Operation-based Update Propagation in a Mobile File System

LEE, Yui-Wah

李銳華

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

in Computer Science and Engineering

Thesis Committee:

Leung, Kwong-Sak

Lui, Chi-Shing, John

Wong, Man-Hon

Satyanarayanan, Mahadev (Carnegie Mellon University)

January 2000

For Wing

UMI Keywords:

Distributed File Systems, Mobile Computing, Operation-based Update Propagation, Surrogate, Forward Error Correction.

Revision History:

Submission for examination - January 26, 2000

University Offical Version (Version 6.0) - May 9, 2000

Minor Revision (Version 6.1) - June 28, 2000

Abstract

This dissertation addresses a bottleneck problem in *mobile file systems*: the propagation of updated large files from a *weakly-connected client* to a server. It proposes a new and more efficient mechanism called *operation-based update propagation*, or *operation shipping*. In the new mechanism, the client ships the *user operation* that updated the large files, rather than the files themselves, across the weak network. (In contrast, existing mechanisms use *value shipping* and ship the files.) The user operation is sent to a *surrogate client* that is strongly connected to the server. The surrogate replays the user operation, re-generates the files, and checks whether they are identical to the originals. If so, it will send the files to the server on behalf of the client. If not, it will report the failure to the client, which will then *fall back* to use value shipping to propagate the files. Note that the new mechanism neither compromises *correctness* nor hampers *server scalability*.

There are two types of operation shippings: *application-transparent* and *application-aware*. Their feasibilities and benefits have been demonstrated by the design, implementation, and evaluation of three realistic prototypes. These prototypes are extended from three existing systems: the *Coda File System*, the *Bourne Again Shell* (`bash`), and the *GIMP* (an image application). The `bash` shell is chosen as an example interactive shell that works with Coda in application-transparent operation shipping; the GIMP is chosen as an example interactive application that works with Coda in application-aware operation shipping. In the former case, many existing non-interactive applications do not need to be modified for being involved in operation

shipping; in the latter case, the existing interactive applications do need some moderate modifications.

The prototypes have been evaluated using controlled experiments. The experiments have demonstrated the huge performance improvements of operation shipping, which ranged from 40 percents to nearly three orders of magnitude in terms of reduced network traffic and elapsed time.

Besides, this work has made the following three main contributions: techniques for fixing *non-repeating side effects* (a novel use of *forward error correction*, and the technique of *temporary-file renaming*), a technique for re-enabling the use of *cancellation optimization* with operation shipping, and a study on the *design alternatives* for the support of application-aware operation shipping.

論文撮要

移動式檔案系統中基於演算的傳送更新的方法

本論文旨在解決**移動式檔案系統**中的一個瓶頸問題：在只有低頻寬的**弱網絡**的情況下，怎樣可以更快地把更新了的檔案，從客戶機傳送到伺服機上。其解決方案，稱為**基於演算的傳送更新的方法**（下稱新傳送方法）。其原理如下。由於弱網絡頻寬有限，不宜於傳送巨型的檔案，因此如其傳送被更新的檔案，不如傳送引致檔案被更新的**用戶演算**（相比之下，傳統的方法可以被稱為**基於算值的傳送更新的方法**，因為檔案本身是用戶演算的算值。）。用戶演算會傳送到**代理機**（一部與伺服機以寬頻的強網絡所連接好了的特殊客戶機）。代理機會負責把用戶演算重演一次，以重新產生有關的檔案，並檢測其是否與原檔案相同。若是，則代理機可代表原客戶機把檔案傳送予伺服機；若否，則代理機會通知原客戶機其無能為力。客戶機退而求其次，會因應而採用傳統的傳送方法。應注意新方法既無危及到傳送更新檔之**正確性**，也沒有妨礙到伺服機在**服務多客戶機情形下的性能表現**。

新傳送方法可再細分為兩類，其一是對應用程式透明的、其二是應用程式參與其中的。本研究設計、實現、以及評估了幾個原型系統，並從而顯示了新方法既可行、又有效。這幾個原型是通過擴充三個現有的系統來實現的，即：Coda檔案系統，bash，以及GIMP圖像編輯程式。bash與Coda示範了第一類新傳送方法；而GIMP與Coda則示範了第二類新傳送方法。前者對現有程式**兼容**；而後者則需要溫和地改動現有的程式。

