uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Raed AlShaikh
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Master of Computer Science
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Towards Building a Fault Tolerant and Conflict-Free Distributed File System for Mobile Clients

TITRE DE LA THÈSE / TITLE OF THESIS

Azzedine Boukerche
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Abdulmotaleb El Saddik

Michael Weiss

Gary W. Slater
LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES / DEAN OF THE FACULTY OF GRADUATE AND POSTDOCORAL STUDIES

Towards Building a Fault Tolerant and Conflict-Free Distributed File System for Mobile Clients

By

Raed A. AlShaikh

A Thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of

The requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institution for Computer Science

School of Information Technology and Engineering

University of Ottawa

Ottawa, Ontario, Canada

Copyright © 2006 Raed AlShaikh

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-14882-9
Our file Notre référence
ISBN: 0-494-14882-9

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

The following publications by the author are relevant to this thesis:

Journal:

1) A. Boukerche, R. AlShaikh. "*A Conflict-Free Distributed File System for Mobile Clients*", International Journal of Wireless and Mobile Computing. To be Submitted.

Conference:

1) A. Boukerche, R. AlShaikh, B. Maeleau, "*Highly Available File System for Mobile Distributed Computing*", In 30th IEEE Proceedings of Local Computer Networks Conference (LCN), Sydney, Australia, November 2005, pp.608-614.

2) A. Boukerche, R. AlShaikh, "*Towards Building a Fault Tolerant and Conflict-Free Distributed File System for Mobile Clients*", 1st IEEE International Workshop on Performance Analysis and Enhancement of Wireless Networks (PAEWN'06), Vienna, Austria, April 2006.

3) A. Boukerche, R. AlShaikh, "*Towards Building HA-Clusters for High Performance Computing*", 5th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS 2006), Rhodes Island, Greece, April 2006.

Abstract

The rising demand for mobile computing has created a need for improved file systems that support mobile clients. Current file systems with support for mobility provide availability through file replicas that are cached at the client side. However, mobile clients may experience different obstacles in regard to the local cache, such as the intermittent connection, and serious conflicts when synchronizing back to the server.

In this thesis, we present a comprehensive classification of distributed and mobile file systems, and propose a novel distributed file system model for mobile clients with cache-less wireless devices. We discuss the implementation of our model, investigate its high availability functions and report on its performance evaluation using a cluster of workstations as a test-bed. Our test run results indicate clearly that our technique exhibits a significant degree of automation and conflict-free mobile file system.

Last but not least, we have proposed a novel scheme based on the FBR scheme and file pre-fetching to enhance the server-side caching strategy of our distributed file system. We present our scheme and discuss its evaluation performance using an extensive set of experiments.

Acknowledgment

Performing this research turned out to be a larger challenge than I ever imagined. I could not have completed it without the care and support of many wonderful people, and I'm delighted to be able to acknowledge them here.

First, I would like to thank my supervisor Dr. Azzedine Boukerche. I couldn't have had a better mentor. He always made time to see me, no matter how busy his schedule was. He was a constant source of good ideas and a perfect detector of bad ones. He challenged me when I needed to be challenged and boosted my confidence when it needed so. More than anything, he has been a true friend.

I warmly thank my colleague Bo Marleau. She is a very talented person and it has been a pleasure to work with her. Her help in the design and implementation was invaluable.

Finally, my family deserves greater thanks than I can possible give. Without their grounding and their love, I never could have come this far. Thank you, Mom and Dad, brother Ala and sister Lamees for your continuous support and constant encouragement.

Raed A. AlShaikh
University of Ottawa
February 2006

Contents

1 **Introduction** **1**

1.1 Problem Statement 3

1.2 Motivation 3

1.3 Thesis Outline 4

2 **Related Work on Mobile Distributed File Systems** **5**

2.1 Overview 5

2.2 SUN Network File System (NFS) 6

2.2.1 NFS Architecture 6

2.2.2 NFS Communication Modes 8

2.2.3 NFS Name Space 9

2.2.4 File Locking in NFS 9

2.2.5 Caching and Replication in NFS 11

2.2.6 Fault Tolerance in NFS 11

2.2.7 Enhancements to NFS version 4 13

2.3 CODA File System 15

2.3.1 CODA Architecture 16

2.3.2 CODA Communication 18

2.3.3 CODA Name Space 20

2.3.4 Disconnected Operations in Coda 20

2.3.5 Semantics of CODA File Sharing 22
2.3.6 Caching and Replication in CODA 24
2.3.6.1 Client Caching 24
2.3.6.2 Server Replication 26
2.4 Other Distributed File Systems 27
2.4.1 InterMezzo File System 28
2.4.1.1 Conflict Resolution in InterMezzo 29
2.4.2 Ficus File System 30
2.4.3 OceanStore File System 33
2.4.3.1 Object Replication in OceanStore 34
2.4.3.2 Basic Routing and Servers Manipulation 35
2.4.3.3 Caching in OceanStore 36
2.5 Summary 35
3 The System Model of a Conflict-Free File System for Mobile Clients 39
3.1 Overview 39
3.2 The Distributed File System Architecture and Design 40
3.2.1 File Alteration Monitor (FAM) 42
3.2.2 Concurrency Control 43
3.2.3 Servers Interaction and Integration Model 44
3.2.3.1 The Logging Process 45
3.2.3.2 The Reintegration Process 46
3.2.4 Conflict Detection and Resolution Strategy 47
3.2.5 The Cache Replacement Algorithm 52

3.2.5.1 The Prefetching Scheme 54

4 File System Request Redirection and Integration into the VFS Layer ... 56

4.1 Overview 56

4.2 How VFS Works 57

4.3 Approach for Redirecting Accessing Path for A File Access 58

4.3.1 System Call Interface 59

4.3.2 The VFS Layer 61

4.3.3 Stackable Fan-Out File System 61

4.3.3.1 The Approach 61

4.3.3.2 What is a Stackable Fan-Out File System? 62

4.3.3.3 Applying the Stackable File System 63

5 Experimental Results 64

6 Conclusion 72

6.1 Future Work 73

Appendix 75

List of Figures

1.1 A Distribued File System for the Health Care System 2
2.1 NFS interface to the Operating System 7
2.2 Lookup Operation in NFS v3.0 vs. NFS v4.0 15
2.3 Client/Venus/ Vice Interaction in Coda File System 17
2.4 Parallel RPC Replies in Coda File System 19
2.5 State Transition in Coda File System 21
2.6 Callback model in Coda 25
2.7 A Potential Constructed Path From the Sender to the Receiver in OceanStore File System 35
3.1 The Layout of our File System 41
3.2 STL Integratin Algorithm 47
3.3 Conflict-Detection Algorithm 49
3.4 STL/CTL Replaying Rules 50
3.5 FBR Cache Structure 54
4.1 File System Integration with VFS Layer 58
4.2 Changing sys_open to Another User Defined System Call 60
4.3 Redicecting sys_open Call to the Cache Server Path 60
4.4 Stackable File System Interface with VFS 62
5.1 Average Daily File System Use 65

5.2 CRA Benchmark 66

5.3 STL Size vs. Number of Conflicts 68

5.4 Conflict vs. Conflict-Free STL Reintegration Time 68

5.5 Reintegration Time vs. Number of Servers 69

5.6 Cache Size vs. Number of Cache Hits 70

5.7 Size of N vs. % of Correct Predictions 70

5.8 Size of N vs. % Increase in Network Traffic 70

5.9 Size of N vs. % of Correct Predictions Found in Cache 71

5.10 Size of N vs. % of Correct Cache Hits 71

List of Tables

2.1 File System's Comparison 38

5.1 File System Benchmark 67

Chapter 1

Introduction

The ability to share disk space and files over a network is one of the most significant advantages of distributed computing. It facilitates reducing local disk space requirements by making it easy for users to work together without ending up with duplicates of the same files. With the advancement of wireless networks and mobile computing, there is an increasing need to build a mobile file system that can access data efficiently and correctly anywhere and at anytime. At the same time, however, careful considerations have to be made when designing such a mobile file system for the currently available mobile devices, such as Personal Digital Assistants (PDAs) and cellular phones. As we know, these devices have limited storage space, yet they require accessing a large amount of data at anytime and anywhere. Thus, a trade-off between local limited resources with the need of continuous availability has to be balanced.

Consider the following typical scenario where this trade-off is crucial: A Hospital with hundreds of employees wants to exchange text and image through a set of transmission access points scattered around the campus. As shown in figure 1.1, each employee's files must at anytime and anywhere be available to anyone else among the other employees, and vice versa. The main challenge consists in guaranteeing continuous

availability coupled with a caching methodology which ensures near continuous freshness. The question is: *how can a high degree of availability be achieved, with at the same time a high degree of data freshness?*

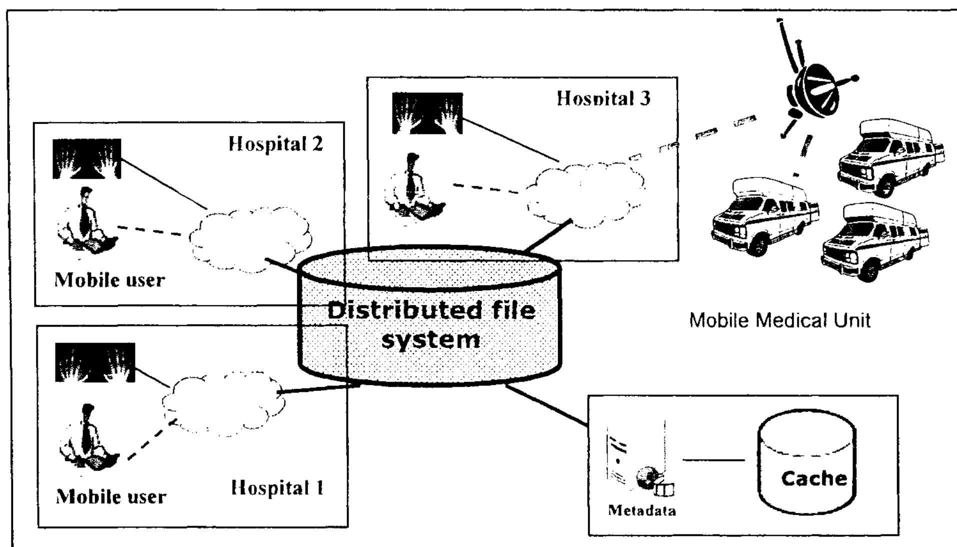

Figure 1.1: A Distributed File System in the Health Care System

Henceforth, there is a need for a novel cache and a node management technique, different than those used in existing distributed file systems aimed at mobility. More specifically, mobile clients may experience different obstacles such as disconnection from the network resulting from denial of access to the other nodes, the limited network bandwidth, or serious conflicts when synchronizing back to the server [5]. Therefore, it is essential to come up with a distributed file system solution that solves these problems and provides file system services during mobile clients' disconnections, manages node arrivals and departures, and works out data reintegration and conflict resolution on various file system objects.

1.1 Problem Statement

Currently, most heterogeneously connected operating systems use NFS (Network File System) as a mean to share files over the network [12]. However, several problems exist in the traditional NFS protocol: (1) NFS was designed with the assumption that the communication network is fast and reliable, and minimal attention was paid to the cache design and replication; and (2) scalability is a major problem with NFS [12,42]; several considerations in terms of *network bandwidth*, *disk space* and *memory* have to be made when deploying an NFS file server. In particular, the more clients are connected to the centralized file system, the more overhead is put on the server and the slower the requested service would be processed. Therefore, separating services across many machines has been used so far in many distributed file systems in order to cope with the increasing number of clients and servers. [8].

1.2 Motivation and Contributions

Client-side caching and servers' replication have been major design paradigms in order to guarantee data availability for fat clients (i.e., clients that are able to cache large amount of data) in a distributed environment. However, our goal in this thesis is to focus on those *thin clients* that cannot cache all needed data, i.e., cache-less devices. The proposed file system targets applications that benefit from clients' mobility without sacrificing data availability, such as heath care services, mobile-web services, and distributed mission-critical applications that consider continuous data availability as their highest priority. In particular, the goal of our design is to:

1) Provide a distributed and fault-tolerant file system for cache-less devices, and

2) Provide a *seamless* backup source, incase of a file system disconnection.

Consider a file server that gets disconnected from the network. Let us now assume that clients have not yet cached data that resides on that server. Obviously, clients will not be able to access this data unless the server reconnects back to the network. Therefore, a mechanism is needed to serve those *connected clients* and make up for the lost data that are caused by the *disconnected servers*.

It is important to mention that in this thesis, we consider distributed file servers as *peers that hold data*, and they are vulnerable to disconnections and network partitions.

1.3 Thesis Outline

The remainder of this thesis is organized as follows:

- Chapter 2 presents related work on mobile and distributed file systems and shows how each implementation targets fault tolerance and replication.
- Chapter 3 is the core of this thesis. It discuses in details the architecture and design of our distributed file system implementation, analyzes its characteristics and shows its algorithm.
- Chapter 4 proposes ways to integrate our design to the Virtual File System (VFS) layer in any Operating System.
- Chapter 5 demonstrates the test run results and evaluates the performance of our distributed file system accordingly.
- Chapter 6 concludes the thesis and provides some directions for future research.

Chapter 2

Related Work on Distributed and Mobile File Systems

2.1 Overview

The last two decades have witnessed an increase in complexity and maturity of distributed file systems. Both well-established commercial and research systems have addressed a vast palette of users' needs in today's highly distributed environments [3,4,38,46]. Those needs range from failure resiliency to mobility, to extended file sharing, and to dramatic scalability. In particular, the move towards mobility is noticeably visible in the early implementations of distributed file system, such as NFS (Network File System) [2,3,49], and in more advanced systems, such as Coda [1,3,4,39]. In many cases, other services such as electronic mail delivery and printing are layered on top of the distributed file system, furthering its importance [2].

In this chapter, we shed light on several existing file systems, with the emphasis on NFS and Coda. The reason is that NFS is considered a base of any distributed file

system [12], and Coda is one of the new distributed file systems that focus on high availability.

2.2 SUN Network File System (NFS)

NFS, or the Network File System, was originally developed by Sun Microsystems in the 1980's as a way to create a file system on diskless clients [41]. NFS provides remote access to shared file systems over the network. This means that a file system is actually sitting on machine A, and machine B can "*mount*" this file system that appears to users on machine B as if the file system actually resides on their local machines, as shown in figure 2.1. Therefore, NFS File System is transparent to users, and allows fast, seamless sharing of files across a network. NFS was also designed to be machine, operating system, network architecture, and transport protocol independent and is currently running in Windows, UNIX and in all Linux flavors [41].

2.2.1 NFS Architecture

The basic architecture of NFS is built on top of the remote file service protocol [42]. In its model, clients can transparently access files that are managed by a remote server, without knowing the actual location of these files. Instead, clients communicate through an interface to a file, which is made of various file operations, similar the interface offered by a conventional local file system. However, in NFS, the NFS server is responsible for implementing those operations. This model is described graphically in Figure 2.1

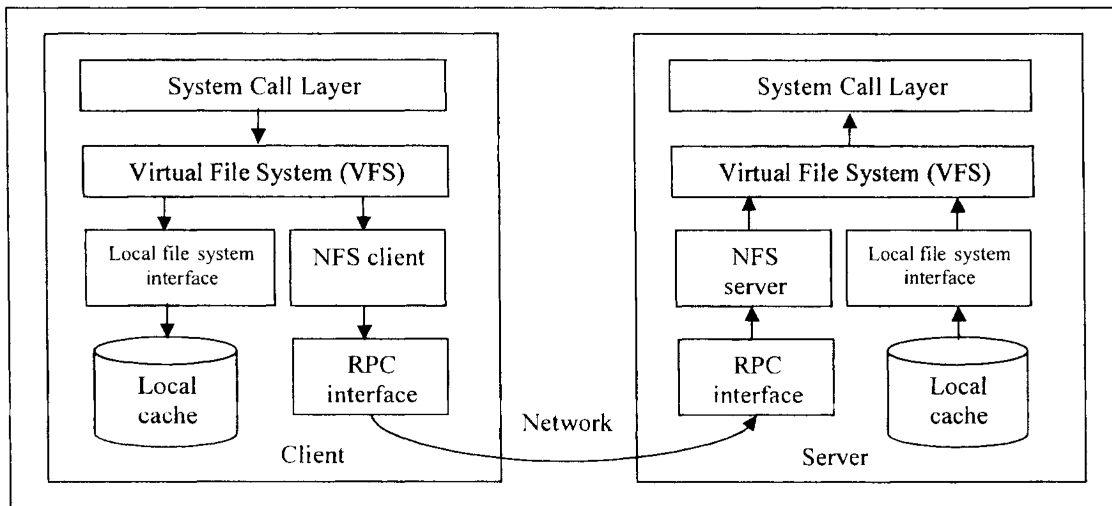

Figure 2.1: NFS Interface to the Operating System

A client accesses the NFS file system using the usual system calls that are integrated in its local operating system. However, the local UNIX file system interface is replaced by an interface to the Virtual File System (VFS), which by now is a standard for interfacing to different distributed file systems [48,49]. In particular, operations on the VFS interface that are made by the client, are either passed to a local file system if the requested files resides locally, or passed to a separate component known as the NFS client, which takes care of handling access to files stored at a remote server. In NFS, the remote file service communication is done through RPC. That is, the NFS client implements the NFS file system operations as RPCs to the server.

