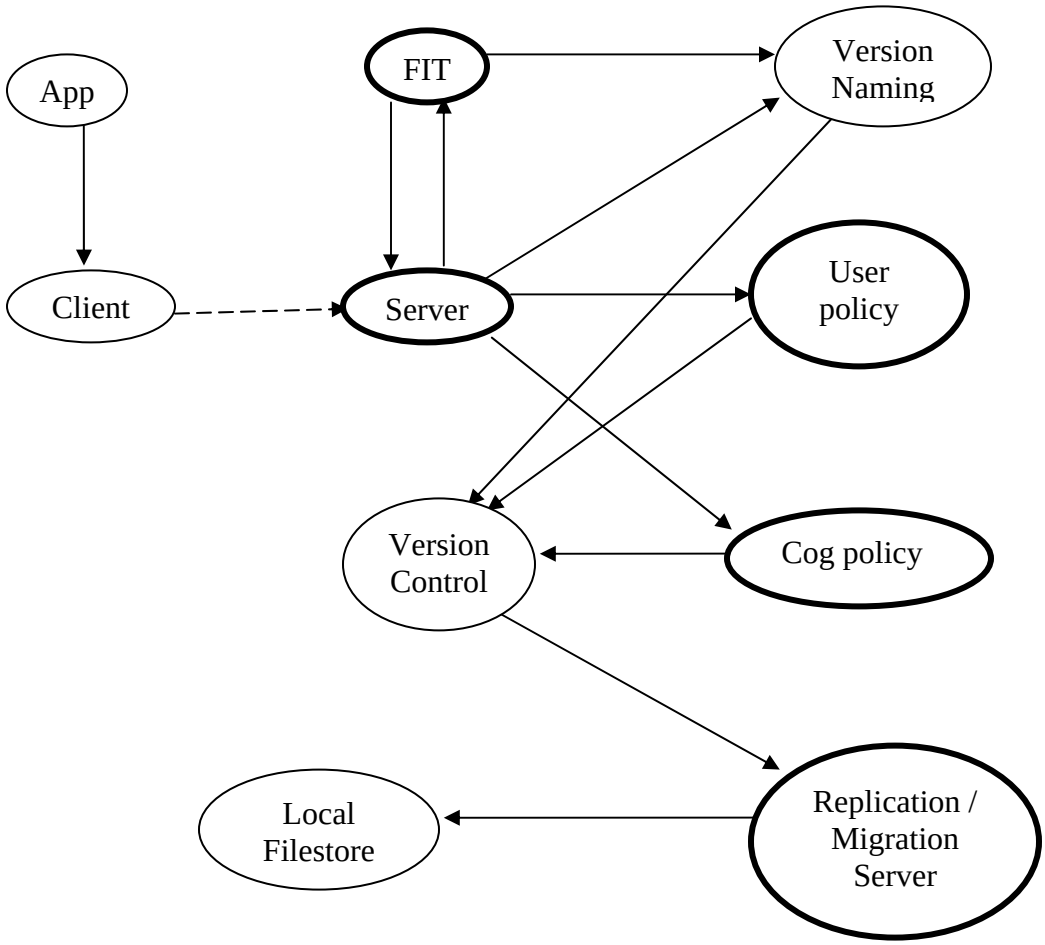

Figure 4-1. DFS architecture. The modules in bold interact with their counterparts on other servers.

