Translucent Cache Management for Mobile Computing

Maria R. Ebling
March 1998
CMU-CS-98-116

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:
Mahadev Satyanarayanan, Chair
Bonnie E. John
James H. Morris
Douglas B. Terry, Xerox PARC

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA), Air Force Materiel Command, USAF under agreement numbers F19628-96-C-0061 and F19628-93-C-0193, the Xerox Corporation, and the Intel Corporation. Some of the equipment used in the process of conducting this research was acquired through the National Science Foundation Equipment Grant #9022511.

Keywords: Disconnected operation, distributed file systems, high availability, mobile computing, caching, transparency, optimistic replication, hoarding, user interface, Coda.

To Chris

Abstract

Recently, distributed files systems have aggressively exploited caching to provide high availability to their clients. Such systems allow users to access and modify cached data even when clients become disconnected (e.g. a laptop removed from its docking station) or weakly connected (e.g. a laptop connected via a modem) to servers. In doing so, these file systems violate one of the key properties of caching: transparency.

When disconnected or weakly connected, these systems cannot always maintain the illusion of connectivity. For instance, the system may not be able to service a cache miss or may only be able to do so with a lengthy delay. When the illusion of connectivity breaks down, the usability of the system is undermined. Early designers of these systems either did not recognize or failed to appreciate the extent to which users would be affected. After extensive experience with the Coda file system, however, we have witnessed the arduous learning curve experienced by new Coda users. We have also observed occasional, but serious, confusion caused by modal behaviors introduced to cope with the range of network connectivity encountered by mobile clients.

This dissertation describes the design, implementation, and evaluation of the CodaConsole, a graphical interface that makes caching *translucent* to users of such systems. To make caching translucent, the interface exposes critical aspects of caching to support availability, while hiding noncritical details to preserve usability. For example, the system informs users about the availability of their tasks, but does not bother them with minutia of cache consistency. The interface also offers opportunities for users to control where network resources are spent.

To evaluate the CodaConsole, I performed a usability test. This study showed that users were able to learn how to use the interface in about one hour. They were able to operate the interface with little difficulty. They were also able to understand problems indicated by the interface and then determine what actions might resolve those problems. Perhaps the most surprising result, however, was the fact that novice Coda users performed almost as well as expert Coda users. This interface successfully makes caching, in the presence of disconnected or weakly connected operation, translucent.

Acknowledgments

Graduate school has been a long and arduous journey. It has been a time of learning and of personal growth. People make Carnegie Mellon a special place and I would like to thank those who helped me during my journey here. In writing these acknowledgements, I have undoubtedly forgotten people who deserve to be mentioned. If I forgot to mention you by name, I apologize!

I would like to begin by thanking my advisor, Satya. He's a brilliant researcher who has provided me with valuable technical guidance and made time for me in his busy schedule. More than anything else, Satya taught me to ask questions and to let those questions be my guide. Other members of my committee also have my gratitude. Jim Morris made me consider users of non-Unix systems; he also had many insightful comments about the bigger picture. Bonnie John opened the doors to HCI and provided support throughout. Doug Terry, my host when I was a summer intern, introduced me to industrial research. In the process, he provided a light at the end of a very long, very dark tunnel.

The past and present members of the Coda group—Peter Braam, Jay Kistler, Puneet Kumar, Yui-Wah Lee, Qi Lu, Lily Mummert, Brian Noble, Henry Pierce, Josh Raiff, and David Steere—deserve special thanks. Without many of their contributions, my thesis would never have been. The Coda users throughout the years also deserve special recognition for the frustration inherent in depending on a research system. I would like to personally thank Stuart Clamen, David Eckhardt, Juan Leon, Henry Rowley, and Amy Moormann Zaremski for the comments and insights they gave me about the problems new and experienced users of Coda faced.

My fellow graduate students lent a unique kind of support during their own journeys. I extend my thanks to Vince Cate, Herb Derby, David Maltz, Francesmary Modugno, Lily, Brian, Sue Older, John Pane, Sasha Wood, and especially to Amy, Jennifer Kay, and Matt Zekauskas. I greatly appreciate the neighborly advice, friendship, and encouragement you all provided. Friends beyond the confines of Wean Hall—including Rea Freeland, Roseanne Hickey, Mark Zaremski, Carrie O'Donnell, David Groat, and Mike and Marcy Langer—reminded me about life on the outside.

A very special thanks goes to Amy, Jennie, Rea, David, and Chris Okasaki for reading earlier versions of my thesis. They all provided me with valuable comments and greatly improved the final document. Sharon Burks deserves more thanks than anyone could ever give her. She cuts through red tape and provides invaluable advice.

I would like to thank the administrators of Carnegie Mellon's User Studies Lab for the use of their facilities, particularly Sandy Esch and Tom Pope. Sandy deserves special recognition for her commitment to the lab. Without her expertise and practical advice, performing a user study would be a much more difficult process. The participants in the user study, whom I cannot thank by name but who know who they are, also deserve special recognition. Each of them volunteered the better part of a day to help me with my research. I greatly appreciate their generosity.

A number of programs also deserve special recognition. These include tck/tk and the tix widget library, Kurzweil Voice, and Microsoft Word. The graphical interface that I built as part of my thesis research was implemented with tcl/tk using the tix widget library. Though I believe that my interface gradually outgrew these programs, I found them to be incredibly useful tools. I would like to thank Brent Welch for his advice in using canvases, back before we had his wonderful reference book. I would also like to thank Ioi Lam for building the tix widget library. This thesis was written with the assistance of the Kurzweil Voice program. Voice recognition has finally matured into toddlerhood. At times, it amazes you with the wonder of the things it can do; yet, at other times, it frustrates you with its stubborn defiance of your authority. As a whole, Voice has given me more freedom than I have had in years and it has enabled me to work much longer hours at the computer. For that, I am extremely grateful, but I eagerly await the day when voice recognition systems mature beyond the "terrible twos." Microsoft Word, on the other hand, deserves a different kind of recognition. Its ability to corrupt documents never failed to amaze me. In fact, this *commercial product* holds the dubious distinction of corrupting more of my data in six months' time than the *experimental* Coda file system corrupted in approximately seven years. Some day, I hope that its designers have the joy of writing a large document using their own product. May they live interesting lives.