On the server side, the running NFS service does the client calls translation. That is, the NFS server is responsible for handing the incoming client requests and converts the RPC calls to regular VFS file operations. Again, the VFS layer is the standard interface in which it is responsible for implementing a local file system in which the actual files are stored.

An important point of NFS is that it is independent of local file systems implementations. That is, NFS is not aware if the operating system at the client or the server is a real UNIX file system. The only important point is that these file systems are compatible with the file system model offered by NFS.

2.2.2 NFS Communication Modes

NFS requires that there exists at least one running process at the server, in order to establish the connections with the clients. Server processes may be duplicated if the server becomes too busy and cannot handle all clients' requests [40]. Moreover, one important NFS feature compared to other distributed file systems is the fact that servers could be stateless [54]. In other words, the NFS protocol doesn't require that servers know about the client's status. For example, when a stateless server crashes, there is no need to switch to a recovery phase to bring the server to the previous state or up to date with its clients. This approach was implemented in NFS version 3, but has been changed in NFS Version 4 as we will see in section 2.2.7. The main advantage of the stateless protocol is its simplicity with a low communication overhead. However, stateless protocol [54] does not provide any guarantees if the client's request has actually been carried out or not [54]. Moreover, the problem of concurrency control and locking mechanisms cannot easily be implemented inherently within a stateless server [10]. Therefore, a separate lock manager is used to handle the server's maintained state with its clients.

2.2.3 NFS Name Space

As mentioned earlier, the idea behind the NFS naming scheme is to provide clients a complete transparent access to a remote file system maintained by a server. This transparency is achieved by letting a client to "*mount*" a remote file system into a local directory. The server is said to "*export*" a directory and its entries available to clients.

Instead of mounting an entire file system that is exported from the server, clients can mount only part of a file system. However, users do not share name spaces. For example, a file named */csi5311/user1* at client A might be named as */university/ottawa/user1* at client B. The drawback of this approach in a distributed file system is that sharing files becomes much harder in the sense that every client would have its own structured mount point. Another point that is worth mentioning is that an NFS server can itself mount directories that are exported by other servers. However, a server is not allowed to export mounted directories to its clients. Instead, a client will have to explicitly mount such a directory from its original server.

2.2.4 File Locking in NFS

For a distributed file system in which servers can be stateless, locking files can be complicated [54]. In earlier NFS versions, file locking has traditionally been handled by means of a separate protocol [40]. However, file locking using the NFS locking protocol has never been very popular due to the poor performance, or even faulty implementations. In later implementations, as we will see later, NFS version 4 integrates file locking into the file access protocol [41]. It is expected that this approach will make it

simpler for clients to apply file locking and avoid all the locking downsides of the earlier NFS versions.

File locking in a distributed file system is complicated by the fact that clients and servers may fail while locks are still held [54]. Proper recovery then becomes important to ensure consistency of shared files. There are only four operations related to locking a file:

Lockt: *test whether a conflicting lock has been granted.*

Lock: *create a lock for a range of bytes.*

Locku: *remove a lock from a range of bytes.*

Renew: *renew the lease on a specified lock.*

The *Lockt* operation is used to test whether a conflicting lock exists. For example, a client can test whether there are any read locks granted on a specific range of bytes in a file, before requesting a write lock for those bytes. In the case of conflict, the requesting client is informed exactly who is causing the conflict and on which range of bytes.

Operation *Lock* is used by the client to request an actual read or write lock on a consecutive range of bytes in a file. It is non blocking operation; if the lock cannot be granted due to another conflicting lock, the client gets back an error message and has to check with the server at a later time.

Locku operation removes a lock from a file. However, unless a client renews the lease on its granted lock using the Renew operation, the server will automatically remove it. In other words, each lock is granted for a specific time and has an associated lease. This approach helps in recovery after failures.

2.2.5 Caching and Replication in NFS

Caching in NFS version 3 has been implemented outside of the protocol [41]. This approach has led to data inconsistency and, at best, cached data could be out of date for a few seconds compared to the data stored at a server. NFS version 4 solves some of these consistency problems.

As in the model, each client can have a memory cache that contains data previously read from the server. In addition, there may also be a disk cache that is added as an extension to the memory cache, using the same consistency parameters.

NFS supports two approaches for caching file data. The first approach is when a client opens a file and caches the data it obtains form the servers as the result of the read operation. In addition, a write operation can be done in the cache as well. Once a file has been cached, a client can keep its data in the cache even after closing the file. When the client closes the file, NFS requires that if modifications have taken place because of cache-writes, the cached data must be flushed back to the server. Moreover, NFS requires that whenever a client opens a previously closed file that has been cached; the client must immediately revalidate the cached data. Revalidation is done by checking when the file was last modified, and then updating the cache in case it contains outdated data.

2.2.6 Fault Tolerance in NFS

In this section, we refer to fault tolerance as how NFS would handle crashes and recover from a faulty state. However, we refer to fault tolerance in other distributed file systems as a way to continue serving other clients while one of the servers or services is not functioning.

Until the most recent NFS version, fault tolerance was not seriously considered [41]. The reason for this is that the earlier NFS implementations did not require servers to be stateful. However, by altering the stateless design, fault tolerance and recovery is critically considered in NFS version 4. File locking in NFS version 3 was handled through a separate server [45]. In this way, the stateless nature of the NFS protocol could be maintained and problems related to fault tolerance were handled by the lock server. However, with the introduction of file locks in NFS version 4, it became necessary to integrate the basic mechanism of handling the client and server crashes as part of the NFS protocol.

To resolve client crashes in NFS version 4, the server issues a lease on every lock it grants. When the lease expires, the server removes the lock, thus releasing the associated files. However, there are cases where the lock should be maintained. In order to prevent the server from removing a needed lock, the client should renew its lease before it expires. For this purpose, NFS provides a renew operation explained before. Furthermore, there are situations in which locks are removed even if the client is not crashing, but because of other problems. In particular, false lock removal may happen if the client cannot reach the server to renew a lease, for example, because the network is temporarily unavailable. Until now, no special measures are taken to handle these situations, and the only resolution to the problem is by restarting the locking and NFS services [41].

When the server crashes and subsequently recovers, it will most likely lose information on locks it granted to clients. In NFS version 4, this situation is solved by allowing the protocol to enter a grace period in which a client can reclaim locks that were

previously issued to it. In this way, the server builds up its previous state with respect to locks. Note that this lock recovery is independent of the recovery of other lost data. During this grace period, only requests for reclaiming locks are accepted. Normal locks are denied until the grace period is over [41].

At this point, using leases introduces other potential problems. For example, leasing requires that the client and the server have their clocks synchronized. If the server state that a lease expires at a certain point of time *T*, then the client will need to have the same notion of what the actual time is as the server. On the other hand, if the sever state that a lease expires after some duration *D*, the time it takes to send the lease to the client will have to be known. These timing issues may not be easily solved in wide area systems [45]. Another problem concerns the reliability of sending a lease renewal message to the server. If message delivery fails or is delayed, the lease may undesirably expire. In either case, the sever needs to explicitly issue a new lease to the client before the latter can continue to use a file.

2.2.7 Enhancements to NFS version 4

In this subsection, we illustrate the main enhancements of NFS version 4 [40]. The main intention for this version is to consider Wide Area Networks (WANs), and therefore, most of the enhancements are in the locking and replication mechanisms [41]:

NFS v.2 and v.3 are stateless protocols. However, NFS v.4 introduces additional protocol states [40]. An important reason is that NFS version 4 is expected to work across WANs as well. Using NFS Version 4, a client uses the states to notify an NFS Version 4 server of its intentions on a file, such as *locking*, *reading*, *writing*, etc. An NFS Version 4 server can return information to a client about what other clients have intentions on a file

to allow a client to cache file data more aggressively via delegation. In order to ensure keeping the states consistent, more sophisticated client and server reboot recovery mechanisms are being built using the NFS Version 4 protocol.

NFS Version 4 introduces some support mechanisms for the byte-range locking and share reservation. Locking in NFS Version 4 is *lease-based*, thus, an NFS Version 4 client must maintain contact with the NFS Version 4 server to ensure the continuity of its open and lock leases.

Furthermore, NFS Version 4 introduces a file delegation paradigm. An NFS Version 4 server can allow an NFS Version 4 client to access and modify a file in its own cache without sending any network requests to the server, until the server indicates via a callback that another client wishes to access a file. This reduces the amount of traffic between NFS Version 4 client and server considerably in cases where no other clients wish to access a set of files concurrently.

Up until NFS version 3, two consecutive RPC requests were necessary for most of the NFS operations. For example, in order to read data from a file, a client normally has to look up the file-handle using the *lookup* operation, and then issues a *read* request. NFS Version 4 uses compound RPCs [41]. An NFS Version 4 client can combine several traditional NFS operations (LOOKUP, OPEN, and READ, for example) into a single RPC request to carry out a complex operation in one network round trip, as shown in Figure 2.2. This approach becomes advantageous when considering NFS in Wide Area Networks.

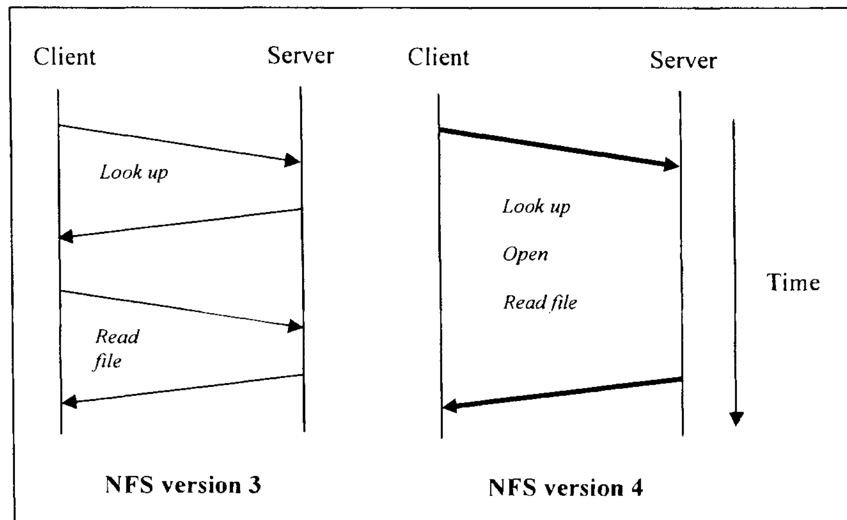

Figure 2.2: Lookup Operation in NFS v3.0 vs. NFS v4.0

2.3 CODA File System

Coda was designed to be a *scalable* and *highly available* distributed file system. An important goal was to achieve naming and location transparency so that the system would appear to its users very similar in all machines in the network [4,8,39]. Coda is a descendant of version of the Andrew File System (AFS), which was also developed at Carnegie Mellon University, and inherits many of its architectural features.

Looking at the conceptual design of Coda, we see that nodes are partitioned into two groups. One group consists of a relatively small number of dedicated file servers called Vice. The other group consists of a very large number of workstations that give users and processes access to the file system, called Virtue. Every Virtue workstation hosts a user-level process called Venus, which is responsible for providing access to the files that are maintained by the Vice file servers [4]. Venus's function is similar to that of

an NFS client. In Coda, high availability is achieved by Venus, which is responsible for allowing the client to continue operation even if access to the file servers is (temporarily) unavailable [3,4].

Like in NFS, there is a separate Virtual File System (VFS) layer that receives all calls from client applications, and forwards these calls either to the local file system or to Venus. Venus, in turn, communicates with Vice file servers using a user-level RPC protocol.

In terms of running processes, there are three different processes running on the server side [7]. The great majority of the work is done by the actual Vice processes, which are responsible for maintaining a local collection of files. In addition, trusted Vice machines are allowed to run an authentication server, as a separate process. Finally, update processes are used to keep meta-information on the file system consistent at each Vice server.

2.3.1 CODA Architecture

Coda appears to its users as a traditional UNIX-based file system. It supports most of the VFS operation calls. However, unlike NFS, Coda provides a globally shared name space that is maintained by the Vice servers. Clients have access to this name space by means of a special subdirectory in *their local name space* [4], such as */Coda*. Whenever a client looks up a name in this subdirectory, Venus ensures that the appropriate part of the shared name space is mounted locally.

To understand *how* Coda can operate when the network connections to the server have been disconnected, we analyze a simple file system operation by giving an example: consider a simple system call (e.g. file read operation) generated by a client to a file that

resides in */Coda*. Recall that clients see the file system */Coda* as if it is local to the system. The system call will communicate with the VFS layer in the local kernel, and when the VFS realizes that the request is for a file that resides in */Coda*, the call is handed to the Coda file system module in the kernel, which in turn directs the call to Venus. Then, Venus explores more than one path in order to answer this kernel call [7]: First it tries to reply by looking in the local cache (the cache manager). If it can not find the requested file, it searches Vice servers through RPC calls over the network. Figure 2.3 explains the procedure in a more schematic way.

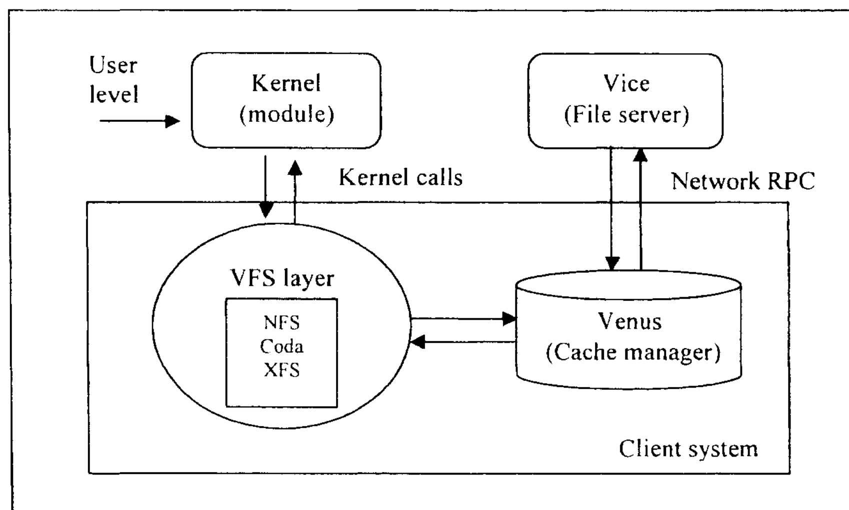

Figure 2.3: Client/Venus/ Vice Interaction in Coda File System

When Venus passes the read request to Vice for the first time, Venus fetches the entire file from the server, and then stores the file in the cache. However, if the file is opened a second time, it will not be fetched from the Vice again, but the local copy will be available for use immediately. Directory files as well as all the attributes (ownership, permissions and size) are all cached by Venus, and all operations are proceed without contacting the server if the files are present in the cache. However, if the file has been

modified and it is closed, then Venus updates the servers by sending the new file. In particular, other operations which modify the file system, such as making directories, removing files or directories and creating or removing (symbolic) links are propagated to the servers.