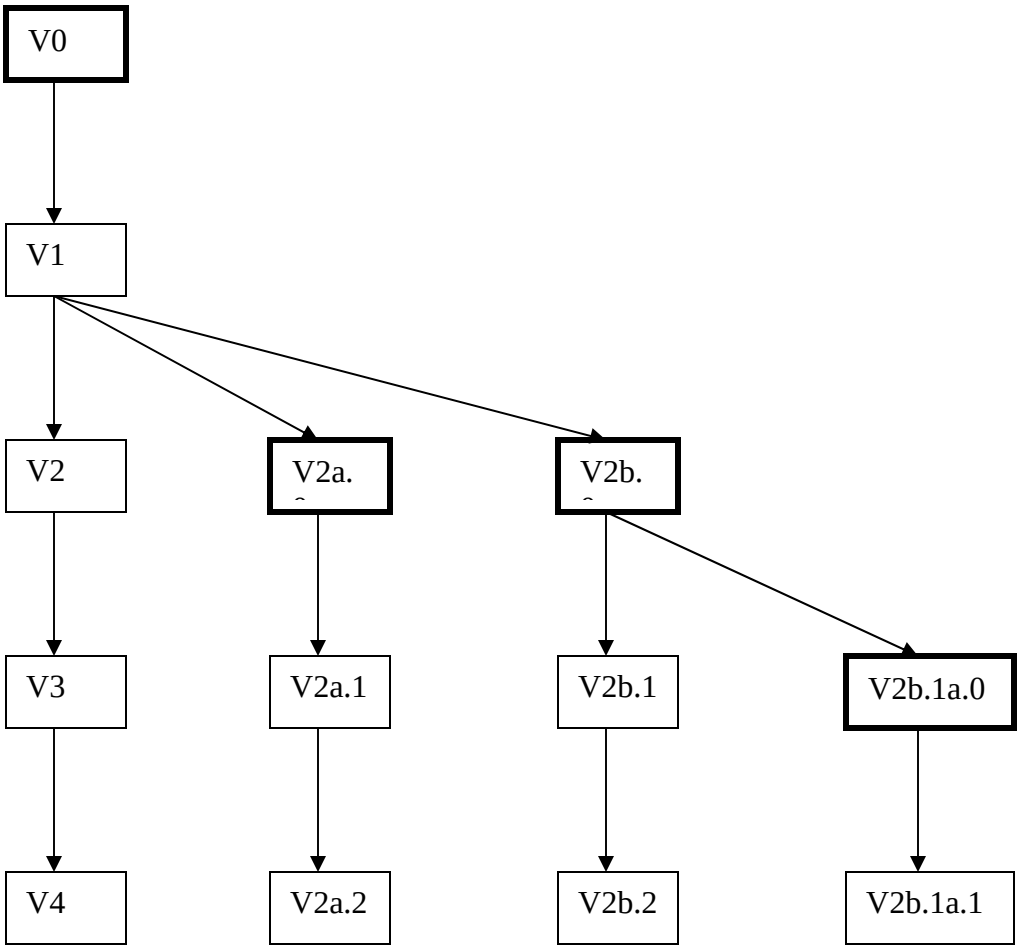

Figure 4-2. Version tree in DFS

CHAPTER 5
EVALUATION

Testing

DFS is one of the modules of DCS and depends on some of the services provided by DCS. Unfortunately, the components of DCS have not been completely integrated. In order to test DFS, specific test classes have been created to simulate the missing DCS services.

DFS was tested on both Windows and Linux and was found to be satisfactory. The test bed had three machines running DFS server process and multiple clients. One of the aspects tested, was the ability of DFS components to maintain synchronization.

Tests were conducted to verify all components of DFS. Replication/Migration Manager, Version Control Manager, User Policy Manager and Cog Policy Manager were the focus of the tests. Additional tests were made to ensure that servers could be added on the fly, to increase space.

Evaluation

The overall objectives of DFS were stated previously in Chapter 3. These are reproduced below for convenience.

- The file system should provide a uniform name space.
- Files and directories should be location transparent and location independent.
- Concurrency control should allow users to modify the same files, without losing changes made by any user.
- The consistency semantics should be predictable.

- File migration and replication must be supported to improve availability and performance.
- New servers must be allowed to join the conference without having to shut it down. Addition of servers should not change user experience negatively.
- The file system should provide an interface to allow users and applications to interact efficiently.

DFS provides a uniform name space for all files in the conference. In fact, it is possible to extend it to provide a global namespace. In this respect, it is similar to Coda. The second objective is achieved by providing a File Information Table (FIT). The FIT acts as a mapping table and translates between the logical name of the file and its actual location. The immutable versioning system in DFS ensures that concurrent changes made to a file do not overwrite each other. Changes made by users are saved as new versions. This ensures that writes do not interfere with each other and files never lose changes. DFS file consistency is predictable and configurable. Users can see the latest changes by specifying their policy. The semantics are quite close to session semantics.