My parents taught me to persevere in the face of obstacles, but they also taught me a love of knowledge. My mother typed in the whole of Appendix C and parts of Appendix B on her "vacation" and also proofread a draft of the thesis in her "spare" time. Without the influence of my father, I would never have come as far as I have. He kept me in the mathematics pipeline even as most of my friends dropped out, and I thank him. They provided me with a strong foundation on which to build, but let me find my own path. I hope I can do as well for my son!

Colin taught me to leave school at school and to enjoy the simple things in life. Motherhood has been a new challenge. It is infinitely rewarding, but also requires infinite amounts of patience, especially now that we have hit the two-somethings. You are a wonderful little boy and I look forward to seeing the man that you will one day become!

I do not quite know how to thank Chris, my husband and best friend. He stood with me when I hit the lowest of the lows, and suggested that I wait one month before quiting (a trick we learned in our childbirth class with "month" substituted for "hour" and "quit" substituted for "painkillers"). Without his support and encouragement these past few years, I can honestly say I would not have completed this degree. Thank you!

Maria Ebling
Pittsburgh, Pennsylvania

Table of Contents

List of Figures

1 Introduction

Programmers have traditionally exploited caching to improve performance. The power of this technique has been its ability to substantially improve performance in a way that is completely transparent to higher levels of the system. Coda was the first system to exploit caching not only to improve performance but also to improve *availability*. The term availability refers to providing continuous access to data.

Exploiting caching for this purpose fundamentally violates transparency. For example, the system can no longer guarantee its ability to service a cache miss in a reasonable amount of time. Instead, the system must either halt until such time as the miss can be serviced, or return an error indicating that the miss cannot currently be serviced. Neither of these options maintains transparency to higher levels of the system or to the user, and both options reduce the usability of the resulting system.

The goal of this thesis is to explore ways in which caching can be made *translucent* to higher levels of the system by exposing critical aspects of caching to support availability, while hiding noncritical aspects of caching to preserve usability. For example, when the cost of using the network is high, the system might expose cache misses to the user, allowing users control over network usage. In addition, if the system had high-level knowledge of the tasks that users must perform, it could inform them of the availability of those tasks.

Because Coda is a Unix-based file system, the strategies I use must be compatible with Unix idiom and usage custom. One key feature of Unix is the ability for programs to run without user attention [19, 42]. Balancing the desire to retain this feature against the desire to provide better support for attentive users proved to be a major challenge for my research, with sometimes far-reaching consequences. For example, the use of pop-up dialog boxes that block waiting for the user's answer was simply unacceptable. Such out-of-band communication assumes that a human is paying attention to the system and precludes programs that run unattended. My philosophy throughout this research was to maintain the Unix heritage to the greatest extent possible, while adding enough out-of-band communication to improve the usability of the system. When the user is available, the functionality and/or performance of the system is improved. When the user is unavailable or when a program is running unattended, the system must preserve transparency by acting on the best available information.

1.1 The Thesis

Previous research on systems that cache for availability focused almost exclusively on proofs of concept. The outstanding questions were whether it *could* be done. Users of these systems were primarily the developers plus a few of their friends. These users proved not only that availability could be improved, but also that this class of users could make effective use of it. Now, our focus must shift to making these systems accessible to less motivated and less experienced users.

The lack of transparency intrinsic to systems that cache for availability reduces usability in ways that were either unforeseen or underestimated by early designers. To improve the usability of future systems that exploit caching for availability, we must make caching translucent to users. This observation leads directly to the thesis statement:

> Mechanisms that selectively expose internal details of the cache state and operating environment can improve the usability of systems that exploit caching for availability. Further, it is possible to measure the effectiveness of these mechanisms.

Because systems that transparently exploit caching for availability are rare, I examine these mechanisms within the context of the Coda file system. The ideas behind these mechanisms, however, are applicable beyond Coda. As caching for availability becomes a well-understood tool, it is likely to be applied more widely. The Web, whose current consistency model can, at best, be described as primitive and whose current performance is frequently abysmal, may well endeavor to provide high availability in the years to come. Personal Digital Assistants may also strive to provide high availability, as users become less willing to manually manage the increasingly large "cache" of data stored on these systems. This thesis provides a critical opportunity to learn about mechanisms for improving the usability of caching for availability, allowing us to pave the road for future application.

1.2 Background and Historical Perspective

The full effect of caching for availability on the user was not well understood in early versions of Coda. Once Coda had matured sufficiently and had experienced substantial use by many different users with a variety of backgrounds, we realized that we were seeing usability problems with the various tools *as well as* with the violations of the transparency. In this section, I briefly describe the history of Coda, focusing on the influence each advance had on the usability of the system.

In 1991, Kistler introduced *disconnected operation* in the Coda file system [22, 23]. Disconnected operation provided autonomy to clients of a distributed file system by allowing them to continue using cached data during temporary network or server failures. In summary, Coda exploited its file cache to provide high availability as well as to improve performance. The usability of the system seemed greatly increased because of the new functionality.