The above mechanism works well while the network is stable. However, in the presence of failures, things may go wrong. In particular, if Vice servers do not respond to Venus requests, then the latter declare disconnection and start servicing in disconnected mode [7]. Disconnected mode starts to operate when there is no network connection to any server which has the files. Typically, this happens for laptops when taken off the network, or during network failures. In disconnected mode, all system calls are attempted to be answered from the cache manager. Else, the service will return cache-misses to the requestor.

2.3.2 CODA Communication

Intercrosses communication in Coda is carried out using the Remote Procedure Calls (RPC) [10]. However, the RPC2 system for Coda is much more sophisticated than the traditional RPC systems such as ONE RPC, which is used by NFS [42]. RPC2 offers more reliable RPC functions on top of the unreliable UDP protocol. In particular, every time a remote procedure call is issued, the RPC2 client code starts a new thread that sends a request to the server and subsequently blocks until it receives an answer. As processing requests may take a long time to complete, the server regularly send back messages to the client to let it know that it is still working on the request. Nevertheless, if the server dies, sooner or later this thread will notice that the messages have expired and report back a timeout failure to the calling application.

Another feature of RPC2 that makes it different from other RPC systems is its support from multi-casting. Coda uses this feature to effectively keep track of which clients have a local copy of what file. In particular, when a file is modified, a server invalidates its local copies by notifying the appropriate clients through an RPC. However, if a server can only notify one client at a time, invalidating all clients may take some time. Moreover, there could be a problem where an RPC may fail. In other words, invalidating files in a strict sequential order may be delayed considerably because the server cannot reach a possibly crashed client, but will give up on that client only after a relatively long expiration time, while other clients will still be reading from their local copies. Coda solves this problem by using Parallel RPCs, which are implemented by means of the MultiRPC system [10]. MultiRPC is implemented by executing multiple RPCs in parallel. This means that the caller explicitly sends an RPC request to each recipient. However, instead of immediately waiting for a response, it defers blocking until all requests have been sent, as shown in Figure 2.4.

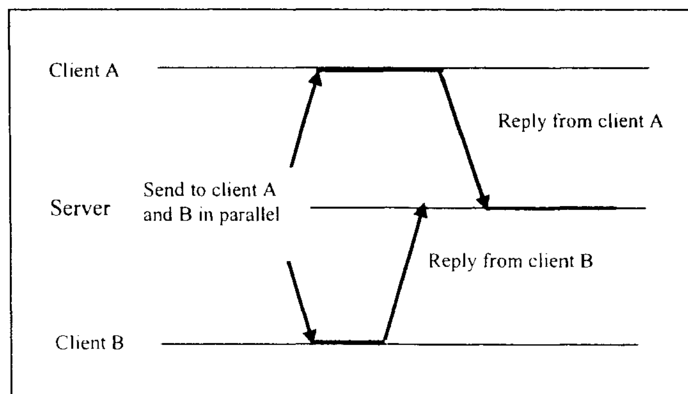

Figure 2.4: Parallel RPC Replies in Coda File System

Instead of invalidation each copy one-by-one, the server sends an invalidation message to all clients in parallel. As a consequence, all non-failing clients are notified in

the same time as it would take to do an immediate RPC. Also, the server notices the usual expiration time that certain clients are failing to respond to the RPC, and can declare such clients as being crashed.

2.3.3 CODA Name Space

In Coda, files are grouped into units referred to as volumes [4], which are similar to the general UNIX disk partitions. Usually a volume corresponds to a collection of files associated with a user. Volumes are important for two reasons. First they form the basic unit by which the entire name space is constructed. This construction takes place by mounting volumes at mount points, similar to the mount process done in NFS. However, unlike NFS, only root nodes can act as mounting points (i.e. clients can mount only the root of a volume). The second reason why volumes are important is that, as we will see later, they form the unit for server-side replication [35].

It is important to note that, unlike NSF, when a volume from the shared name space is mounted in the clients' name space, Venus follows the structure of the shared name space. That is, the name appears the same everywhere in all clients. This structure makes it easier to manage a large number of clients in a distributed environment.

2.3.4 Disconnected Operations in Coda

The problem that Coda wants to solve is that in large distributed file system, it may easily happen that some or all of the file servers are temporarily unavailable. Such unavailability can be caused by a network or a server failure, but may also be the result of a mobile client disconnecting from the file service. Coda tries to make it possible to use

files while disconnected and reintegrate later when the connection is established again, provided that the disconnected client has all the relevant files cached locally.

As we mentioned above, a client is said to be disconnected with respect to a volume if it cannot contact any of its accessible servers that hold a copy of the file. In most file systems, a client is not allowed to proceed unless its can contact at least one server. A different approach is followed in Coda. There, a client will simply report to using its local copy of the file that it had when it opened the file at a server.

The main issue that needs to be solved to make disconnected operation a success, is to ensure that a clients cache the needed files prior to disconnection. In Coda, caching the needed data prior to disconnection is called *Hoarding*. The state transition diagram in figure 2.5 shows how clients in Coda interact with file servers both during connection and disconnection situations.

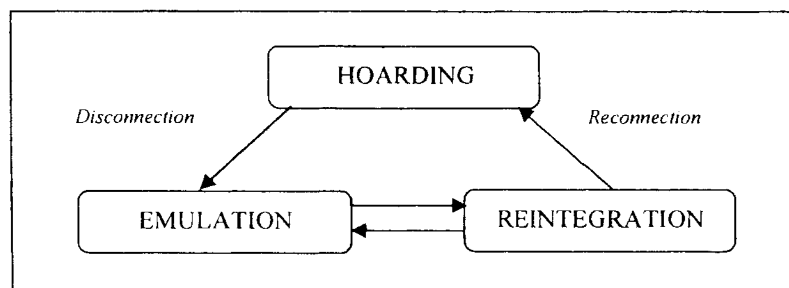

Figure 2.5: State Transition in Coda File System

Normally, a client will be in the *Hoarding* state. In this state, the client is connected to at least one server that contains a copy of the volume. While in this state, the client can contact the server and issue a file requests to perform its work. Simultaneously, it will also attempt to keep its cache filled with the useful data.

At a point of time, the client will be unable to communicate with any of the file servers, bringing the client to an *Emulation* state in which the client will emulates the

behavior of a file server. This means that all file requests will be directly serviced using the locally cached copy of the file. Finally, when reconnection occurs, the client enters the *Reintegration* state in which it transfers updates to the server in order to make them permanent. It is during reintegration that conflicts are detected and, where possible, automatically resolved. It is possible that during reintegration the connection with the server is lost again, bringing the client back into the *Emulation* state.

This technique has shown a vast improvement over the traditional cache management techniques. However, the technique cannot guarantee that a client's cache will always contain the data the user will need in the near future. Therefore, there are still occasions in which an operation in disconnected mode will fail due to inaccessible data.

2.3.5 Semantics of CODA File Sharing

To explore the internals of Coda's file sharing scheme in more depth, consider the following example: When a client successfully opens a file *f* for writing, an entire copy of *f* is transferred to the client's machine and the server records that the client has a copy of *f*. Now suppose client *B* wants to open *f* as well, it will fail, because the server has a record and that client *A* might modify *f*. However, if client *A* opened *f* for reading only, client *B* can also successfully get a copy from the server for reading. An attempt by *B* to open for writing would succeed as well, but later requests from other clients will fail.

Let us now consider what happens when several copies of *f* have been stored locally at various clients. Given what we have just said, only one client will be able to modify *f*. If this client modifies *f* and then closes the file, the file will be transferred back to the server. However, each other client may continue reading its local copy despite the fact that the copy is actually changed on the server.

The reason for this inconsistency is that the session is treated as transaction in Coda [1]. For this purpose, Coda uses a simple versioning scheme. That is, each file has an associated version number that indicates how many updates have taken place since the file was created [4]. In this case, when a client starts a session, all the data relevant to that session are copied to the client's machine, including the version number associated with each file. Now, assume that while one or more file operations are being executed at the client, and the file system switches to the disconnected mode. At that point, Venus will allow the client to continue and finish the execution of its sessions as if nothing happened. Later, when the connection with the server is established again, updates are transferred to the server in the same order as they took place at the client. So, when an update for file f is transferred to the server, it is tentatively accepted if no other process has updated f while the client and server were disconnected. Such a conflict can be easily detected by comparing version numbers:

Let V be the version number of f acquired from the server when the file was transferred to the client. Let N be the number of updates in that session that have been transferred and accepted by the server after reintegration. Finally, let $Vnow$ denotes the current version number of f at the server. Then, a next update for f from the client's session can be accepted if and only if $Vnow + 1 = V + N$.

In other words, an update from a client is accepted only when that update would give the next version of file f. This means that only a single client can do the update. However, in a distributed environment, conflicts arise if f is being updated from other clients having the same copy. When a conflict occurs, the updates from the client's sessions are undone, and the client is forced to save its local version of f for manual

reintegration. In term of transactions, the session cannot be committed and conflict resolution is left to the user.

2.3.6 Caching and Replication in CODA

In this section, we shall concentrate on a client-side caching, which is an important feature in Coda's disconnected operation [46]. Then, we consider the server-side replication of volumes.

2.3.6.1 Client Caching

As seen from the previous discussion, client-side caching is crucial to the operation of Coda and provides a higher degree of fault tolerance as the client becomes less dependent on the availability of the server. However, unlike many other distributed file systems, cache coherence in Coda is maintained by means of callbacks [46]. For each file, the server from which a client had fetched the file keeps track of which clients have a copy of that file cached locally. The server records a callback promise for a client, and when a client updates its local copy of the file for the fist time, it notifies the server, which in turn, sends an invalidation (callback break) message to the other clients that hold the same copy of the file. This invalidation message is called a callback break [3], because the server discards the callback promise it held for the client it just sent an invalidation message. The interesting feature of this scheme is that as long as a client knows it has an outstanding callback promise at the server, it can safely access the file locally. In particular, suppose a client opens a file in its cache. It can use that file as long as the server still has a callback promise on the file for that client. In particular, there is no need to transfer the file from the server to the client again. However, the client have to

check with the server if that promise still holds every time it starts a new session with this file.

Figure 2.6 shows how the callback model works in Coda. When client *A* starts a read session, the server records a callback promise. The same happens when client *B* starts its own write session. However, when *B* closes the session, the server breaks its promise to callback client *A* by sending a callback break. Subsequently, when client *A* closes its session, nothing special happens, the closing is simply accepted.

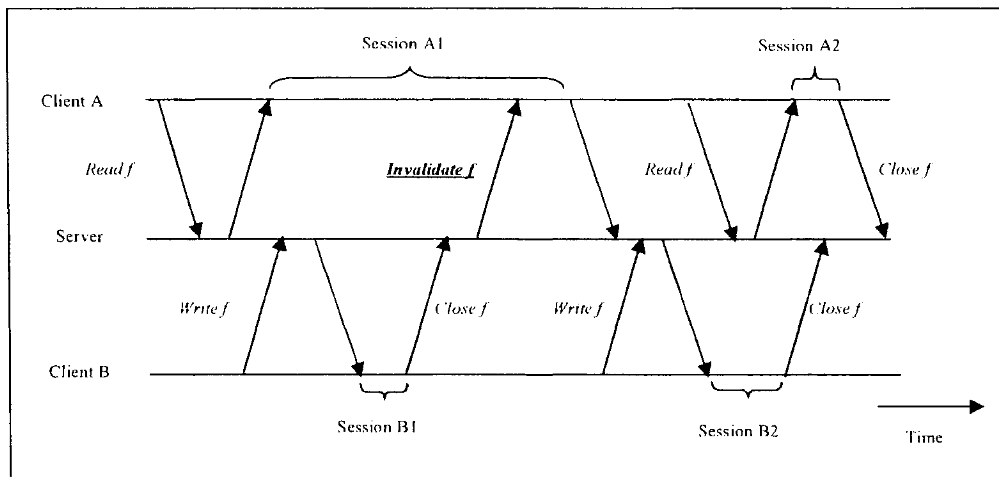

Figure 2.6: Callback Model in Coda. An invalidation was sent to Client A to repeat the operation on file f while client B committed its operation successfully.

The result of the presented mechanism is that when client *A* wants to initiate a new session, it will find its local copy of *f* to be outdated, so that it will have to bring the latest version from the server. In contrast, when client *B* opens a session in future, it will notice that the server still has an outstanding callback, so that client *B* can simply reuse the already cached copy it still has from the previous session.

2.3.6.2 Server Replication

Along with client caching, Coda allows file servers to be replicated in the unit of volumes [3,4]. The collection of server that have a copy of a volume, are known as a Volume Storage Group (VSG) [4]. In the presence of server's failures, a client may not have access to all servers in a volume's VSG. A client's Accessible Volume Storage Group (ASVG) consists of those file servers in the AVSG that the client can contact. However, the client is completely disconnected form the Coda file system if its AVSG list is empty.

Coda uses a replicated-write protocol to maintain consistency of a replicated volume. In particular, it uses a variant of Read-One, Write-All (ROWA). That is, when a client reads a file, it contacts one of the members in its Accessible Volume Storage Group (ASVG), and when closing a session on an updated file, the client transfers it in parallel to each member in the AVSG. This parallel transfer is carried out by means of multiRPC[2].

This scheme works fine as long as there are no failures. However, in the presence of failures, things may go wrong and consistency control becomes necessary. For example, consider a volume that is replicated across three servers *FS1*, *FS2*, and *FS3*, and a network disconnection is taking place in a way that not all clients see all file-servers. In particular, the SVG that client *C1* sees (i.e. its ASVG) resides on *FS1* and *S2*, whereas client *C2* can access *FS3*. Both *C1* and *C2* will be allowed to open a file *f* for writing, update their respective copies in cache and transfer their copy back to the members in their ASVG. Obviously, there will be different versions of *f* stored in the VSG. The question is how this inconsistency can be detected and resolved? The solution adopted by

Coda is that each file server maintains a Version Vector (CVV) for that VSG in the system. Returning to our three servers' example, CVV is initially equal in each server. Then, when client *C1* reads from one of the servers in its ASVG, say *FS1*, it also receives the value of CVV. After updating *f*, client *C1* multicasts *f* to each server in its ASVG, that is, *FS1* and *FS2*. Both servers will then record that their respective copy has been updated, and therefore, update their CVV values (remember that CVV in *FS3* is not updated because *C1* has no access to it). On the other side, client *C2* can access files in server *FS3*, and subsequently updates *f* as well. When *C2* closes its session and transfers the updates to FS3, server FS3 will update its CVV.

When the network disconnection is restored, the three servers will need to reintegrate their copies of *f*. By comparing their version vectors, they will notice that a conflict has occurred that needs to be repaired. In many cases, conflict resolution can be automated in an application-depended way [4]. However, there are also many cases in which users will have to assist in resolving a conflict manually.

2.4 Other Distributed File Systems

There are many other distributed file systems than the ones we review, such as GFS and MogileFS, though many of them are almost similar in their design to NFS and Coda. In this section, we shall take a brief look at three other file systems, namely: InterMezzo, Ficus and OceanStore file systems. We will mainly concentrate on the general principle underlying each of them and show how they resolve conflicts in their systems.

2.4.1 InterMezzo File System

InterMezzo [REFF] can be considered as a filtering file system layer, which sits between the virtual file system and a specific file system such as ext3, ReiserFS, JFS, or XFS [1]. InterMezzo has a distributed file system functionality with a focus on high availability. It uses InterSync, which is a client-server system that synchronizes collections of folders between a server system and the clients.

Unlike Coda, which uses the "callback" method by the server to notify clients of any changes in the file system, InterSync uses a "pull" method to periodically check the server for changes and then pull changes and reintegrates them into the client file system. In particular, the changes are recorded on the server by InterMezzo, by maintains a kernel modification log (KML) [2] whenever the file system is modified. The modification log makes it possible to collect the changes in the server file system without scanning for differences. Client's InterSync then synchronizes the file system by fetching the KML, which is simply a file, using the HTTP (Hyper Text Transfer Protocol). Subsequently, InterSync processes the records in the KML and when it comes across a file modification record, it fetches the file from the server again using the HTTP protocol.