Replication and migration are supported by Replication/Migration Manager. This increases system availability and fault-tolerance greatly and improves performance as well. DFS implementation allows a new server to join the conference as long as it knows the location of one server in the cog. The other servers in the cog bring the server up to speed. A user does not even know that a new server has joined the cog. DFS provides interfaces to both DFS Clients as well as other Servers.

It can be seen from the summary that the main objectives of DFS were achieved. The next section puts forth some of the ideas that could be examined in the next version of DFS.

Future Work

The current implementation of DFS is the first step towards implementing a flexible distributed file system with high availability and great ease of use. It has not been possible to design and implement all aspects of a distributed file system. However, DFS provides a foundation for future work. Some of the ideas that can be examined are given below.

- Integrate DFS with other DCS modules. DFS is designed to interact closely with Conference Control Service. It also depends on Secure Communication Service and Access Control Service for security. A good deal of work will be required to modify calls in all these services.

- Allow DFS to raise events. The Notification Service (NTF) is responsible for registering events and listeners. It also informs the listeners when the event takes place. DFS should use NTF to raise events. One scenario could be to inform specific users when a new version of a file is available.

- Current implementation of the File Information Table Manager (FITM) is not very scalable. Implementing the new design would improve scalability.

- Replace the current command line delta creation and rebuilding utility with a Java package. One of the biggest strengths of Java is platform independence. The use of command line tools has severely restricted the operating systems that DFS can run on. A delta compression package must be identified or developed for use in DFS.

- DFS client should be integrated with the operating system. This requires modification of OS system calls. Tight integration with Windows is virtually ruled out due to source code unavailability. Linux system calls can be modified to support DFS.

- Support file merging and reconciliation. Coda is a distributed file system that supports automatic file merging and reconciliation. Similar work can be done on DFS to allow users to merge files.

Conclusion

Distributed Conferencing System provides support for users to collaborate efficiently and securely. Distributed File System is an important part of DCS and provides the file system for it. Distributed File System must provide concurrency control,

uniform naming, location transparency and location independence, high performance and availability along with security. Other file systems provide some of these features but no other distributed file system supports all of them. Most do not deal very well with concurrency, believing that concurrent writes on the same file are extremely rare. Distributed File System handles concurrency well and allows each user to preserve his/her changes as immutable versions. Another feature of DFS is the ability to specify cog wide properties and replication policies. Though DFS is designed specifically for DCS, the ideas are applicable in any environment where concurrency and availability are important.

LIST OF REFERENCES

[1] 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,” in *Proceedings of the Summer USENX Conference*, Nashville, TN, June 1991, pp. 437-452.

[2] A. S. Tanenbaum and M. Steen, *Distributed Systems: Principles and Paradigms,* Prentice Hall, Upper Saddle River, NJ, 2003.

[3] R. Chow and T. Johnson, *Distributed Operating Systems & Algorithms,* Addison-Wesley, Reading, MA, 1997.

[4] V. Manian, “Access Control Model for Distributed Conferencing System,” M.S. thesis, University of Florida, Gainesville, FL, 2002.

[5] L. Li, “File Services for a Distributed Conferencing System (DCS),” M.S. thesis, University of Florida, Gainesville, FL, 1995.

[6] P. S. Yeager, “A Distributed File System for Distributed Conferencing System,” M.S. thesis, University of Florida, Gainesville, FL, 2003.

[7] R. Sandberg, D. Coldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," in *Proceedings of the Summer USENX Conference*, Portland, OR, 1985, pp. 119-130.

[8] R. Srinivasan, “RPC: Remote Procedure Call Protocol Specification Version 2,” Sun Microsystems, Santa Clara, CA, Request for Comments 1831, August 1995.

[9] B. Nowicki, “NFS: Network File System Protocol Specification,” Sun Microsystems, Santa Clara, CA, Request for Comments 1094, March 1989.