Kistler observed that the voluntary disconnection of a laptop computer from a network was equivalent to a total network failure between client and server; thus, disconnected operation also supported portable clients of a distributed file system. To make this mode of operation useful (particularly for disconnection lengths typical of portable clients), the system allowed users to provide hints to the cache manager about important data. These hints helped the system manage the cache in preparation for potential disconnection, helping to ensure that the most useful data is available to the user.

Although disconnected operation provides important functionality, it is not a panacea. Disconnected clients suffer from several serious limitations, each of which reduces the usability of the resulting system:

- Cache misses may impede progress
- Local updates are not visible remotely, and vice versa
- Updates are vulnerable to theft, loss, and damage of the portable computer
- Update conflicts are more likely
- Exhaustion of cache space is a concern

Further, users of the system must provide extensive hints to make effective use of disconnected operation. The system requires that these hints be specified at the level of files and directories. Users quickly learn that application programs access numerous files—files these users never realized existed. New users of the system encounter a steep learning curve before they are able to make effective use of it.

To address a number of these limitations, Mummert extended Coda to exploit *weak* networks [33, 34]. A weak network is one that exhibits high latency, low bandwidth, intermittence, or high cost. When operating weakly connected, Coda services cache misses either in the foreground or in the background depending on the importance of the missing file and the expected service time. It trickles updates to the servers in the background, making updates visible to other clients and lessening both the likelihood of future update conflicts and the cache space required on the client. It also maintains cache coherence by updating the client's cache as other users change files on the servers.

1.3 Lack of Transparency

Weak connectivity ameliorates many of the limitations experienced by disconnected clients, but, like disconnected operation, it is not a panacea. It is fundamentally limited by the available network bandwidth. Further, it cannot obviate the need for disconnected operation because weakly connected clients may well find themselves disconnected—either involuntarily during a network outage, or voluntarily because of concerns regarding cost, privacy, or the perception that the network is temporarily unusable. It does not address the need for detailed hoard hints and it brings to light several previously unrecognized problems, such as excessive delays, the need for both disconnected and weakly connected operation, and inadequate feedback.

Excessive Delays. Servicing a cache miss of even a moderately large file over a slow network can cause excessive delays. Consider the canonical example of a user accessing a 1 MB file while weakly connected. At 9600 b/s, such a demand fetch would require the user to wait almost fifteen minutes! At 28.8 Kb/s, it would still take just under five minutes. Neither of these delays is tolerable for interactive systems [35]. Although modem speeds are increasing, color picture and movie "illustrations" in documents can increase file sizes substantially beyond the canonical 1 MB. And if the weak network connection is intermittent, the expected delays rise proportionally. The system is fundamentally limited to the available network bandwidth.

Need for Disconnected Operation. Although portable clients may have the ability to operate weakly connected at all times, there remain situations in which the user may choose to operate disconnected. For example, the user may wish to extend battery life by not squandering this limited resource on network transmissions; the user may wish to reduce network charges when costs are high; or, the user may want to avoid network transmissions when the network quality degrades substantially. Thus, even weakly connected clients benefit from the ability to operate totally disconnected from the network.

Inadequate Feedback. Disconnected and weakly connected operation introduce modal behaviors[1], which have long been known to cause usability problems [53]. These modal behaviors are critical to supporting programs that operate unattended. Unfortunately, they introduce a mismatch between the user's expectations and the reality of the current operating conditions. This mismatch contributes substantially to the usability problems we have identified in Coda. For example, until recently, our local network was sufficiently overloaded that network bandwidth regularly fell below the threshold for strong connectivity, causing Coda clients to behave as if they were weakly connected. In this mode of operation, the system delays propagation of updates for a brief period of time to allow future updates to make propagation unnecessary. This delay has confused a number of users because such users were not expecting weakly connected behavior while connected via the building's Ethernet. The standard technique for helping users cope with modal behaviors (when they cannot be avoided) is to introduce feedback to help users recognize the current mode of operation [32].

Disconnected and weakly connected operation increase the range of behaviors exhibited by Coda. These strange and seemingly "abnormal" behaviors occasionally cause substantial confusion and frustration for even experienced users. However, the

---

[1] A system is said to exhibit *modal behaviors* when the same action (e.g., keystroke or mouse click) can cause different things to happen under different circumstances. For example, when using vi, typing the letter 'i' does one of (at least) two things. If the program is already in "insert mode", then the letter 'i' is inserted into the text of the document. If not, then the program is put into "insert mode" and the letter does not appear in the document.

transparent transitions that cause this confusion among our users are *critical* to supporting programs that have not been extended to adapt to changing conditions. One view of my thesis then is to explain the system to users and to allow them control over its behaviors, while not sacrificing the transparency that is so important to programs.

1.4 Perspective of Users

For the benefit of those readers unfamiliar with the lineage of Coda and for whom my brief description of the problem was perhaps insufficient, let me describe the use of Coda and its predecessors from the user's perspective. To make this description concrete, let me tell a story about two fictitious employees, Peer and Yoni, and their equally fictitious boss, Alex. The topic of the story is their love-hate relationship with a distributed file system.