InterMezzo, as in Coda, synchronizes files when they are closed after writes [2]. When desired, more frequent synchronization is possible through having a kernel based I/O daemon [2], which can force synchronizing records when files remain open for a long time. The daemon can also limit the synchronization frequency when files are frequently opened and closed, resulting in an undesired network traffic.

Further KML optimizations exist, such as not creating directories which are removed soon afterworlds, but these optimizations are much of much smaller impact on

the performance. InterSync can also handle a push based reintegration which sends the KML to a peer first. The peer reintegrates all records but does not do the actual files fetching. Instead, the client will file uploads for all records in the KML that affected file contents. The purpose of this mechanism is to make the system a client driven and does not require the server to be RPC dependent.

2.4.1.1 Conflict Resolution in InterMezzo

Like in any other distributed file system, conflicts can occur when updates are made on the same object on two separate systems, and then they try to reintegrate those changes. In InterMezzo, conflicts are resolved in the following way: At first, the system checks if the changes are done on only one of the systems since the last reintegration. If this is the case, then the system can blindly reintegrate all files. However, if the check is false, then the second check done by InterMezzo is to verify a file's version. In particular, a file version in InterMezzo is composed of the timestamp and the size of the file. As in Coda, the idea behind checking the version is that if the versions of the affected objects match, one assumes that the update can be applied. In most cases, the timestamp and the file size are sufficient to make the file unique, but it is possible that objects still keep the same timestamp and size while they are modified. Therefore, if this check also fails, then InterMezzo considers this as a possible fake conflict [1], and tries to fix it according to following three predefined policies:

Mobile policy: this policy is applied when InterMezzo synchronizes between mobile devices, which are basically the clients, and servers. It keeps the conflicts on the mobile device and let the mobile object move out of the way when conflicting server objects are introduced.

HA policy: This policy is used to synchronize between high availability fail-over servers. When reconnections happen there is a failed node re-joining the active node. The conflict resolution policy here is to let the active node prevail and we move conflicting objects on the re-joining failed node out of the way. If the failed node is the client this policy coincides with the mobile policy.

Re-synchronization policy: This is used when re-synchronizing systems when available KML record is not sufficient to perform the synchronization or after KML has been truncated.

As seen from the policies, InterMezzo regards file conflicts as an update/update conflict. In particular, this happens when file data is updated in multiple locations and require corrective action. However, other distributed file systems, such as Coda, usually classify conflicts into four categories:

Name/Name conflicts: This category identifies the case when a file with the same name is created on multiple servers.

Update/Remove conflicts: This category identifies the case when one system removes a file an object while another modifies it.

Update/Update conflicts: This category identifies the case when more than one server updates the same file.

Rename/Rename conflicts: This category identifies the case when a file is renamed by multiple servers to different destinations.

2.4.2 Ficus File System

Ficus [30] is a UCLA developed distributed file system. It shares many characteristics with Coda and InterMezzo file systems. However, it differs in its conflict

resolution and reintegration policies. As in InterMezzo, once a single replica has been updated, the system notifies all accessible replicas that a new version of the file exists via update propagation. Then, replicas use the "pull" method to get the updated file.

Ficus merges sub-trees of files and directories into volumes [31]. A volume can be replicated by a process known as *reconciliation* which runs periodically on each volume replica during a normal operation. The process works by comparing all files and directories of the local volume with a remote replica of the volume, pulling over missed updates and deleting concurrent update conflicts. Let us now discuss how Ficus deals with these conflicts and present each type of these conflicts in more details.

Unlike InterMezzo, Ficus classifies conflicts into more than one category. It also deals with directories' conflicts, and because of the importance of the integrity of directories, directory as well as remove/update conflicts receive special handling [30]..

Directory Conflicts: The reason why directories' consistency is important is that the integrity of a UNIX file system depends on its directories. In particular, if a directory cannot be used because it has received conflicting updates, a portion of the file system's name space may become inaccessible [30]. Thus, conflicts in directories are very serious in a way that either they must not occur often, or they must be resolved automatically. Ficus directory conflicts are repaired automatically during reintegration.

The automatic reintegration mechanism for directories in Ficus is done by examining all entries in both versions of the directory in conflict, determine which entries are common to both, and, for an entry that is present in only one directory, determine whether the file was created or deleted during partition. These checks are possible because Ficus keeps track sufficient information to distinguish precisely the patterns of

file entry additions and deletions while partitioned (similar to KML in InterMezzo), which in turn allows all possible conflicting updates that will be triggered.

- *Directory Creation Conflict*: creation of different files with the same name results in a directory with two identical, effectively indistinguishable names. Ficus detects that the two files are actually different, but upon reintegration, the underlying Unix File System (UFS) that Ficus is based on, does not permit different files to have identical names. Therefore and upon reintegration, Ficus appends unique suffixes to each name and invokes name conflict resolvers to handle the situation. If automatic reintegration fails, Ficus notifies the file owner, who must either rename or remove one of the files.
- *Remove/Update Conflicts*: These conflicts are quite similar to the remove/update conflict introduced in Coda, but Ficus handles it differently. Since Ficus logs operations during network partitions, it is able to recognize such cases. Similar to Coda, it uses the stored version vectors to compare between files and then resolve the conflict. It then moves the conflicting file into a special directory called an orphanage [31]. Each volume has its own orphanage directory located under its root directory. When the reconciliation process moves a file into an orphanage, electronic mail is sent to the owner notifying him, allowing him to decide whether to keep the updated file or discard it.
- *Update/Update Conflicts*: Some update/update conflicts for non-directory files can be resolved automatically and some cannot. Ficus tries to resolve the conflicts by invoking various resolvers [31]. In addition, it allows individual users to specify how they would like their conflicts resolved, but also provides a default

system for resolving conflicts when users have not specified their own methods, or when the users' methods fail.

2.4.3 OceanStore File System

OceanStore [25,26], developed at the University of California at Berkeley, is a global-scale persistent data store that provides a consistent, highly-available, and durable storage utility on top of an infrastructure that is comprised of un-trusted servers. One main aim of OceanStore is to scale to millions of individual servers, each cooperating to provide file service. In fact, we may consider OceanStore as a peer-to-peer system in a way that any computer can join the infrastructure to contribute storage or to provide local user access [26]. Users need only subscribe to a single OceanStore service provider, although they may consume storage and bandwidth from many different providers.

The fundamental unit in OceanStore is the persistent object. At the lowest level, OceanStore converts an object blocks into fragments, which are distributed over many servers, and only a fraction of the fragments are needed to reconstruct the original block. However, this coding technique introduces several concerns, such as data integrity: The system must ensure that a reconstructed object is an exact copy of the original, despite failures or corruption of fragments. OceanStore resolves this issue, as we shall see later, by naming each object and its associated fragments by a globally unique identifier (GUID) [27]. Another issue introduced by this fragmentation is location: When reconstructing an object, the system must locate enough servers storing fragments for that object. OceanStore resolves this problem with *Tapestry* [25]. Tapestry is an overlay routing and location layer on top of TCP/IP that maps GUIDs to individual servers. A server can advertise many GUIDs, and any GUID can be advertised by multiple servers,

for example, in case of the same file is available in multiple servers. Utilizing Tapestry, OceanStore can easily locate the servers containing fragments for a given GUID. Finally, because reconstructing an object from archival fragments is computationally intensive, it should be performed only when necessary. OceanStore servers that have already reconstructed objects from the archive can advertise them through Tapestry. These objects are secondary replicas of the data, distinct from the primary replicas stored on the inner ring. An OceanStore client can retrieve an object either by locating a replica from a nearby server or reconstructing it from the archive.

2.4.3.1 Object Replication in OceanStore

Backed-up files of OceanStore are read-only [25]. OceanStore overcomes this deficiency through versioning, the concept that every update creates a new version of a data object. To support versioning, each OceanStore object has a special name called an active GUID. The system then provides a fault-tolerant mapping from an object's active GUID to the GUID of the most recent read-only version of that object.

Files in the OceanStore are modified through updates, which contain information about what changes to make to a file. In particular, every update to an OceanStore file creates a new version. OceanStore objects exist in both active and archival forms. An active form of an object is the latest version of an updated file, but it is not the permanent version. An archival form represents the permanent, read-only version of the object. Archival versions of objects are spread over hundreds or thousands of servers, making the OceanStore as a highly redundant deep archival storage.

2.4.3.2 *Basic Routing and Servers Manipulation*

Tapestry functions [27] as an overlay on top of IP, using a distributed, fault tolerant data structure to explicitly track the location of all objects. Each network node can act either as *a server* that stores objects, *a client* that initiates requests, or *a router* that forwards messages. Like objects, nodes are assigned unique identifiers called NodeIDs that are location and semantics-independent. Tapestry uses local-neighbor maps to incrementally route messages to their destination NodeID, digit by digit as illustrated in Figure 2.7, where each routing step in any path can be carried out by several hops. This is mainly due to the following constraint where each hop should match the same growing suffix of the previous hop. Tapestry keeps the path to the three closest matching hops and routes to the neighbor with lowest network latency.

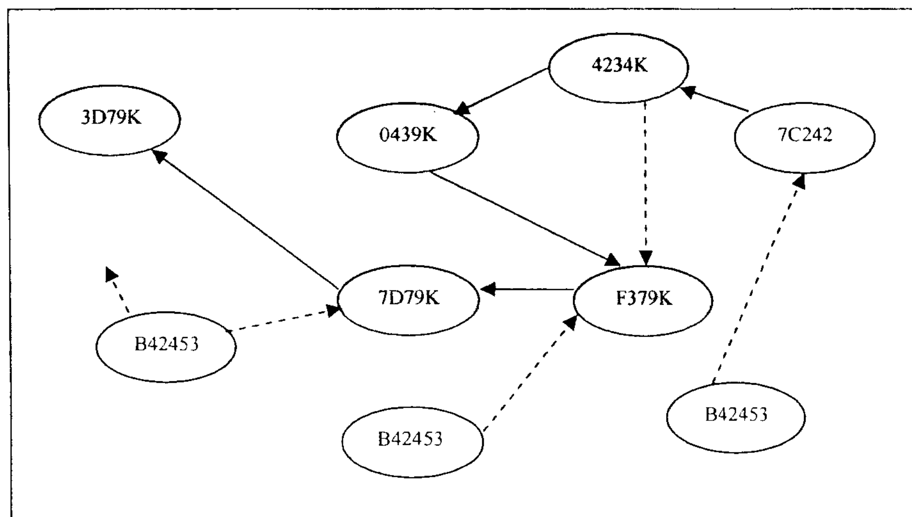

Figure 2.7: A potential constructed path from the sender (7C242) to the receiver (K9837) by routing thru the path ****K → ***9K → **79K → *D79K → 3D79K

The model of how to locate a destination in OceanStore can be presented as follows: When an OceanStore server inserts a replica into the system, Tapestry publishes its location by depositing a pointer to the replica's location at each hop between the new

replica and the object's root node. To locate an object, a client routes a request to the object's root until it encounters a replica pointer, at which point it routes directly to that replica.

To join an existing network, a node chooses a random hop by which to identify itself. It then chooses a Tapestry node that is the closest to itself. Routing to the newly chosen hop through this existing node lets the new node find other existing nodes that have similar, but incrementally longer suffixes. By mimicking and enhancing their routing tables at each level, the new node can produce its own full routing table. The new node then notifies its neighbors of its existence so that they can consider it as a more optimal neighbor [27].

OceanStore uses two separate mechanisms to handle the departure of nodes from the network. A node can simply drop from the network, so that its neighbors detect its death and update routing tables accordingly. Instead, a node can inform the nodes that rely on the departing node for routing to redirect their pointers and then to notify object servers for which it stores location information.

2.4.3.1 Caching in OceanStore

In OceanStore, data is cached at any network location, making the service time of user requests relevant to the load on the replica and the network distance to it. Moreover, because OceanStore is originally designed to scale to thousands of nodes, the cost and complexity of manually administering thousands of objects to ensure is fairly expensive. Therefore, the tasks of object administration, monitoring, event analysis, and self-adaptation are delegated to the introspective replica management subsystem, which is an architectural model that mimics biological adaptation [27]. The introspection layer also

provides tools for automatic collaboration among introspective modules on different servers. Furthermore, Introspective modules try to balance the network traffic by observing access traffic generated by nodes within a few hops. When a server detects heavy demand for an object, it creates a new local replica to handle those accesses. Otherwise, clients are directed to nearby servers if they notice a drop in the quality of service provided from their originally-assigned servers. If the heavy demand persists on the servers, the nearby clients will continue to receive service from a more distant replica. Furthermore, when the load on the replica falls below a certain threshold, its service is removed from the network by the responsible server and the resources are freed to be used by another service. If a replica suddenly becomes unavailable, the failover mechanism, which is formed by the Introspective modules, produces additional load on distant replicas. These activities satisfy the requirements for both fault tolerance and automatic repair.

2.5 Summary

To summarize the most important features of the already discussed file systems and to shed more lights on our design goals, we compare our distributed and mobile file system with existing file systems, see Table 2.1., and show their main differences and similarities. In the next chapter, we will describe each of these design characteristics in more details.

	File Systems				
	Coda	**OceanStore**	**InterMezzo**	**GFS**	**Our Design***
Distribution	X	X	X	X	X
Mobility	X				X
Conflict-resolve	Manual	Automatic (versioning)	inapplicable	inapplicable	Automatic (versioning)
Files availability (in case of a peer lost)		X			X
Persistent Caching	X	X	X		X
Caching Type	Client	Server	Client	Memory	Server
CRA	LRU		LRU	LRU	FBR + file prediction

* design specs are shown in chapter 3

Table 2.1: Distributed and Mobile File Systems: A comparative Study

Chapter 3

The System Model of a Conflict-Free File System for Mobile Clients

3.1 Overview

As we have shown in the previous chapter, file system implementations can be divided into two categories: projects that present a new file system, and projects that rely on NFS or VFS (Virtual File System). File systems, such as Coda, AFS and InterMezzo differ from our work because they are complete network file systems with their own "non-Unix" semantics [1,3]. These systems involve both client and server codes, and use client caching to improve system performance. In our mobile file system design, the code runs only at the server[1], as we will see later, and it is independent of any caching schema that may occur at the client.

In this chapter, we propose a complete distributed file system design that is directed to serve thin mobile clients, and guarantees that file operations are executed in spite of concurrency and failures. Our proposal considers all design aspects, including

[1] we avoid to place any client-side processes since we are only considering thin clients

concurrency control, caching mechanisms, the interaction between file servers and conflict resolution, as we will see in the coming section.

3.2 The Distributed File System Architecture and Design

In this section we shall describe the design of our model in more details. The framework of our model can be divided into three main stages: the *connected stage*, the *disconnected stage*, and *the re-joining stage*. Initially and while clients are connected, file system service is provided by the actual file servers. We define the connected stage as all file servers are viewable to clients and are able to answer their RPC requests. However, if one of the file server(s) does not respond to clients calls within a certain period of time, part of the file system is said to be disconnected and the system will switch to the disconnected stage. During this stage, the client will continue probing the disconnected server(s) on a regular basis. At the same time, part of the file system service is provided by the cache server, which is an independent file service as shown in figure 3.1. Note that while a client is in the disconnected state, it may still be able to contact other servers on the network. In such cases, disconnection will generally have been caused by a server failure rather than the client has been disconnected from the network. Finally, if both the file servers and the communication channel are back available, then the file system will switch into the re-joining stage. In this phase, the communication link between the file system and the previously disconnected servers is re-established and file system services can be provided by these servers again. The propagation of the files, which were updated during the disconnection phase, is performed by the re-integrator module. During this phase, the file system propagates the updates made by the nodes during the disconnected stage back to the file server(s). Upon the successful termination of the re-joining process,

the file server will switch back to the connected stage. Note that it is possible for the connection with the server to be lost again, bringing the client back into the disconnected phase.