[10] B. Callaghan, B. Pawlowski and P. Staubach, “NFS: Network File System Protocol Specification,” Sun Microsystems, Santa Clara, CA, Request for Comments 1813, June 1995.

[11] B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, S. Shepler and D. Noveck, “NFS version 4 Protocol,” Sun Microsystems, Santa Clara, CA, Request for Comments 3010, December 2000.

[12] B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, S. Shepler and D. Noveck, “Network File System (NFS) version 4 Protocol,” Sun Microsystems, Santa Clara, CA, Request for Comments 3530, April 2003.

[13] M. Satyanarayanan, “A Survey of Distributed File Systems,” *Annual Review of Computer Science*, vol. 4, pp. 73-104, August 1990.

[14] M. Satyanarayanan, “Scalable, Secure, and Highly Available Distributed File Access,” *IEEE Computer*, vol. 23, pp. 9-21, May 1990.

[15] M. Satyanarayanan, and M. Spasojevic, “AFS and the Web: Competitors or Collaborators,” in *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, 1996, pp. 89-94.

[16] M. Spasojevic and M. Satyanarayanan, “An Empirical Study of a Wide Area Distributed File System,” *ACM Transactions on Computer Science*, vol. 14, pp. 200-222, May 1996.

[17] R. Needham and M. Schroeder, “Using Encryption for Authentication in Large Networks of Computers,” *Communications of the ACM*, vol. 21, pp. 993-999, June 1978.

[18] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, “Coda: A Highly Available File System for a Distributed Workstation Environment,” *IEEE Transactions on Computers*, vol. 39, pp. 447-459, April 1990.

[19] R. Pike, D. L. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey and P. Winterbottom, “Plan 9 from Bell Labs,” *Computing Systems*, vol. 8, pp. 221-254, February 1995.

[20] R. Y. Wang and T. E. Anderson, “xFS: A Wide Area Mass Storage File System,” in *Proceedings of Fourth Workshop on Workstation Operating Systems*, Napa, CA, 1993, pp. 71-78.

[21] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli and R. Wang, “Serverless Network File Systems,” in *Proceedings of ACM SOSP*, Ashville, NC, 1995, pp. 109-127.

[22] A. Muthitacharoen, B. Chen and D. Mazieres, “A Low-Bandwidth Network File System,” in *Proceedings of ACM SIGOPS Operating Systems Review*, Newport, RI, 2001, pp. 174-187.

[23] M. O. Rabin, “Fingerprinting by Random Polynomials,” Harvard University, Cambridge, MA, Technical Report TR-15-81, 1981.

[24] U.S. Department of Commerce/National Institute of Standards and Technology, “*Secure Hash Standard*,” National Technical Information Service, Springfield, VA, FIPS PUB 180-1, April 1995.

[25] D. Mazieres, M. Kaminsky, M.F. Kaashoek and E. Witchel, "Separating Key Management from File System Security," in *Proceedings of ACM SIGOPS Operating Systems Review*, Charleston, SC, 1999, pp. 124-139

[26] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," in *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, 2000, pp. 46-66.

[27] A. Bhalani, "Implementation of Conference Control Service in Distributed Conferencing System," M.S. thesis, University of Florida, Gainesville, FL, 2002.

[28] R. G. Guy, T. W. Page, J. S. Heidemann, and G. J. Popek, "Name Transparency in Very Large Scale Distributed File Systems," in *Proceedings of Second IEEE Workshop on Experimental Distributed Systems*, Huntsville, AL, 1990, pp. 20-25.

[29] A. V. Date, "Implementation of Distributed Database and Reliable Multicast for Distributed Conferencing System Version 2," M.S. thesis, University of Florida, Gainesville, FL, 2001.

BIOGRAPHICAL SKETCH

Prashant Jayaraman was born in Thanjavur, India in 1983, the son of Raji and S. Jayaraman and the younger brother of Vivekanand Jayaraman. He completed his Bachelor of Technology degree in computer science and engineering from Regional Engineering College, Tiruchirappalli, India in June 2004. He then decided to join the University of Florida in 2004 to pursue his master's degree. After graduation, he plans to pursue a career in industry but hopes to eventually earn a Ph.D.