> Peer, Yoni, and Alex all have desktop machines. Their company has decided to move to using a distributed file system to ease the maintenance problems faced by their facilities staff. Their data is now stored on central file servers and they access it via their local area network. The file servers are maintained and monitored by the company's facilities staff. Their desktop machines interact with these server machines to give them access to their data, caching the data locally to provide improved performance.
>
> Once the initial problems are worked out, the entire company becomes enamored with the new file system. Users like it because they can easily share data with co-workers. In fact, novice users, unwilling to deal with ftp, no longer use sneaker-net[2] to share their data. They simply send a pathname that is accessible from any company computer. The facilities staff likes the system because it greatly reduces the effort necessary to maintain the company computers. They are no longer required to reinstall new versions of software on every single machine. They simply install the new version in the distributed system and let the automatic scripts deposit it on each client machine. Backups have also become much simpler since they need only deal with a small number of server machines and need not backup the client machines at all.
>
> Soon, however, Peer and Yoni learn a few painful lessons about the downside of distributed systems. The first lesson occurred when their network went down the night before a critical deadline. The second occurred when their file server crashed in the middle of an important presentation. The cause was the same in both situations: even though the computer sitting right in front of them was fully operational and had the data cached, they could not access that data. You see, to preserve consistency in the distributed file system, the local client must maintain an active connection to the file server. During periods of server or network failures (when an active connection is not possible), the client cannot allow access to data even if that data is in its local cache because it cannot know whether or not that data is valid. Shortly after those two disasters, Peer and Yoni convince their boss, Alex, to purchase the high availability add-on that will allow them to continue to access their cached data even during periods of server and network failures.
>
> With the new extension, they find that they can access their data even when the server crashes or the network goes down. They are, once again, quite pleased with their system. In fact, it is working so well that Alex decides to invest in laptop computers,

[2] Sneaker-net is defined as the network achieved by writing data to a floppy or tape on one machine, walking it over to another machine, and reading the data on that second machine.

giving them access to their data during their frequent business trips. They soon learn that operating *disconnected* has advantages, but that it is not without pitfalls. Peer's first trip with the laptop was a near disaster. He knew that he needed to provide hints to the cache manager so that it would know what data to keep cached and he carefully did so, but he forgot to authenticate before disconnecting so he did not have permission to write to any of his data during his trip. He was able to read the data, however, so he copied it onto the local disk so that he could work.

Yoni's first trip with his laptop was equally disastrous. He forgot to tell the system about a file critical to starting up his window manager. Without his window manager, he could not present his slides on-line. Luckily, because he was not completely comfortable with the new technology yet, he had printed backup slides and was able to use them for the presentation. After hearing about their experiences, Alex was very careful to test his hints before leaving to make sure that everything worked as expected. His presentation went smoothly and he was very pleased, until he tried to implement some of the suggestions made during his presentation that happened to affect files he had not anticipated needing. He found that the system provided no support for fetching missing files over the internal modem; the process was, thus, tedious and time consuming. He made a note to look into the newly released add-on supporting *weak connectivity*.

Upon returning to the office, he asked Peer to evaluate the weak connectivity support and get back to him. Shortly thereafter, Peer recommended that they purchase it. Alex began using weak connectivity immediately and found it extremely useful. On his next trip, he dialed-in from his hotel room to update the servers with his changes and download any files that Peer and Yoni had modified. Once again, all three of them were in love with the new system and found it greatly increased their ability to share data on the road. They experienced only a few minor annoyances, in that they were never quite sure when the updates had finished transmitting, but nothing major.

Shortly thereafter, Yoni's wife, Leah, accepted a job offer in a different city. Rather than lose such an experienced and valued employee, Alex decided that it would be more productive to allow Yoni to work from a rented office in the new location, accessing his data via an Internet Service Provider (ISP) and taking full advantage of weak connectivity. Yoni typically remained weakly connected to the home office via the Internet. He found that, although performance was usually acceptable, he occasionally had to disconnect because network characteristics degraded beyond usability.

Once Yoni started interacting closely with Peer on a new project, however, he became frustrated with the system. He found that, because the network performance between his ISP and the home company was variable, he was never quite sure whether he was operating weakly connected or strongly connected. He was frustrated because sometimes his updates did not appear on the servers when he thought they should have. In fact, one day, he and Peer wasted a large of amount of time tracking down what appeared to be bugs, but in fact, turned out to be version skew resulting from these consistency problems. They finally remembered that, when the system is operating weakly connected, updates are not propagated to the servers immediately; instead, the system ages them for a few minutes and then ships them over. They have since learned to check the state of their connectivity whenever such confusion arises. After hearing about these problems, Alex suggested that Yoni look into a recently released interface add-on that addresses these problems.

This story summarizes the development of AFS and Coda. It gives a broad overview of how users might use systems like AFS and Coda as well as the problems those users might encounter. The story ends where my thesis begins, with the promise of an interface to help users of a highly available distributed file system that supports weak connectivity.

1.5 Roadmap of Dissertation

The Coda file system is the research vehicle used to test and to validate the ideas in this thesis. Chapter 2 provides background on the current Coda system. It focuses on disconnected and weakly connected operation, discussing those aspects of Coda relevant to this thesis.

Chapter 3 discusses the design rationale for the translucent caching interface. It begins with a discussion of what aspects of caching need to be exposed to the user. It then continues with a discussion of the potential burdens that translucent caching places on users. It concludes with design principles for limiting those burdens.

Chapter 4 describes the detailed design of the CodaConsole interface, which was inspired by the dashboard of an automobile and features a set of nine indicator lights. The indicators of the interface differ from those of a dashboard because the interface allows the user to expand the indicators to obtain more detailed information. This chapter presents each indicator, describing what it indicates as well as what additional information is available.

Chapter 5 describes the implementation of the interface. It presents an overview of the system architecture. It continues with a discussion of the details of the implementation, including a summary of the Translucent Caching API. It then presents three examples of the use of the Translucent Caching API: an initialization sequence, a simple event notification, and a complex series of event notifications. It concludes with a brief reflection on the implementation.

Chapter 6 examines three techniques used by the interface to reduce the burden translucent caching places on users. The first technique allows users to specify the name of a program and then automatically tracks the references made by that program to identify the files on which that program depends. The second allows the system to offer the user advice regarding her hint specifications. The third allows the system to model the user's patience for waiting for network activity.