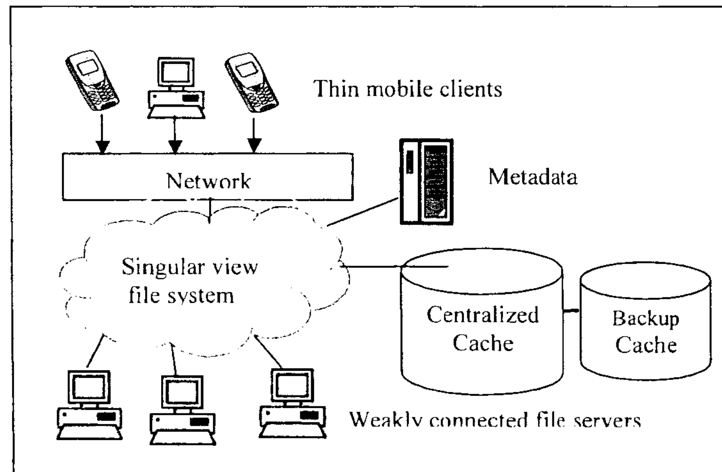

Figure 3.1: The Layout of Our File System Design

Let us now explain the framework of our proposed architecture in more details. Clearly, the role of the cache comes into play during the disconnection and re-integration phases. Returning to stage one of the model, all clients initially see the distributed file system as a single directory hierarchy. The server-side caching algorithm takes place by monitoring the file system for any file access (file reads, writes, executes, and creation). This task is accomplished by FAM (File Alteration Monitor) library, which is developed by Silicon Graphics Inc. [6]. FAM, as we will see in the next subsection, works by monitoring the file system and sending a signal to the process that has asked to be informed when the content of a directory or partition is changed. When such an event is detected, another program is initiated to copy what has been trigged by FAM to the cache server. However, before doing the actual copy, the cache server checks its contents to see if any "older" copy of the file was cached before. Older copies are those files with similar

names but probably with different contents. If such a file is found on cache, then an MD5 Checksum is done on the new file and compares it with the one that resides in cache. The purpose of this checking is to avoid unnecessary caching operations incase the two files are exactly similar (i.e., identical MD5 checksum), and therefore, minimize the network utilization. However, if the MD5 Checksum is different, then we resolve this file conflict by moving the old file to another partition called a backup cache, with a version number appended to it, starting at 1. The backup cache is a maintained repository, usually smaller than the main cache, used to save the resolved conflicts between files (e.g. files with the same name but different MD5 Checksum) and thus, avoid overwriting important files by having newer versions.

The backup cache repository is exported to the users, allowing them to view their backed-up files. It is the file owner's call to move these files from the backup cache to the actual file system.

A worth mentioning point is that if a cache-miss occurs in the cache server, the system will automatically explore the backup cache to look for any older version of the file. The user is then notified that the obtained file is actually coming from the backup-cache, and not from the actual file system or the primary cache.

3.2.1 File Alteration Monitor (FAM)

FAM daemon monitors the file system and listens for requests coming from the client code. It monitors files based on name, not inode. Therefore, renaming a monitored file will cause it to be deleted by the `FAMDeleted` event. In particular, even a change in permissions (`chmod`) appears as a file change. That means FAM tracks some file system attributes in addition to the file itself. Below are some important remarks using FAM [2]:

- Each file registered using `FAMMonitorFile()` throws a `FAMExists` and `FAMEndExist` event. This is useful--when watching directories.
- Formally shutting down the monitor with `FAMCancelMonitor()` sends a `FAMAcknowledge` event.
- FAM doesn't follow symbolic links of watched files, but it catches changes on hard links. For example, removing a hard link to a file yields a file-changed event, because that file's link count has decreased.
- FAM enables monitoring directories and subdirectories as well. Watching a directory means reporting events on its immediate children as well as the directory itself. It's very similar to watching files, except that you call `FAMMonitorDirectory()` instead of `FAMMonitorFile()`.
- Registering a directory with FAM will report several `FAMExists` events: one for the directory itself, plus one for each element contained therein. A single `FAMEndExist` event marks the end of this list. Such information can be useful for statistics, such as tracking the number of a directory's child elements.
- For monitoring text-based log files, the notion of an update means the addition of new entries (lines). A FAM-based application could track such a log file and provide real-time processing of its entries.

3.2.2 Concurrency Control

The lack of concurrency control makes the file system susceptible to cache updating conflicts. For example, assume that file server *FS1* is online and is updating the cache. While doing so, file server *FS2* is re-integrating after a previous disconnection and

intending to synchronize with the cache. There is a possibility that both processes will try to write to the same file space, resulting in a write/write conflict.

Most current approaches to concurrency control in database systems rely on locking of data objects as a control mechanism. In our model, we adopt Optimistic Concurrency Control paradigm [7,10,37] because it allows a transaction to avoid using locks until just before the data is to be *updated* or *deleted*, which helps avoiding unnecessary locks and improves performance [10]. Optimistic concurrency derives its name from the "optimistic" assumption that collisions between transactions will rarely occur. A collision is said to have occurred when another transaction updates or deletes a file between the time it is read by the current transaction and is updated or deleted. This is the opposite of Locking, or "pessimistic" concurrency control, in which the application developer believes that collisions between transactions are commonplace. In optimistic concurrency, a file is left unlocked until just before it is updated or deleted. At that point, the file is reread and checked to see if it has changed since it was last read. To determine whether or not the data has been changed, its current version is compared to a previously cached version. In our model, this comparison is based on the MD5sum checking between files.

3.2.3 Servers Interaction and Integration Model

Servers' reintegration is a transitory state through which the file servers reconnect back to the file system network and synchronize their data after a disconnection. In this stage and as previously presented, the file servers reach a consistent state by synchronizing all modified data with the centralized cache and resolving all conflicts.

This state is achieved by having two supporting phases, the logging stage and the reintegration stage.

3.2.3.1 The Logging Process

The logging process starts once the file server loses connection with the metadata. To coordinate this action, the metadata periodically exchange heartbeat packets with the file servers to detect disconnections. When this event is triggered, both the cache and the file server will start the logging process. In particular, the cache server will start maintaining a log file, which we refer to as Cache Transaction Log (CTL), *for each* file server that is disconnected from the network and will log all client accesses to the files in the centralized cache. On the other side, there will be other log files in each disconnected server, referred as Server Transaction Log (STL) that will log the actions done by the local users on the disconnected servers. Notice that for logging write operation, the system only considers the final write operation to a file. If there are several write operations upon a file, the system only keeps the final version of after those write operations. This is because the cache keeps a single copy of the file for the efficient usage of the disk space, while duplicates are kept in the backup cache. Therefore, the log entry only keeps the last write operation to the file.

At the end of the disconnection stage, the system would be in an inconsistent state where objects on both sides, the file servers and the centralized cache, are modified and need to be synchronized. Both the STL and the CTL files consist of records that represent the changes to the file system during disconnection. The records track in details: The modified file name, the file size, the file owner and the MD5 signature of the file.

In our design, we minimize the network communication by shipping the STL logs and making the comparison locally on the cache server. This way, not only the communication is minimized, but the burden of replaying STL logs is shifted from the file servers to the centralized cache, which meets our goals of freeing the file servers as much as possible.

3.2.3.2 ***The Reintegration Process***

The reintegration process is started the moment metadata triggers the heartbeat packets back. To demonstrate the reintegration process, consider file servers *FSi and FSj* that are re-synchronizing with the cache server *(CS)*. At first and as illustrated in figure 3.2, each file server will ship its STL file to the cache server, which will replay all the received STL files in parallel. In particular, *CS* will read each transaction for each reintegrating server, and identify which objects have changed during the disconnection stage. Then, *CS* communicates with *FSi* and issue one lock at a time on these objects. The locking step is important in order to prevent the local users on the file server from doing any modifications while the reintegration process. Once locks are successfully placed, the cache server will start the reintegration process, and files will be moved between the file servers and the cache server, according to certain policies. During this stage, conflicts will arise as a reason to these file movements, and therefore, conflict detection and resolution techniques are essentially needed, as we will see in the next section.

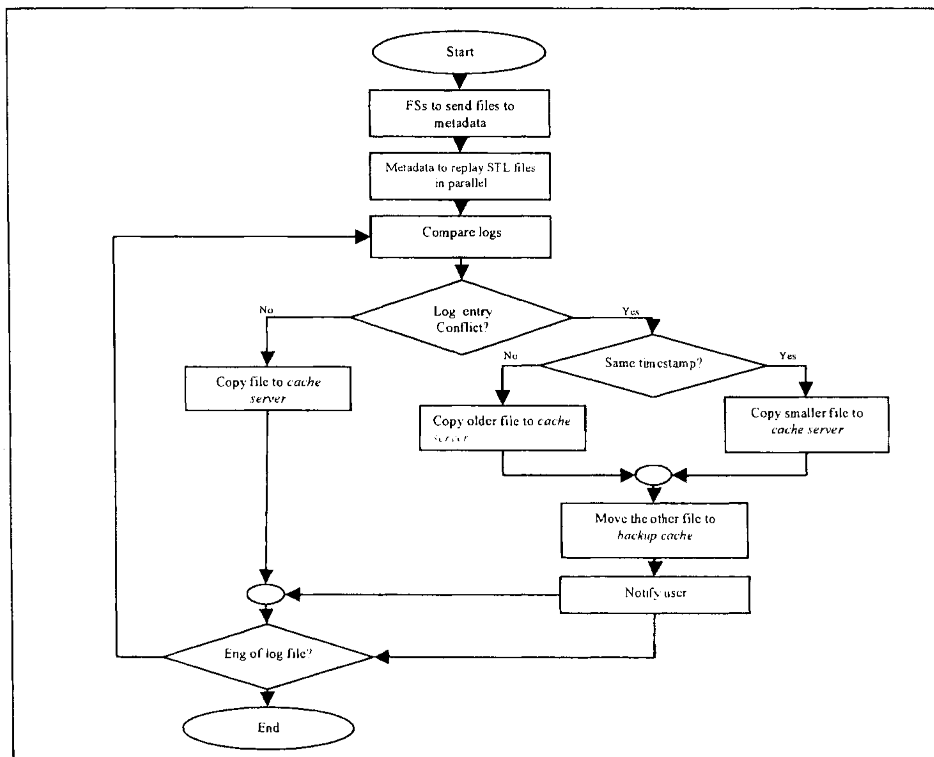

Figure 3.2: STL Integration Algorithm

3.2.4 Conflict Detection and Resolution Strategy

The file system framework that was previously discussed works well in connected environment. However, disconnected operation is a fact of life in network-centric mobile computing environment. Hence, there is a possibility that a file is modified during the disconnection period. In our design, two forms of conflicts are introduced; namely, file conflicts between the caching server and a reintegrating file server, and conflicts between two reintegrating file servers. Now we analyze each of these cases in more detail:

1) *Server-Cache Conflict*: this case describes the situation where there is file conflict between the cache and a previously disconnected file server. To illustrate, consider the case where a file server is disconnected with a file *f* in its repository. At the same time, there is an exact copy of this file, with the same file name in cache, and has

been modified by another connected client. Now suppose that the disconnected file server is back and intends to synchronize with the cache repository. A natural question is: how can the system detect data conflict and decide which file to maintain? We answer this question by first giving the following definition:

Definition1: Let t be the last modified time of a file f that was fetched from a file server FS while caching it in the cache server CS. Therefore, $f \in$ FS and $f \in$ CS before the disconnection stage. At the end of the disconnected period, a conflict is detected if the modified time t of the file f in the server is not equal to t.

Since the last modified time t of an object is maintained, the system can use this information to detect any data conflict. For the case of no conflict, the system basically will do nothing. However, if a conflict is detected, then the timestamp of f will be compared on both sides: the rejoining server and the cache. In case the file's timestamp is newer on cache, then the corresponding file f is written back to the server. That is, the changes of the object in the cache's local disk are propagated back to the reintegrating server, while backing-up the other file in the backup cache. On the other hand, if the file's timestamp is newer on the reintegrating server, then the corresponding file f is written back to the cache, as illustrated in Figure 3.3.

Begin: conflict resolution procedure

For each $f \in FS$

If f.modtime $\in FS \geq f$.modtime $\in CS$ **then** //conflict found

Make a new version of $f.CS$ and copy to backup-cache

Copy f $FS \rightarrow CS$

else-if f.modtime $\in FS \leq f$.modtime $\subset CS$ **then**

//conflict found

Make a new version of $f.FS$ and copy to backup-cache

Copy f $CS \rightarrow FS$

else

Do nothing //no conflicts were found.

End: conflict resolution procedure

Figure 3.3: Conflict-Detection Algorithm

2) *Server-Server Conflict* since reading STL logs are done in parallel, it could happen that the cache server locates two entries in two different STLs with the same file namespace, causing a conflict (e.g., two files are created with the same name space on two disconnected servers). We refer to this type as *server-server* conflict. In this case, the system works out to reach a consistency state and maintain a single copy of the file, and the caches server has to decide which log entry to start with. The decision is made based on the following definition:

Definition2: Let t be the last modified time of object f. $f \in FS1$ and $f \in FS2$ during the reintegration stage. In case of object f exists on both file servers, the cache

server will resolve the conflict by locating the entry that has an older timestamp t. The file with an older timestamp entry will be moved to the backup cache.

The rationale behind this definition is that we assume the newer objects will have more probability that will be accessed in the near future, and therefore they should be available in primary cache (cache temporal locality [11]). However, if by coincidence both have the same timestamp, then the metadata will break this tie by basing its comparison on the sizes of these objects. That is, the smaller size object is moved to the backup cache (see figure 3.4). Our justification to this decision is that once files have the same access probability (*same modification time t*), then it is safe to assume that both are of the same importance, and it does not matter which file to keep in the primary file system, as long as we have the other object saved in the backup cache. Therefore, we choose to save the network traffic by moving the smaller size object. The owner of the smaller object is notified by this operation.

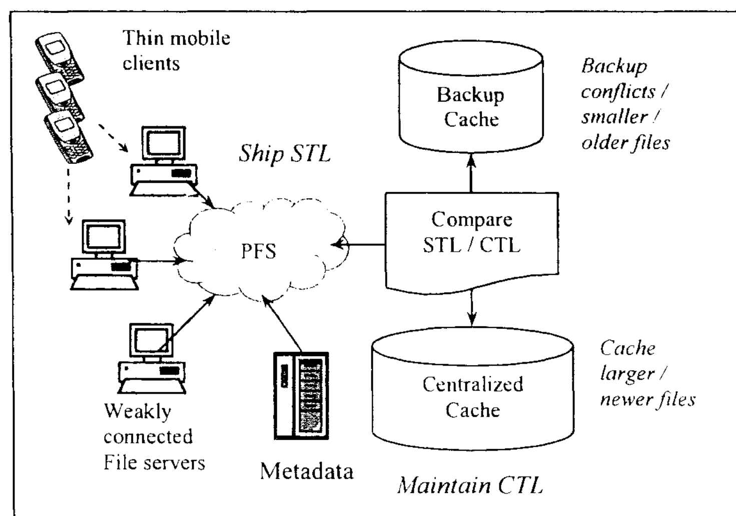

Figure 3.4: STL/CTL Replaying Rules

The design of our conflict resolution is simple when compared to other algorithms, since we do not need to maintain synchronized timestamps or version vectors across multiple servers. Files in the actual file system are not replicated, but are *completely partitioned* among different servers, and maintaining consistency between them and the caching server is accomplished by the algorithm discussed above. On the contrary, file system replication introduces the need for consistency control between participating servers [39]. For example, consider Coda file system where it allows file servers to be replicated in the level of volumes [4]. It uses a variant of Read-One, Write-All (ROWA) protocol. That is, when a client reads a file, it contacts one of the members in its ASVG, and when closing a session on an updated file, the client transfers it in parallel to each member in the AVSG [3].This scheme works fine as long as there are no failures. However, consistency control becomes necessary in the presence of failures. For example, consider a volume that is replicated across three servers *FS1*, *FS2*, and *FS3*. For client *C1*, assume the SVG that it sees (i.e., its ASVG) resides on *FS1* and *FS2*, whereas client *C2* can only access *FS3*. Both *C1* and *C2* will be allowed to open a file *f* for writing, update their respective copies in cache and transfer their copy back to the members in their ASVG. Obviously, there will be different versions of *f* stored in the VSG. The question is how this inconsistency can be detected and resolved? The solution adopted by Coda is that each file server *FS(i)* maintains a Version Vector (CVV) for that VSG in the system. Returning to our three servers example, CVV is initially equal for each server *FS(i)*. When client *C1* reads from one of the servers in its ASVG, say *FS1*, it also receives the value of CVV. After updating *f*, client C1 multicast *f* to each server in its ASVG, that is, *FS1* and *FS2*. Both servers will then record that their respective copy has

been updated, and therefore, update their CVV values (remember that CVV in *FS3* is still not updated because *C1* has no access to it). On the other side, client *C2* can access files in server *FS3*, and subsequently update *f* as well. When *C2* closes its session and transfers the updates to *FS3*, server *FS3* will update its CVV. When the partition is restored, the three servers will reintegrate their copies of *f*. By comparing their version vectors, they will notice that a conflict has occurred that needs to be repaired. In many cases, conflict resolution can be automated in an application-depended way [4]. However, there are also many cases in which users will have to assist in resolving a conflict manually.