Chapter 7 presents the results of a usability test evaluating the interface. The usability test has two primary goals: first, to determine how well users, particularly novice Coda users, are able to learn, operate, and understand the interface; and second, to uncover usability problems in the interface, primarily as a way to focus future work on improving the interface.

Chapter 8 describes work related to this thesis, including work on automatically predicting the files and directories users will use during a disconnected session and work related to exposing the internal workings of other systems to their users.

Chapter 9 concludes the thesis with a discussion of the contributions it makes and suggestions for future research directions.

2 Context and Background

This work was performed within the context of the Coda file system, a distributed file system, whose primary goal is to provide continued access to data during network partitions or server failures. Secondary goals, largely inherited from its predecessor—the Andrew file system (AFS)—include scalability, security, performance, and ease of system administration.

The designers of AFS used a client/server model, partitioning the system into two mutually exclusive sets of machines. The first set, called *servers*, consists of a relatively small number of trusted machines administered by a central authority and dedicated to servicing requests from the second set. The second set, called *clients*, consists of a much larger group of untrusted machines administered and operated by independent groups or individual users. Client and server machines communicate over a local area network via a remote procedure call (RPC) mechanism [5].

Both AFS and Coda make similar usage assumptions. The workload is assumed to be that of an office or engineering environment, including applications such as electronic mail, word processing, program development, and data analysis. Specifically excluded are applications requiring highly concurrent, fine-grained access to data, such as database applications. Further, each client is assumed to be a powerful computer with a moderate amount of disk space. AFS also assumes that each client communicates with servers over a high-bandwidth, low-latency network connection. Coda, however, assumes only that clients will have occasional access to a high-bandwidth, low-latency network connection. In addition, Coda assumes that a single user controls each client.

In the next section, I discuss how these design goals and usage assumptions influenced the basic design of AFS, and therefore, of Coda. The following section continues from the point where Coda diverges from AFS, with an overview of Coda's mechanisms for providing high availability. Finally, I summarize Coda's user interfaces. The discussions in these sections are limited to only those details necessary to understand the work described in this thesis. For further details, please refer to previous AFS and Coda publications [13, 46].

2.1 Influence of AFS

The dominant factor in the design of AFS was the desire to support up to approximately 10,000 clients. In the early 1980s, this was a very ambitious goal—at least an order of magnitude beyond the largest existing distributed file system. Secondary factors in the design of AFS include security, performance, and ease of administration. In the sections that follow, I examine how each of these goals influenced the design of AFS, and therefore that of Coda. Given the ambitious goal of creating a distributed file system expected to service an entire university, it is not surprising that scalability became the dominant concern in its design. As I review the design, issues of scalability will arise time and again.

2.1.1 Client/Server Model

The client/server model addresses two important issues: system administration and security. First, the relatively small number of server machines makes the scale of the system appear, from the point of view of system administrators, to be significantly smaller than the total number of machines in the system. Second, the integrity of the system as a whole relies primarily on the integrity of the server machines. These machines are secured behind locked doors, are accessible only to system administrators, run only trusted programs, and are closely monitored. The integrity of client machines, however, is not assumed. If a client is subverted, the potential for damage is localized and cannot extend to arbitrary portions of the namespace. Unauthorized release of data is limited to the data cached on the subverted machine and to data accessible to any authenticated user logged onto that machine. Similarly, unauthorized modification of data is limited to that data accessible to an authenticated user on that machine. Authentication tokens automatically expire 25 hours after they were originally acquired, so the timeframe for unauthorized modification is limited.

2.1.2 Volumes

AFS divides the file system hierarchy into manageable pieces, called *volumes*. A volume is a subtree of the file system containing related files and directories, such as a user's home directory. A volume corresponds roughly to a mountable Unix file system. Like disk partitions, volumes are mounted within the file system hierarchy at what is called a *mount point*. Unlike partitions, a volume mounted by one client is visible to other clients. In this way, volumes are glued together to form the complete file system hierarchy. The client recognizes mount points during name resolution and crosses them transparently.

Each file system object, which can be a file, directory, or symbolic link, is identified by a unique *file identifier* (*fid*), a 96-bit quantity consisting of a 32-bit *volume identifier* and a 64-bit *vnode number*. To locate a file system object, the client begins by looking up its volume identifier in a database to determine the servers for the object. It then contacts those servers to fetch the file.

The volume abstraction is crucial for supporting name and location transparency. Since mount points are stored globally in the file system rather than locally at each client (as in NFS), every user has the same view of the file system hierarchy. Furthermore, since file names never contain location information, system administrators are free to move volumes to different servers or disk partitions to balance the load among the servers. Thus, both users and system administrators benefit from this important abstraction.

2.1.3 Client Caching

Clients cache data from servers in AFS. Thus, when a user accesses an uncached file (i.e., the user encounters a cache miss), the client cache manager, called *Venus*, fetches that file, in its entirety, from the servers. A copy of the file is then stored on the client's local disk. Should the user access the file again in the near future, the request can be serviced from the local disk and the user need not wait for the file to be fetched from the servers again.

To maintain cache coherence, the client obtains a *callback promise* from the server for every object in its cache. If another client updates a file for which a server has promised a callback, the server must issue a *callback break* before completing the update. Thus, a client can confidently allow access to objects for which it holds a callback promise as long as it remains in contact with the server making the promise.

The design of this caching scheme has important implications for the overall goals of AFS. Caching improves the performance of the client since there are substantial amounts of locality of reference in the anticipated workloads. Caching also improves scalability since it reduces the amount of server load imposed by each client, thus allowing more clients to be supported by each server.