Clearly, the network traffic caused by this consistency operation is minimized in our solution. In particular, there is only one single place of versioning (i.e., the cache server), and therefore, only one system request is required to check for the version and update it if necessary.

3.2.5 The Cache Replacement Algorithm

Cache management in flexible file systems deals with the problem of determining a cached file to be replaced when the local cache space is exhausted. Looking at the Cache repository, we may notice that, eventually, the backup cache will get full and hence comes the need for cleaning up old cached files. Through performance comparisons done by Darryl Willick et al [11], it was shown that locality based algorithms such as LRU, which are known to work well as standalone disked workstations are at client workstations in distributed systems, are inappropriate at a file server (temporal locality means that blocks which have been referenced in the recent past will likely be referenced again in the near future). Other frequency based algorithm, such as Least Frequently Used (LFU) requires that a reference count be maintained for each

block in the cache. When a replacement is necessary, LFU chooses the block which has the lowest reference count. LFU is known to work poorly in file servers because certain blocks may build up high reference counts and never be replaced [52].

In our design, we use a Frequency Based Replacement algorithm (FBR) [9]. FBR is a hybrid scheme that combines LRU and LFU algorithms in order to capture their benefits without their associated drawbacks. To accomplish this, FBR divides the cache space into three segments: a new segment, a middle segment, and an old segment. The sizes of these segments are specified by two pre-defined parameters: F-new is basically a percentage that sets the boundary of the total number of cache files which are contained in the new segment (the most recently used (MRU) end), while F-old indicates the percentage of files contained in the old segment (the least recently used (LRU) end). The middle section consists of those files not in either the new or the old section, as illustrated in figure 3.5. When a cache hit occurs to a block in the new section, its reference count is *not* incremented. This is done to resolve the temporal locality which is the primary reason for the past failure of frequency based algorithms. References to the middle and old sections do cause the reference counts to be incremented. As a result, when a file must be chosen for replacement, FBR chooses the file with the lowest reference count, but only among those files that are in the old section of the LRU ordered stack. Cache files that have the same reference counts are resolved by choosing the least recently used of those files. This is done because the files in the new and middle sections have not had enough time to build up their reference counts.

It may be possible that no file accumulates its reference count, keeping all files in the new and middle sections, and leaving the old section empty. In this case, when a file must be chosen for replacement, FBR uses basic LRU to replace files in the new section.

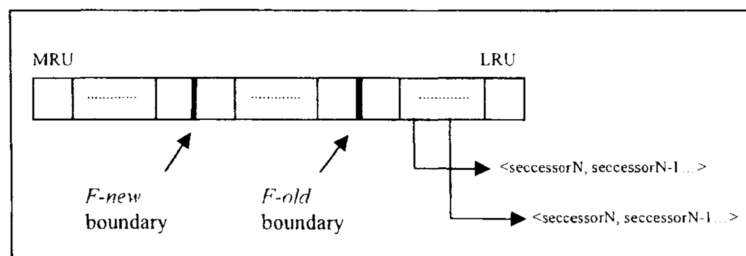

Figure 3.5: FBR Cache Structure

3.2.5.1 The Prefetching Scheme

One of the most difficult problems facing mobile file systems designers is finding the best way to deal with future file accesses and obtain the optimal file-fetch latency time. For that reason, file prediction, which is the process of bringing necessary files into cache *before they are needed*, has been used in many environments including memory designs, databases and web servers [7]. Recently, this concept has been developed in file systems to support distributed and mobile file systems, such as the previously presented Coda [4] and InterMezzo [1]. However, incorrect prediction not only wastes cache space and disk bandwidth, it also increase the time required to bring needed data into the cache if a cache miss occurs while the incorrectly predicted data is being transferred from the disk. Consequently, incorrect predictions can lower the overall performance of the system regardless of the accuracy of correct prediction. Therefore, the success of file prefetching depends on file prediction accuracy [21,22].

The prefetching algorithm of our file system design can be seen in the "file prediction prefetch procedure" in appendix A: Paradise File System (PFS) File System

Pseudo-code. During low system activity, the file system schedules to run the prefetching scheme to predict future file requirements and attempt to make the data available in the cache before it is needed. The file prediction scheme works as follows: the system scans each file in the old section (F-old) of the FBR cache and checks if it has a prediction log (*plog*), otherwise it will create a new empty log for it. *Plog* file maintains a list of files that were accessed immediately after accesses to the corresponding file. The length *N* of *plog* needs to be adjusted so that it is large enough to extend the applicability of the prefetching algorithm and small enough to avoid prefetching unnecessary files. While going through the *plog* of each file, the cache is searched for each successor in the *plog*. If the successor is already cached, the system gets its associated LRU access time changed to the current time. Changing the access time causes the successor to remain in the cache longer, especially if it is still in the new section (*F-New*), preventing it from being flushed before being accessed. However, if the successor file is not in cache, then the system prefetch the file and update its LRU as well.

Chapter 4

File System Request Redirection and Integration into the VFS Layer

4.1 Overview

The Virtual File System (VFS) is an interface that provides a clearly defined link between the operating system kernel and the different File Systems [49]. The VFS supplies the user applications with the system calls for file management (for example, "open", "read", "write" etc.), maintains internal data, handles generic tasks, and does the required locking and communication with user space. In addition, VFS forwards several, but specific, tasks onto the appropriate actual File System. VFS has a unique way of passing tasks to file systems to deal with different physical layouts of data. That is, it uses structures of vector requests in order to communicate with the specific code defined in file systems [51]. As a result, VFS can easily communicate with different file systems, and also, a new file system can be easily added on top of VFS.

In this chapter, we propose different ways of how to seamlessly mask the lookup failures in the main file system. That is, we want to redirect users to the cache server

incase of a cache miss occurs on the main file system. In our design, users should not care whether they are being served from the main file system repository or the main caching server. In order to do so, we need to integrate our distributed file system in the UNIX operating system, more specifically, to interface it with the VFS layer.

We chose Linux OS for implementation, simply because it gives a sensible view of files and directories held on the hard disk of the system, regardless of the file system type or the characteristics of the underlying physical device. Linux transparently supports many different file systems (for example MS-DOS, EXT3 and even CODA) and presents all of the mounted files and file systems as one integrated virtual file system.

Before discussing the integration options to VFS, we demonstrate the mechanism of how VFS communicates with the specific file system in the next section.

4.2 How VFS Works

In general, the communication between the VFS and specific file system does not take place until a file system is mounted to a device, as in figure 4.1. When a device is mounted with ext3 file system, for example, a function is called to read the *superblock*[49]. If it succeeds in reading the *superblock* and is able to mount the file system, it fills in the super_block structure with information that includes a pointer to a structure called *super_operations*, which contains pointers to functions which do operations related to *superblocks*; in this case, pointers to functions specific to ext3. We don't need to worry about how the pointers are filled in, that is VFS's job. But we do have to load the module which includes all the functions for the pointers to pointer at.

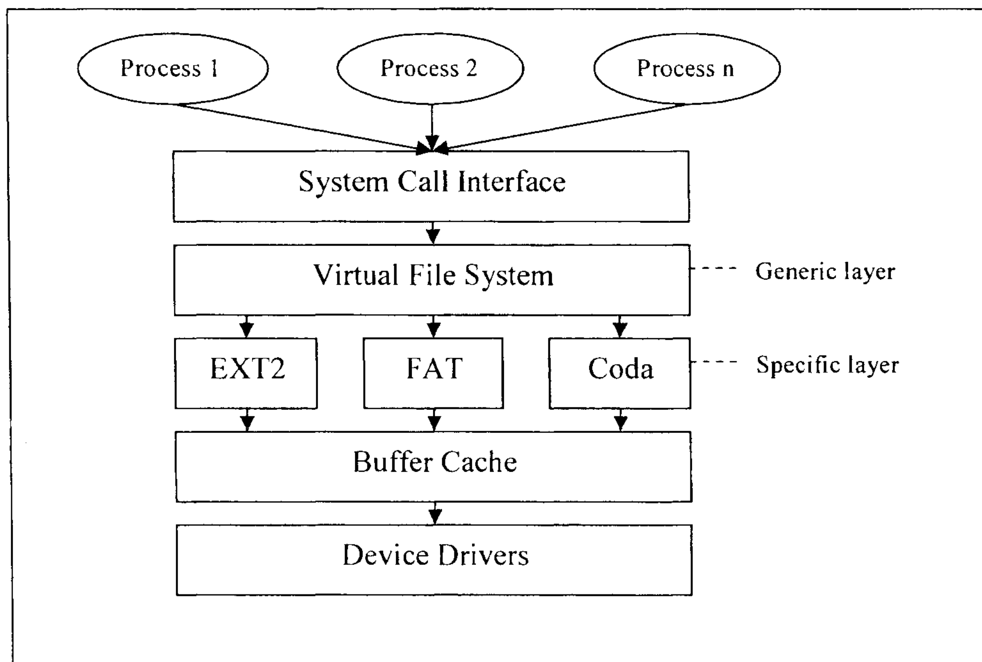

Figure 4.1 File System Integration with VFS Layer

For example, in case of redirecting users to the caching server using the ext2 file system, we need to define all the extra functions in ext2 module. That is, we need to:

- Define our own operations in a file system module.
- Fill in the operation structures like the *file_operations* with the function names.
- Insert the compiled module into kernel.
- Register the module in kernel with the file system name.
- Mount the files system to device for VFS to call.

4.3 Approaches for Redirecting Accessing Path for A File Access

In this section, we show three different approaches on how to seamlessly redirect users to the cache server, in case of a cache miss happens in the main file system. Namely, we propose integrating our file system in the System Call Interface, the VFS

layer and the File System Specific Layer. We present each approach in details, while discussing its strengths and weaknesses.

4.3.1 System Call Interface

As discussed in the previous section, every system-call has a defined number, which is actually used to make the system-call. The system-call number is an index in an array of a kernel structure [48] called *sys_call_table[]*. This structure maps the system-call numbers to the needed service function. For example, opening a file in a user space is represented by the *sys_open* system-call in the kernel, the defined number can be found in the path /usr/include/sys/syscall.h as:

#define SYS_open 5

Therefore, we can intercept system-calls and modify them in order to change the way the system reacts. To do so, we need to write a module with our own function to replace the old one. Then, depending on which system call we want to change, we need to get the *index*(system call defined number) and change the pointer in the index of that *sys_call_table* to our function. For example, if we want to change *sys_open* call, then we would have a code similar to the one in Figure 4.2.

```
Int hacked_open(const char *path)
{
  return 0; /*everything is ok, the new systemcall does nothing*/
}
int init_module(void) /*module setup*/
```

```
{
  orig_open=sys_call_table[5];
  sys_call_table[5]=hacked_open;
   return 0;
}
void cleanup_module(void) /*module shutdown*/
{
  sys_call_table[5]=orig_open; /*set mkdir syscall to the origal one*/
 }
```

Figure 4.2: Changing *sys_open* to Another User-Defined System Call

To redirect the file access for file open, we just need to modify *sys open* as in figure 4.3.

```
call the original sys_open with original path
if sys_open fails, then
        Call original sys_open with  cache server path.
    enf-if
```

Figure 4.3: Redirecting sys_open call to the cache server path

The main issue in this approach is the potential vulnerability in changing the system calls. These changes would affect all the existing file systems in the OS. Changing all file systems means that there is a possibility of rendering them to be incompatible with other "standard" modules in the system. However, changing the

system call interface is considered to be the easiest way to accomplish the redirection task.

4.3.2 The VFS Layer

According to our description in Section 4.2, file system redirection can be accomplished by changing the functions in the VFS layer itself. This way, the file system direction will be controlled by the VFS layer, and not the system calls. The *link_path_walk* is the main VFS function that needs to be changed is. In particular, we can redefine *link_path_walk* function in such a way that if the function returns an error, then the link is directly changed to cache path.

Note that the main issue in this approach is the potential risk in changing the generic layer of VFS. Changing the VFS code would make it even harder to debug because changes are made at kernel level, and hence, even a small bug may crash the whole system. Therefore, as in changing system calls, this approach means having the risk of making the existing file system incompatible with other modules.

4.3.3 Stackable Fan-Out File System

4.3.3.1 The Approach

As we have seen from the earlier options, changing the VFS layer would make it hard to debug and even a small bug would cause the system to crash. Furthermore, modifying existing file systems to reflect the VFS changes is unrealistic.

The most appropriate solution to the redirection problem is found in the Stackable File System [55]. In this approach, a new file system layer is tacked on top of the existing one, ext3 for example, instead of modifying it. This new layer adds the required

functionality in a modular way, and therefore can be developed between the VFS, the stackable file system and the lower file systems.

4.3.3.2 What is a Stackable Fan-Out File System?

Stackable fan-out file systems are file systems that can access more than one lower file system at once. In other words, they helps to ease the development of file systems by providing a higher-level infrastructure for layering new file system functionality on top of existing file systems. Fan-out file system can therefore offer interesting applications such as unification, load-balancing, replication, failover, sand-boxing, and more [55]. This idea is not new, and a lot of attempts have been made. There are some existing experimental file systems using this technology, such as the translucency project [56]. In this file system, access is redirected to files within the directories of the based and another file system.

Figure 4.4 shows the relationship between the VFS, the fan out stackable layer and the lower layer.

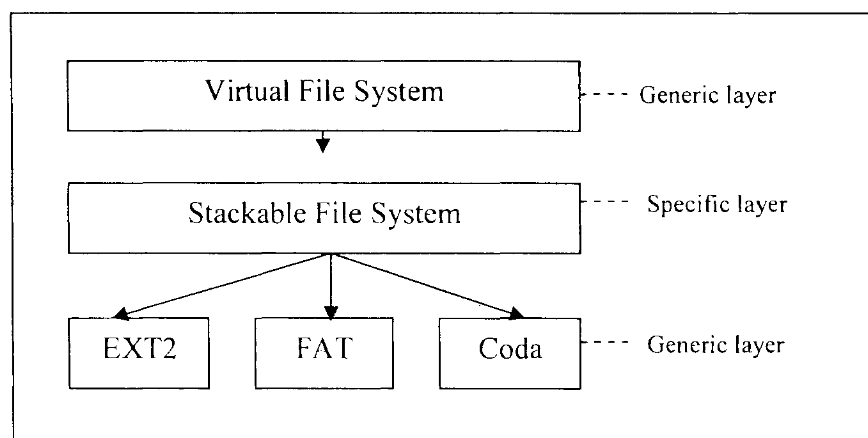

Figure 4.4 Stackable File System Interface with VFS

In figure 4.4, we notice that the stackable layer needs to communicate with different layers: to the lower level as the VFS and to the higher as a traditional file system. When the VFS calls a function of the stackable layer, at first the generic tasks are performed. Next the stackable layer invokes the same function on the layer beneath it.

4.3.3.3 Applying the Stackable File System

The Linux file system consists of a hierarchical tree of directories and files, whereas the head of the tree is consequentially called *root*, which is denoted by "/". In many cases, a user will require an access to a different file system located on a device. This might be a different partition on the same hard drive, a CD in a CD-ROM driver or a network file system. To have access, the operating system needs to be informed about the access operation, and make the file system appears under a directory of the root file system, in other words, the mount point.

Once the file system hierarchy is understood, we can easily figure out the redirection algorithm using the stack layer. In our design, we can safely assume that we have two different file systems, the main file system and the cache system. In particular, the main file system can be of any type, NFS, Coda or even EXT3, while the cache file system can be an NFS mount point. These two file systems need to be mounted separately, but the path under each mount point should be the same. That is, for every file system lookup in the main file system, given the component of the path and the two mount points, we can simultaneously return the *inodes* of these two file systems to the requesting client. Therefore, if there is a failure (e.g., a file system miss) in the lookup of the main file system, the failure is masked and inode of the cache server is returned to the application, instead of the main inode.

Chapter 5

Experimental Results

In this section, we present our experimental results and evaluate the performance of our proposed distributed file system. In our prototype, we used ten-clustered machines. The first eight are running RedHat® Linux which acted as the file system servers, while the ninth machine acted as a Windows XP® client, running Samba (a UNIX tool that allows file sharing between Windows and UNIX). The nodes are connected by a 100Mb/s Ethernet. The file system is also exported to the tenth machine, which had the CRA code running on the file system and acting as the cache server as well.

In order to determine the right cache size, we need to study the frequency and the approximate size of the accessed files. An interesting work has been done by Timothy G. and Ethan M. [40], where they did four months analyzes of file system activity in a university file system. Their study showed that most files are not used very often. Figure 5.1 clearly shows that on a typical day, only 5,000 files (less than 3%) are used on a system with 210,000 files [10].

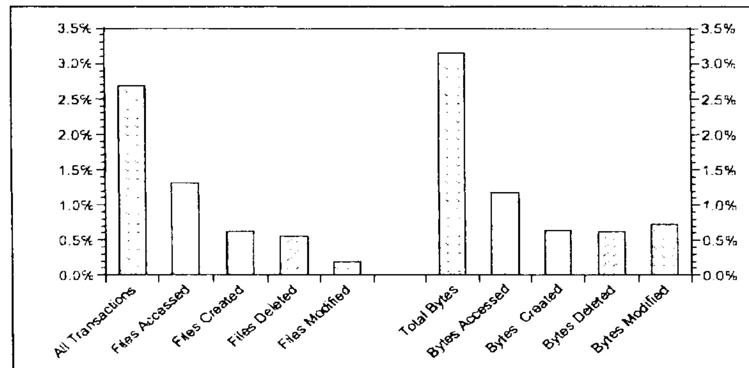

Figure 5.1: Average Daily File System Usage [40]

Based on these findings, we carry a simple analysis to find out the right cache size:

Let us assume that we have a total of 100 file servers, with 50GB of data in each. Then, generally speaking, we would need a cache size of 5,000GB to accommodate the whole space and guarantee 0% cache misses. Of course this impractical, because it is not realistic that all nodes are offline at a certain point of time, and we do not need to cache all servers' information at once. Therefore, we assume in our experiment that at anytime, at most 20% of servers are off the network. This reduces our cache needs to 1,000GB (only these nodes need to be cached). As a result, if we use the above findings for the file access frequency, we may see that we only need 3% (files accessed per day) out of the 1,000GB space, to maintain the information in the cache in a daily basis. Moreover, we may require that we want to cache servers data that has been accessed in the past two weeks, which makes it 42% of information (3% * 14 days).

Therefore, we saw that we may use 400GB of data as a cache space, to handle 5TB of data, in nodes that are available 80% of the time. Keeping the same ratio, the primary file system size in our experiment is 2GB, the primary cache size is 150MB and the backup cache is 100MB. Average file size is 50KB.

First, we ran a number of experiments that compare the caching algorithms (FBR vs. LFU). We simulated 5000 accesses to the file system using different cache sizes. As can be seen in Figure 5.1, our file system performed better using FBR caching algorithm than using LFU. Notably, the two algorithms behaved almost the same in the two extremes (i.e. very low / very large cache sizes). The reason for this behavior is that when cache size is small, CRA will continuously replace files and the effect of caching will not be obvious. Similarly, if cache is too large (almost the same as the file system size) then CRA would be minimal in both LFU and FBR cases. In terms of network traffic, the enhanced caching algorithm resulted in less network utilization (because of fewer file system calls). In our tests, FBR has averaged 13% less network traffic than LFU.

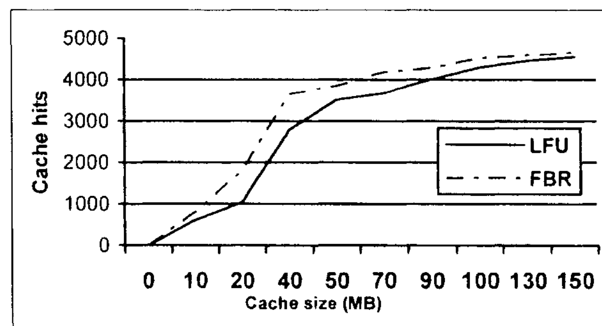

Figure 5.2: CRA Benchmark

We chose Coda file system to base our tests on, basically because Coda provides the functionality of partitioning files over multiple servers, yet providing a singular view of all files to the clients. To get an indication of file system performance compared to the unmodified Coda, we used IOZone tool to measure the file system performance. IOZone measures a variety of operations (read, write, re-read, re-write, read backwards, read strided, fread, fwrite, and random read). We ranged the block sizes between 50K and 1024K. As shown in table 5.1, our file system ran a little more slowly than the

unmodified Coda, due to the overhead of calling and executing the replication module (counted for about 7% overhead).

File system	File Read	File Write	Random read	Random write
Coda (original)	*68.48Mb/s*	*65.52Mb/s*	*66.90Mb/s*	*63.78Mb/s*
Coda (modified)	*64.02Mb/s*	*60.35Mb/s*	*62.34Mb/s*	*59.13Mb/s*

Table 5.1: File System Benchmark

Although the amount of overhead put over the unmodified Coda looks realistic, it is important to note that our current implementation is not integrated in the VFS level, and it is being called as an external application. We expect that this overhead to be reduced and be negligible when we integrate the replication calls within VFS.

Furthermore, we ran various tests to observe the behavior of the file servers during the reintegration process. Typically an STL file with 10,000 records is 50-75KB in size. Figure 5.2 shows the relation between the STL size and the number of files conflicts. As shown in the figure, the larger STL file is, the more likelihood that an entry is repeated in the log, causing more object movements from the cache server to the backup repository. Repeated entries are generally resulted from updating a particular file more than once during a disconnection period. As we may notice in the figure, probability of conflicts tend to increase in a higher rate when STL logs exceed 10,000 entries.

Likewise we expect the relation between the number of STL logs and the likelihood of conflicts to be the same. That is, having more servers to reintegrate (i.e.,

more STL logs to be shipped), would cause more conflicts to occur. Of course, we expect the process to consume more time because the cache will start switching from one STL log to another.

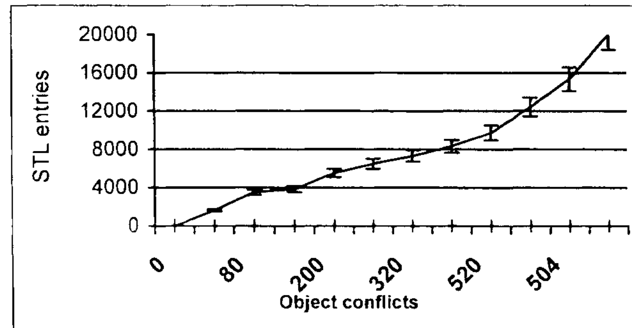

Figure 5.3: STL Size vs. Number of Conflicts

The time of reintegration process is the period of which the metadata allocate the joining server to the time of integrating the last record in the STL and CTL files. Figure 5.3 shows how reintegration time is affected when more STL conflicts are introduced. Clearly, the metadata would need more processing time to resolve file conflicts and move the objects to the backup repository. Our test run results shows that replaying a conflicting entry takes 21% more time than a conflict-free entry.

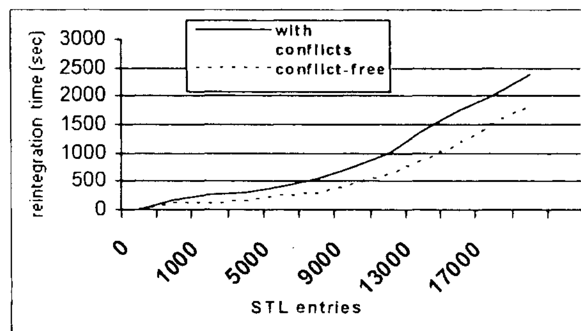

Figure 5.4: Conflict vs. Conflict-Free STL Reintegration Time

In Figure 5.4, we show how reintegration time is affected by the number of reintegrating servers. The bottleneck for the integration process is determined by how

many STL logs the cache server can process. In our experiment, the time taken to reintegrate all eight servers is almost linear, indicating that the cache server did not reach this bottleneck yet.

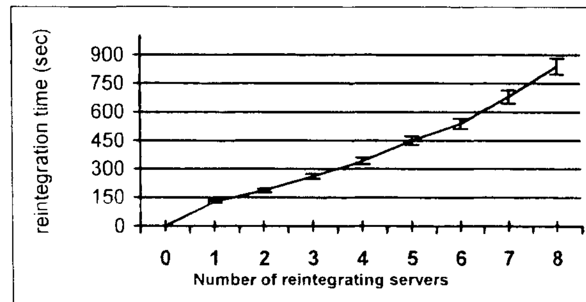

Figure 5.5: Reintegration Time vs. Number of Servers

Furthermore, we performed several tests to measure the performance achieved by our cache-based file prediction. To determine the performance benefits of prefetching and for comparison purposes, we varied the cache size, the number of successors for each file (*N*) for each test. As can be seen in Figure 5.5, our file system performed better using FBR caching with prefetching, than without. Notably, the two algorithms behaved almost the same in the two extremes (i.e. very small / very large cache sizes). The reason for this behavior is that when cache size is small, CRA will continuously replace files and the effect of files prefetching will not be obvious. Similarly, if cache is too large (almost the same as the file system size) then FBR will replace minimal files, and most of the predicted flies will be already available in cache.

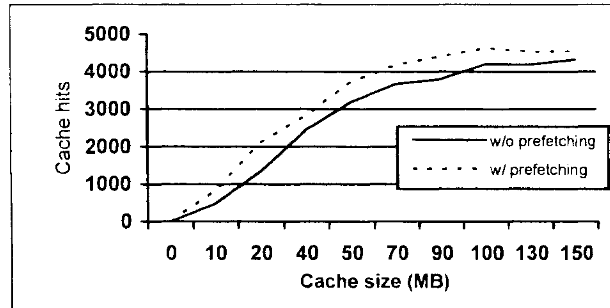

Figure 5.6: Cache size vs. Number of Cache Hits

In our next test, we varied the value of N (the number of successors for each file) and we observed the percentage of correct predictions. As shown in figure 5.6, we could reach 80% correct predictions, using N=6. Although there is a slight improvement in going from N=5 to N=6, this increase is not justified as the increase in network traffic caused by N=6 is relatively high (figure 5.7). Ultimately, we choose N=5 for our model.

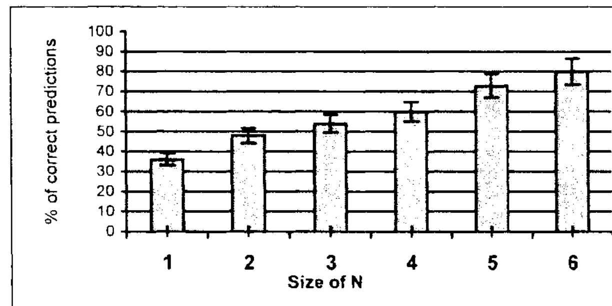

Figure 5.7: Size of N vs. % of Correct Predictions

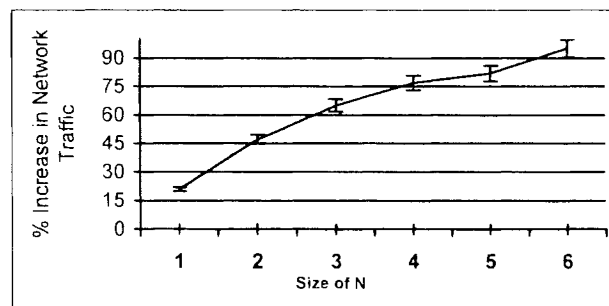

Figure 5.8: Size of N vs. % Increase in Network Traffic

Figure 5.8 demonstrates the relation between the cache size and the percentage of correct predictions. Clearly, the larger the cache size is, the more correct predictions are found in cache. This relationship is then flattened when the cache size is the maximum, because there are less cache misses when the cache size is large, slowing the FBR algorithm in replacing files. Finally, figure 5.9 shows the relation between the size of N and the percentage of correct cache hits. This figure supports our judgment by choosing N=5. Clearly, the best performance is gained when the number of successors is 5 for each cached file. As the number of successors goes beyond 5, the cache will be filled with unneeded data (at least not urgently needed), leaving no space for new data to be stored. Therefore, FBR will continuously do more files replacements, which therefore increases the network traffic on the system (as figure-6 shows with N=6).

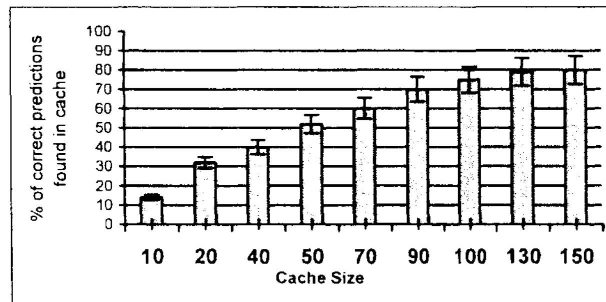

Figure 5.9: Size of N vs. % of Correct Predictions Found in Cache

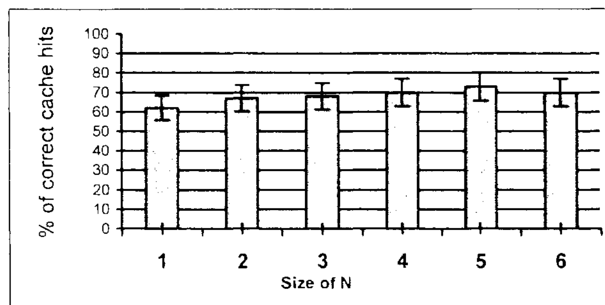

Figure 5.10: Size of N vs. % of Correct Cache Hits

Chapter 6

Conclusion and Future Work

In this thesis, we have proposed highly available file system for mobile clients, as part of the Paradise File System Project (PFS) that is being developed at Paradise Research Laboratory, University of Ottawa. We have presented the main implementation design of our system. Our file system design incorporates novel ideas for providing mobility without sacrificing data availability. Most notable features of our proposed system are its use of the backup cache and the introduction of a versioning mechanism to resolve file conflicts. Moreover, we have showed that the right server-side caching with the right caching algorithm, FBR with file prefetching in our case, may support mobile file systems, reduce file misses, and may outperforms client-side caching in some cases.

The design of our system is relatively independent of any particular file service that it uses, i.e., any file service that uses UNIX file service should be able to use our design to provide better availability. In particular, our results show up to 13% increase in cache hits with up to 80% correct file predictions when file prefetching is used with FBR. Moreover, we presented our servers reintegration design that supports the objectives of building a conflict-free distributed file system for mobile clients; our results clearly

indicates that efficient servers' reintegration is achievable in distributed file systems, bearing in mind the right size of STL logs and the number of reintegrating servers.

6.1 Achievements of This Work

The work described in this thesis presents and discusses the following contributions:

- *To provide a classification of Mobile and Distributed File System designs based upon their characteristics.*
- *A novel distributed file system model for mobile clients*: The file system provides high available and reliable storage for files and guarantees that file operations are executed in spite of concurrency and failures. The design is intended to fit mobile clients (e.g. PDAs and cell phones) that have limited storage space and cannot store all data they need, yet they require to access these data at all times. The model has tackled several design issues, such as resolving conflicts between file servers, caching and file prefetching schemes, assuring concurrency control VFS layer integration. Our test run results indicate that our algorithm exhibits a significant degree of automation and conflict-free mobile file system.
- *An enhanced caching mechanism that is based on the FBR scheme and file* prefetching: As we showed in the performance evaluation section, our caching scheme has outperformed the traditional LFU and LRU caching models, as well as the unmodified FBR version.