2.1.4 User Authentication

Every AFS user has an AFS identity and password. To access data stored in AFS, the user authenticates to an AFS authentication server, providing a username and password [44]. If the user provides the correct password, the server issues *tokens* to Venus on the user's behalf. Venus then stores these tokens, allowing it to establish secure connections to the file servers. Tokens are valid for a period of 25 hours, after which the user must reauthenticate. Users may view their tokens with a special program.

2.2 Extensions for High Availability

AFS had a strong influence in the design of Coda. In fact, the previous section could have been written to describe Coda. Coda now diverges from AFS, however, as it strives to provide high data availability.

High availability becomes the dominant goal. Scalability, though still important, assumes a secondary role to high availability. Certain assumptions also change slightly. Clients are now assumed to have only the *ability* to connect to a high-bandwidth, low-latency network at least occasionally. At other times, clients may be disconnected from the network or connected via a low-bandwidth network. Clients are further assumed to be controlled by a primary user; in fact, certain activities are limited to that user.

Coda provides high availability through the use of three complementary mechanisms. The first mechanism, *server replication*, provides resiliency in the face of server failures and network partitions. The second mechanism, *disconnected operation*, maintains the illusion of connectivity when a client is completely isolated from the servers. The third mechanism, *weakly connected operation*, alleviates some of the shortcomings of disconnected operation when only a low-bandwidth network connection is available.

2.2.1 Server Replication

Coda uses *optimistic replication* at the volume level to provide resiliency to server failures and network partitions [28, 46]. With optimistic replication, updates to file objects are allowed provided that at least one replica is visible to the client performing the update. Thus, conflicting updates (caused by write sharing) can and do occur. Coda detects such updates, automatically resolves most types of conflicts, and provides tools to help users manually resolve the remaining ones. Fortunately, in an office and engineering environment, write sharing is a relatively infrequent occurrence. This thesis depends minimally on Coda's replication mechanism so I will not delve into further details on this topic; instead, interested readers can refer to Kumar's thesis [28].

2.2.2 Disconnected Operation

Disconnected operation provides resiliency to failures that completely isolate a client from all Coda servers. Because of this resiliency, Coda supports mobile clients [22]. When operating disconnected, clients act as pseudo-servers and allow the user to work entirely out of the client cache. The trick to making a disconnected session successful, then, is to make sure that all necessary file objects are cached. Coda uses two techniques to prepare for voluntary as well as involuntary disconnection. These two techniques are *prioritized cache management* and *user-assisted hoarding*.

In the version of Coda described by Kistler [22, 23], each volume cached by Venus is in one of two stable states (*connected* or *disconnected*) or in a transitory state (*reintegrating*). While connected, operations are serviced as in AFS with minor differences caused by the details of replication. While disconnected, operations are serviced as long as the object is cached locally. All mutations to a volume are logged in the volume's *client modify log* (CML). Upon reconnection, the entries in the CML are replayed to the servers, in a process called *reintegration*. If reintegration is successful, the volume transitions to the connected state and nothing further need be done. Otherwise, the problematic operations are excised from the log, marked for user attention and the volume then transitions to the connected state. Obviously, this description has glossed over many important details regarding the system's activities in each of these states. For the most part, these details are irrelevant for the purposes of this thesis. One exception, however, is what happens in preparation for disconnected operation.

The key to maintaining the illusion of connectivity is ensuring that all file system objects the user will access while disconnected are cached prior to disconnecting from the network. This is a tricky, but critical task. Coda uses prioritized cache management together with user-assisted hoarding to enable the user to specify what objects are likely to be accessed. Prioritized cache management allows Venus to combine explicit information (optionally provided by the user) with implicit information (gathered by Venus) to manage the content of its file cache.

The user's role in this system is to assign priorities to file system objects[3]. Venus stores these user-assigned priorities, called *hoard priorities*, in the Hoard Database (HDB). User-assigned priorities range from 0 to 1000, with higher values representing greater importance. Venus determines an object's *overall priority* using a linear combination of the object's hoard priority and the object's *reference priority* [22]. The reference priority, implicitly set by Venus, measures how recently the object has been accessed. Symbolically,

$$p(x) = \alpha \cdot h(x) + (1 - \alpha) \cdot r(x)$$

where $p(x)$ is the object's overall priority, $h(x)$ is the object's hoard priority, $r(x)$ is the object's reference priority, and α is a constant called the *horizon parameter*. When space is required to fulfill a demand request, Venus evicts the object with the lowest overall priority.

The horizon parameter, which ranges from zero to one, balances the hoard priority and the reference priority. Values closer to zero favor recently referenced objects and force the cache to rely upon locality of reference to provide data during failures. Values closer to one favor objects explicitly mentioned by the user as being important during disconnected operation, and force the cache to rely upon user-supplied information to provide data during failures. Currently, the horizon parameter is set to 0.75.

Venus' goal is to maintain *cache equilibrium*, by ensuring that the cache contains only the highest priority objects at all times. Maintaining a perfect equilibrium, however, would be impractical because it would cause unacceptable overhead and could potentially suspend user activity for long periods of time. Instead, Coda takes a periodic approach to maintaining cache equilibrium. Every 10 minutes, Venus performs a *hoard walk*. Users may also initiate hoard walks, and typically do so immediately before voluntarily disconnecting from the network.

During a hoard walk, Venus equilibrates the contents of the cache, ensuring that the highest priority objects are available. Hoard walks occur in two phases. During the first phase (called the *status walk*), Venus ensures that the highest priority objects have a valid

[3] The user specifies this information to better prepare for disconnected and weakly connected operation. The information is optional. If provided, it need not be complete. In the absence of information, the object is managed using only the implicit information.

status block cached. During the second phase (called the *data walk*), Venus fetches the data for those same objects.

When all the pieces fit together, users are blissfully ignorant of whether they are connected or disconnected. Unfortunately, the illusion crumbles the moment the user attempts to access an object that was not anticipated. The request results in an ETIMEDOUT error message, and the user becomes painfully aware of the fragile nature of disconnected operation.

2.2.3 Weakly Connected Operation

Mummert extended Coda to allow clients to operate over *weak* network connections [33, 34]. She defined a weak network connection to be one with high latency, low bandwidth, or high costs. Thus, a client operating over a weak network is said to be operating *weakly connected*, whereas one operating over a network not characterized by high latency, low bandwidth, or high cost is said to be operating *strongly connected*. Many aspects of the system had to be rethought to support weakly connected clients.

Of these, the aspect most relevant to my work is reintegration. Originally, reintegration was a transitory state used to transition from the disconnected state to the connected one. The system performed this transition by replaying the mutations stored in the CML. The problem is that it replayed the entire CML. If the CML was large, this process could dominate the weak network for a substantial period of time, blocking all other network activity. Although this design worked well for clients reconnecting to a strong network, its performance for clients connected via a weak network was abysmal. To correct this problem, Mummert introduced the idea of reintegrating the CML in pieces proportional to network bandwidth. She called this process *trickle reintegration*.

The implementation of trickle reintegration required her to transform the transitory reintegrating state into a stable state, called the weakly connected state. The new volume state diagram is shown in Figure 2.1. A volume in the weakly connected state services read requests as if it were connected, but logs mutating operations to the CML as if it were disconnected. Mutations remain there until they have *aged* a reasonable amount of time. Aging each mutation allows the system to continue to take advantage of the proven benefits of *log optimizations*. Log optimizations reduce the size of the CML substantially, thus conserving disk space and reducing the amount of data that must be transmitted to and processed by the file servers [23]. Once a set of operations has aged sufficiently, it is considered ripe for reintegration. The reintegrating process locks a subset, or *chunk*, of these ripe operations to prevent ongoing mutations from optimizing them out from under the reintegrating process. The size of this set depends upon the current network bandwidth and is determined by calculating how many updates can be transferred within a time limit imposed by Venus. If the reintegration succeeds, the operations in the set are removed from the log. If it fails, the operations are unlocked and any pending optimizations are processed. Between reintegrating these chunks, the reintegration process defers to high priority network traffic. In this way, updates are “trickled” back to the servers.

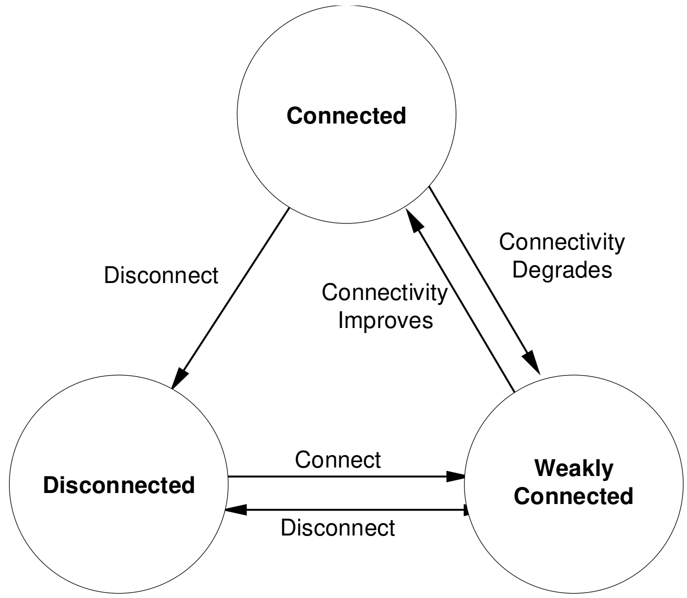

Figure 2.1: Venus States and Transitions

This figure illustrates Venus volume states and transitions as described by Mummert [33].

A second aspect that required modification and is relevant to this work is the way in which Venus validates its cache after a disconnection. Disconnected operation encourages users to increase the size of their cache (limited only by the amount of free space on the disk) and fill that cache with hoarded data. Upon reconnecting to the network, Venus validates that entire cache, one object at a time. Over an Ethernet, this process was not noticeably painful, but over a modem, it could take up to 20 minutes! To address this problem, Mummert introduced *volume callbacks*. Volume callbacks increase the granularity at which clients track server state. Each volume now has a version stamp that the client can store and compare with the current version stamp on the servers. If those stamps are equal, the client knows that all objects from that volume are valid. If the stamps are unequal, the client must resort to validating individual objects within a volume. These large granularity callbacks allow Venus to validate a large subset of the name space with a single RPC almost 97% of the time, substantially reducing the costs associated with validating the entire contents of even a large cache following a network partition [33].

2.3 User Interfaces

To judge the contributions of this thesis, it is important to understand the interfaces available to users of Coda. Coda users are provided with a tool, `repair`, to help them manually repair files that have experienced conflicting updates. Users of disconnected operation are provided with two additional programs. The first, `hoard`, enables the user to specify the hoard priorities of file system objects. The second, `spy`, assists the user in identifying objects to which she should assign priorities.

2.3.1 The Repair Tool

During a network partition, an object may be updated in ways that cannot be automatically resolved by the system. In such situations, the user must manually repair conflicts using a special tool, called `repair`. This program offers a command-line interface that diagnoses the problem, generates suggested actions to repair the object's replicas, and, after confirmation, issues the commands to fix the object in question.