6.2 Future Work

We can identify several directions for further research:

- Our performance results so far have indicated that comparative results to the unmodified Coda, with a small compromise due to the overhead of the calling the remote caching procedure. However, with integrating the code into the VFS level, we expect this compromise to be negligible and a file server can be tailored to use our design without users noticing the difference.
- Moreover, even though our file prefetching mechanism runs during low system activities, it is clear that, in general, file prediction and prefetching embarrasses the network, making the model an unattractive option for low latency connections. Therefore, file prefetching algorithm should be extended so that the system becomes aware of the network connection, and adapt to the varying network speeds accordingly.

Appendix

A: Paradise File System (PFS) File System Pseudo-Code

The following is the complete pseudo-code for the PFS File System

METADATA:

```
Begin: nodes checking procedure:
Check for connected node (heartbeats):
      if node is not available, then
            remove from nodes list;
      else-if
            a new node connects, then:
            Register the node's files in metadata;
            Synchronize files with caching-server (call Caching-server
procedure).
      end-if
End: nodes checking procedure.


Begin: File-request procedure
If client X request a file, then:
      Metadata to check the availability of the file in the domain
(consult other nodes in the domain).
```

***If** file is available, **then**:*

Cache the requested file in the caching server (call Caching-server procedure).

Client is permitted to lock and use the file, and unlock it.

else-if

Metadata to check the caching server:

If** file is available, **then

Renew the timestamp in file.

Client is permitted to lock and use the file from cache, and then unlock it.

end-if

else

The Requested file is not available (Cache-miss: error message to client).

end-if

end-if

End: File-request Procedure

CACHE SERVER: using FAM (File Alteration Monitor)

Monitor the file system for any (created, deleted, accessed, modified or executed) files;

If** found, **then

call (caching-server procedure)

end-if

Begin: Caching-server procedure:

For each file accessed in the file system

```
do

    Create the same path of the original accessed file on the Cache
    server;
     Search the caching server for a similar file;
    If found, then
        Check the MD5Sum of the file with the copy in the cache
        If similar, then
            Call CRA with parameter  t;  // to track the access
            time of the file
            delete temp files
            exit 0;           //files are similar, no need to
                              copy again to cache
        else
            Search the Backup Cache file;
            If found, then
                Find the length of the filename;
                Determine the version of the filename;
                //because we are in the Backup cache loop, then
                we know for sure that it has a version.
                Increment the version by one
                Call CRA with parameter-c;  // to make sure
                there's enough storage.
                Copy the original file in Cache to the Backup
                Cache with the new version.

            else,
            //not found in the Backup Cache but available in the
            Cache
                Call CRA with parameter -c;
```

```
                //to make sure there's enough storage.
                Copy the first version to the Backup Cache
                (version .001).
            end-if
        end-if
    end-if
Copy the file from the file system to the Cache Server.
Call CRA with parameter -t;  // to track the access time of the file
Done.
Remove temporarily files;
End: Caching-server procedure:


Begin: CRA procedure (FBR)
Check the parameter of the procedure
If parameter equals to -t then          //keep track of the file
                                        access
    Read file access information into link list in access time order
    Check if the tracked file is in the list
    If found then
    Check if the file is in new section according to F-new parameter
    If file is not in new section then
        Increase one more access to access frequency of the item
    end-if
        Change the access time of the tem to current time
    else
        Create new file access information item for the file
        Set access frequency of the item as 1
        Set access time of the item as current access time
        Add the item to the link
```

```
        end-if
        Write link list back to the file
Else
If parameter equals to -c then    //clean cache according to FBR
                                        algorithm
        Read file access information into link list1 in access time order
        According to the F-old parameter,
        Read files in old section into link list2 in frequency order
        While available space < defined_space and link list2 is not empty
        //defined_space is a parameter
                Get file to be removed from link list2
                Remove the file from file system
                Delete the file from link list1
        end-While
Write back the link list1 into file
        end-if
  end-if
End: CRA procedure


If disk activity is low then
        Call file prediction prefetch procedure
end-if


Begin: file prediction prefetch procedure
For each file f in the old section in cache
        If f.prediction_log doesn't exist then
                Build f.prediction_log: <file successorN, file successorN-
                1, .., file successor0>
```

```
    else // file f already has a prediction_log
        If f.successor_file is not in cache then
            Fetch <file successorN, file successorN-1, .., file
            successor0> to cache
            Update <file successorN, file successorN-1, .., file
            successor0>  access times to current time
        else
            Update <file successorN, file successorN-1, .., file
            successor0>  access times to current time
            //file successors are already cached
        end-if
    end-if
End: file prediction prefetch procedure
```

B: Modified File Alteration Monitor (FAM) Algorithm

The following is the modified algorithm for the File Alteration Monitor (FAM)

```
While a file is added to the monitoring list:
      Get the basename and see if we have an entry for that directory.
      If we don't then
            Make an entry for this directory
            make a file entry for that directory.
                  if we do then
                        check to see if we have file entries.
                  elseif (we're not monitoring the directory, just some
                  files in it)
                  add the file to the filetree for this directory.
                  else (we're already monitoring the dir)
                  throw out the file


While we add a directory:
      check to see if we have an entry for the directory.
      if we don't then
            make an entry for this directory.
            if we do then
                  check to see if we're monitoring files in this
                  directory
                  if we are then (monitoring this whole dir overrides
                  those files)
                        empty the filetree associated with this dir.
                              if we aren't then (we're already
                              monitoring this dir)
                              throw out the dir
```

Case: *When I get a message from the Fam Server:*

while *there are messages pending from the Fam Server,*

get a fam event

figure out the filename that the event is for

if *the event is a changed or created event* ***then***

check to see if it was a file

if *it is a file* ***then:***

show the filename to stdout

if *it is a directory created* ***then*** *(it's a new directory)*

if *I'm in recursive mode* ***then*** *monitor it.*

if *it was a moved or deleted event and I'm in show-deleted mode* ***then***

if *it's a file* ***then*** *show the filename to stderr.*

(revamped) when I get a message from the Fam Server: start timer

while *there are messages pending from the Fam Server and this loop hasn't been going too long,*

get a fam event

figure out the filename the event is for

if *the event is a changed or created event* ***then***

if *it was a file* ***then***

add it to the list of "recently changed files" list

if *it was a new directory and I'm in recursive mode* ***then***

monitor the file

if *the event is a moved or deleted and I'm in show-deleted mode* ***then***

if *it was a file* ***then***

show the filename on stderr

While *I'm done the loop, go back to the select*

Else-while *the "recently changed files" list is checked every second for files that are X seconds old that have been changed.*

C: Glossary of Terms

AFS: Andrew File System

AVSG: Accessible Volume Storage Group

CS : Cache Server

CTL: Cache Transaction Log

CVV : Coda Version Vector

EXT3 : Extension 3 File system

FAM : File Alteration Monitor

FBR: Frequency Based Replacement

FS : File Server

GUID: Globally Unique Identifier

KML: Kernel Modification Log

LFU: Least Recently Used

LRU: Least Frequently Used

NFS: Network File Server

PDA: Personal Digital Assistant

PLOG : Prediction Log

PFS : Paradise File System

RPC: Remote Procedure Call

STL: Server Transaction Log

UFS: Unix File System

VFS: Virtual File System

VSG: Volume Storage Group

WAN: Wide Area Network

References

[1] Peter J. Braam Philip A. Nelson. Removing Bottlenecks in Distributed Filesystems: Coda & InterMezzo as examples. *Proceedings of Linux Expo* 1999, May, 1999.

[2] Peter J. Braam. InterMezzo: File Synchronization with InterSync.1999.

[3] Braam, P. J. The Coda Distributed File System, *Linux Journal*. #50, June 1998.

[4] Kistler, J.J., Satyanarayanan, M., Disconnected Operation in the Coda File System, *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 3-25, Feb. 1992.

[5] Ebling, M.R., Mummert, L.B., Steere, D.C., Overcoming the Network Bottleneck in Mobile Computing, *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994.

[6] "SGI Developer Central OpenSource | FAM", http://oss.sgi.com/projects/fam/

[7] Satyanarayanan, M. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, Vol. 23, No. 5, May 1990.

[8] Kistler, J.J. Disconnected Operation in a Distributed File System. School of Computer Science, Carnegie Mellon University, May 1993.

[9] Silvano M., Cache Management Algorithms for Flexible File Systems, *ACM SIGMETRICS Performance Evaluation Review*, December 1993.

[10] Lu Q., Satyanarayanan, M. Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions. *Proceedings of the Fifth IEEE HotOS Topics Workshop*, May 1995.

[11] Willick, D.L., Eager, D.L. and Bunt, R.B. Disk Cache Replacement Policies for Network Fileservers. Proc. *13th International Conference on Distributed Computing Systems*, pages 2-11, May 1993.

[12] John C.S. Lui, Oldfield K.Y. So, T.S. Tam. NFS/M: An Open Platform Mobile File System. *The 18th International Conference on Distributed Computing Systems (ICDCS'98)*, May, 1998.

[13] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Trans. on Computer Systems*, pages134-154, 1988.

[14] G. A. S. Whittle, J.-F. Pâris, A. Amer, D. D. E. Long and R. Burns. Using Multiple Predictions to Improve the Accuracy of File Access Predictions. *Proc. 20th IEEE Symposium on Mass Storage Systems & Technologies*, pages 230–240, April 2003.

[15] Griffioen, J., Appleton, R. Reducing File System Latency using a Predictive Approach. *Proceedings of USENIX Conference*, 1994.

[16] Kroeger, T. M., and Long, D. D. E. Design and Implementation of a Predictive File Prefetching Algorithm. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 105–118, Jan. 2001.

[17] H. Lei and D. Duchamp. An Analytical Approach to File Prefetching. In *Proceedings of the 1997 USENIX Annual Technical Conference*, Jan. 1997.

[18] E. Shriver, C. Small, and K. Smith. Why Does the System Prefetching Work?. *USENIX* (Monterey, California), pages 71-83, 1999.

[19] C. Bouras, A. Konidaris, D. Kostoulas. A Most Popular Approach of Predictive Prefetching on a WAN to Efficiently Improve WWW Response Times. *The Second International Workshop on Grid and Cooperative Computing*, Shangai, China, pages 344 – 351, December 2003.

[20] Ahmed Amer, Darrell D. E. Long, Jehan-Francois Paris, and Randal C. Burns. File Access Prediction with Adjustable Accuracy. *Proceedings of the International Performance Conference on Computers and Communication (IPCCC 2002)*, April 2002.

[21] Tsozen Yeh, Darrell D. E. Long, and Scott Brandt. Performing File Prediction With a Program-based Successor Model. *In Proceedings of the 9th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '01)*, OH, pages 193–202, August 2001.

[22] Tsozen Yeh, Darrell D. E. Long, and Scott A. Brandt. Using Program and User Information to Improve File Prediction Performance. *In Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS '01)*, pages 111–119, November 2001.

[23] Tsozen Yeh, Darrell D. E. Long, and Scott A. Brandt. Increasing Predictive Accuracy Through Limited Prefetching. In *Proceedings of Communications Networks and Distributed Systems Modeling and Simulation* (CNDS 2002), pages 131–138, January 2001.

[24] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[25] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Extremely Wide-Area Storage System. Appears in U.C. *Berkeley Technical Report* UCB//CSD-00-1102, March 1999.

[26] Dennis Geels. Data Replication in OceanStore. Appears in *U.C. Berkeley Master's Report and Technical Report* UCB//CSD-02-1217, November 2002.

[27] Patrick R. Eaton. Caching the Web with OceanStore. Appears in *U.C. Berkeley Master's Report and Technical Report* UCB/CSD-02-1212, November 2002.

[28] Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz. Maintenance-Free Global Data Storage. Appears in *IEEE Internet Computing*, Vol 5, No 5, pages 40-49, September 2001.

[29] Brent ByungHoon Kang. S2D2: A Framework for Scalable and Secure Optimistic Replication. Appears in UC Berkeley, Dissertation, 2004.

[30] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus Replicated File System. Proc. *1990 Summer USENIX Conf.*, pages 63-71, 1990.

[31] Popek, G.J., Guy, R.G., Page, T.W., Heidemann, J.S. Replication in Ficus Distributed File Systems. In *Proceedings of the IEEE Workshop on Management of Replicated Data*. November, 1990.

[32] P. Honeyman and L.B. Huston. Communications and Consistency in Mobile File Systems. October 1995. *IEEE Personal Communications*, December 1995.

[33] Timothy J. Gibson and Ethan L. Miller. Long Term File Activity Patterns in a Unix Workstations Environment. *Fifteenth IEEE Symposium on Mass Storage Systems*, Greenbelt, MD, pages 355–371, March 1998.

[34] Steven Dropsho. Comparing Caching Techniques for Multitasking Real-Time Systems. *Computer Science Department Technical Report* 97-65 November, 1997.

[35] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp File System. In *13th Symposium on Operating System Principles*, pages 226–238, Pacific Grove, CA, October 1991

[36] Levelt, W.G., Kaashoek, M.F. Bal, H.E., and Tanenbaum, A.S. A Comparison of Two Paradigms for Distributed Shared Memory. *Software Practice & Experience*, vol. 22, pages 985-1010, Nov. 1992.

[37] Lu, Q. and Satyanaranyanan, M. Isolation-Only Transactions for Mobile Computing. *Operating Systems Review*, Volume 28, Number 2, pages 81 to 87, 1994.

[38] M. Satyanarayanan, Fundamental Challenges of Mobile Computing. *ACM Symposium on Principles of Distributed Computing*, 1995.

[39] Satyanarayanan, M., Kistler, J.J. Coda: A Resilient Distributed File System. *IEEE Workshop on Workstation Operating Systems*, Cambridge, MA, USA, Nov 1987.

[40] FreeBSD Handbook, NFS Chapter 24: Network Servers

http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/network-nfs.html

[41] Network File System Version 4

http://advantage.sun.com/protected/solaris10/adoptionkit/general/features/nfs_v4.htm

[42] Networked File System http://www.scit.wlv.ac.uk/~jphb/comms/nfs.html

[43] F. Pister, L. Hess and V. Lindenstruth. Fault Tolerant Grid and Cluster Systems. *Kirchhoff Institute of Physics* (KIP), University Heidelberg, Germany.

[44] Z Chen, G. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault Tolerant High Performance Computing by a Coding Approach. *ACM SIGPLAN*, pages 213 – 223, 2005.

[45] Mummert, L.B., M. R. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, Dec. 1995.

[46] James J. Kistler: Increasing File System Availability through Second-Class Replication. *Workshop on the Management of Replicated* Data, pages 65-69, 1990.

[47] Brian Noble, Mahadev Satyanarayanan. An Empirical Study of a Highly Available File System. *SIGMETRICS,* 138-149, 1994.

[48] The Linux Kernel Programming http://marvin.kset.org/~cardi/filez/linux/chap5.html

[49] Linux File system Drivers http://inglorion.net/documents/tutorials/tutorfs/

[50] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 198–212, Oct 1991.

[51] Butnaru Radu & Muresan Ovidiu. Implementing a Linux File system as a Kernel Module, http://maria.utcluj.ro/~rbb/files/my_work/os/sfsimpl.html

[52] John T. Robinson, Murthy V. Devarakonda: Data Cache Management Using Frequency-Based Replacement. *SIGMETRICS*, pages134-142, 1990.

[53] Gray, J., Helland, P., O'Neil, P., Shasha, D. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, Montreal, Canada, pages 173-182, 1996.

[54] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a Serverless Distributed File System.. Proc. *ACM SIGMETRICS*, pages 34-43, Jun 2000.

[55] Stackable Fan-Out File Systems: http://filesystems.org/project-fanout.html

[56] Operating System Support for Fan-Out File System: http://filesystems.org/docs/fanoutfs-sosp-poster/

[57] Mogile Distributed File System: http://www.danga.com/mogilefs/

[58] RedHat Global File System: http://www.redhat.com/en_us/USA/home/solutions/gfs/