2.3.2 The Hoard Program

The special tool with which users provide hoard information to Venus is called the `hoard` program. It provides a simple front end to the HDB interface [22]. This interface allows entries to be added to or deleted from the HDB. In addition, it allows the contents of the HDB to be listed, cleared, and walked. The command set for the `hoard` program directly reflects this interface and is shown in Figure 2.2.

Figure 2.3 illustrates some typical input to the `hoard` program. These examples are taken from my own hoard profiles. Each line of input consists of a command, a pathname, a hoard priority, and possibly meta-information. In each of the above examples, the command was `add` (abbreviated by the '`a`'). The number, which follows the pathname, is the hoard priority I have assigned this object. Optional meta-information, such as "`:c`" and "`:d+`", can be added to input lines whose pathnames specify a directory. '`c`' indicates that the current children of this directory should be hoarded, while '`d`' indicates that the current descendants of this directory should be hoarded. Adding a '+' indicates that *future* children or descendants should also be hoarded. In essence, the hoard program provides an assembly language for hoarding.

2.3.3 The Spy Tool

In addition to the `hoard` program, Coda provides a tool, called `spy`, which enables users to observe all Coda file references. The tool works with the cooperation of Venus. It connects to Venus over a socket and Venus informs it of every file open event that arrives from the kernel. These open events are printed to standard output. The tool makes no attempt to separate open events based upon the process or user generating them.

This tool has proven particularly helpful in tracking down file references made by other programs. One example of a program that requires this sort of tool is `latex`, a document formatting utility. When processing a document, `latex` accesses other executable programs as well as many different fonts and library files. Some of these file references are relatively easy to track down because they are explicitly referenced in the document source file. Others, however, are used as auxiliary files and are not specifically mentioned in the document source file. These are the files that require a `spy`-like tool. Figure 2.4 shows the output produced while spying on the LaTex program.

Command	Function
add filename [priority] [:{c|d}[+]]	Add `filename` to the database with optionally specified `priority` and meta-expansion attributes (‘`c`’ and ‘`d`’ indicate children and descendant expansion respectively, with ‘+’ indicating the dynamic rather than static variant).
clear [uid]	Clear all database entries or those for specified `uid`.
delete filename	Delete database entry with specified `filename`.
list filename [uid]	List all database entries or those for specified `uid` to specified output file.
walk	Perform an immediate hoard walk.

Figure 2.2: Hoard Program Command Set

This table describes the parameters associated with each hoard command. This table is provided courtesy of Jay Kistler [22].

```
# Hoarding individual programs
a /usr/local/tex/bin/latex 50
a /usr/local/tex/bin/xdvi 50
a /usr/local/tex/bin/bibtex 40

# Hoarding project files
a /coda/project/synrgen 20:c+
a /coda/project/synrgen/cfg 20:c+
a /coda/project/synrgen/doc 20:c
a /coda/project/synrgen/lib 20:c+
a /coda/project/synrgen/include 20:c+
a /coda/project/synrgen/man 20:c+
a /coda/project/synrgen/src 20:c+

# Hoarding private files
a /coda/usr/mre/thesis 100:d+
```

Figure 2.3: Sample Hoard Profile

These lines were taken from three of my personal hoard profiles, `tex.hdb`, `synrgen.hdb`, and `personal.hdb`. In the first set, I hoard individual programs associated with formatting my documents. The second set hoards specific directories inside the SynRGen project directory. In each case, the directory as well as its current and future children are hoarded. The third set hoards my thesis directory and all of its current and future descendants.

```
<7f0000f1, ./i386_mach/omega/.COMMON/bin/latex>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/bin/pstex>, 2660, 50
<7f0000f1, ./i386_mach/omega/bin/virtex>, 2660, 50
<7f0000f1, ./i386_mach/omega/bin/xdvi>, 2660, 50
<7f0000f1, ./i386_mach/omega/bin/bibtex>, 2660, 40
<7f0000f1, ./i386_mach/omega/lib/formats/psplain.fmt>, 2660, 50
<7f0000f1, ./i386_mach/omega/lib/formats/lplain.fmt>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/macros/article.sty>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/macros/art10.sty>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/macros/psfonts.sty>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/macros/times.sty>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/macros/plain.bst>, 2660, 40
<7f0000f1, ./i386_mach/omega/.COMMON/lib/fonts/times.tfm>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/fonts/timesb.tfm>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/fonts/timesit.tfm>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/fonts/courier.tfm>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/pk/lw/common/pk300>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/pk/lw/common/pk300/cmbsy10.300pk>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/pk/lw/common/pk300/circlew10.300pk>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/pk/lw/common/pk300/cmr10.746pk>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/pk/lw/common/pk300/lasy6.300pk>, 2660, 50
<7f0000f1, ./i386_mach/omega/.COMMON/lib/pk/lw/common/pk300/cmmib10.329pk>, 2660, 50
                                         .
                                         .
                                         .
```

Figure 2.4: Sample Spy Output

The figure shows output typical of the spy program. For each Coda object referenced, spy shows the volume in which that object resides (e.g., 7f0000f1) as well as a volume relative pathname (e.g., ./i386_mach/omega/.COMMON/bin/latex). In addition, it shows the id of the user who has hoarded the object and the hoard priority it has been assigned.

2.4 Summary

The Coda file system has been greatly influenced by its AFS heritage. Many of the underlying assumptions and much of the basic design are the same. Coda's primary goal, however, differs. Rather than focusing on issues of scale, Coda focuses on high availability. It offers continued access to data during periods of complete isolation from servers as well as during periods of weak connectivity with servers. Coda assumes that users are willing to help the system prepare for these periods of poor connectivity and that users are willing to accept the responsibility for fixing any conflicting updates that occur. It provides basic tools to the user for performing these tasks.