通過對照實驗，我們評估了原型系統的性能表現。其結果顯示，新傳送方法有極佳的表現。在實驗中，它把傳送更新所需的網絡負荷以及時間大幅減少，其幅度由百分之四十以至於近乎三個數量級。

除此以外，下列是本論文的其他三個主要貢獻：消除**不能重演的副作用**的技術（包括**向前糾錯編碼**的一個新應用，以及一個稱為**臨時檔改名**的技術），使**刪除優化**與新傳送方法可以並存的技術，以及探討了第二類新傳送方法（即應用程式參與其中的）的兩個不同設計方案。

Acknowledgments

Throughout these four years, there were hurdles and difficulties, but I have been lucky that helping hands were always around.

First, I must thank my supervisor, Professor Kwong-Sak Leung. I have learned a lot from him. Technical matters aside, I have learned from him much wisdom about life. Also, even though he has a very busy schedule, he is always available for help and advice. Most importantly, he has given me a free hand to pursue what really interests me. Without his open-mindedness, this thesis could not have been possible.

I am also in deep gratitude to Professor M. Satyanarayanan (or Satya, as we usually affectionately call him) in Carnegie Mellon University. He has been a constant source of advice and support for me, from thesis matters to career planning. He is really a great mentor and a true friend of mine. He always gives me the most appropriate advice – even if it might not be to his ultimate benefit. Furthermore, he has invited me to visit Coda group in CMU for one year, which proved to be a turning point of my research career. Also, I am honored that Satya agrees to serve in my thesis committee.

I must express my appreciation to Professor John C.S. Lui and Professor Man-Hon Wong. They both serve in my thesis committee and have been teachers and friends of mine for many years.

In CMU, I had the chance to work with the Coda and Odyssey groups. I believe they are among the best computer-science research groups in the world. I am grateful for the help of the many current and past members of the two groups: Bob Baron, Peter Braam, Anne Byrne, Maria Ebling, Tracy Farbacher, Jason Flinn, Jan Harkes, Qi Lu, Lily Mummert, Dushyanth Narayanan, Brian Noble, Henry Pierce, Josh Raiff, Roy Taylor, Eric Tilton, and Kelvin Walker. I must especially mention David Eckhardt. He is one of the greatest guys that I have met in Pittsburgh. Also, I am indebted to Jay Kistler, Puneet Kumar, Hank Mashburn, and David Steere. Although they have left the Coda group by the time I was there, their contributions to Coda and RVM definitely have enlightened me a lot.

Many other friends and colleagues of mine have provided me great support throughout the years, and the following is an incomplete list. In CUHK: Lui-Yuen Lai, Alan S. H. Lam, Fiona N. Lam, Hon-Man Law, Kim-Wai Law, Wai-Shing Luk, Wallace S. H. Or, Angus T. T. Siu, Oldfield K. Y. So, Peter T. S. Tam, Anita Y. Y. Wong, Terence Y. B. Wong, Tien-Tsin Wong, Tony S. T. Wu, and Gang Xing. In CMU: Armin Biere, Joaquin Fernandez, Chi-Keung Luk, Harry H. Y. Shum, Ye-Yi Yang, Jie Yang, Wei-Yi Yang, and Yun-Shan Zhu.

I must also acknowledge Matt Mathis of Pittsburgh Supercomputer Center and Phil Karn of Qualcomm, Inc. Matt gave me the idea of using forward error correction for handling the non-repeating side effects of time stamps, and Phil allowed me to use his package of Reed-Solomon Code in my prototype.

I am indebted to my parents, Chuen-Hing and Mui-Ying Lee, and my sister and brothers, Yee-Fun, Ho-Wah, and Cheuk-Wah, for they have given me a family filled with love and care.

Finally, my wife, Wing, deserves greater thanks than I can possibly give. She encouraged me to pursue my PhD study, at a time when I was still in doubt. I often think that she knows me better than I myself. Her supports for me, in various forms, over the years have been enormous. For example, in 1996, I decided to visit CMU for one year and would thus be separated from her. She did not complain; instead, she supported me enthusiastically. We mitigated the hardship of separation by staying together in all major vacations. For that she flew three times from Hong Kong to Pittsburgh and earned 48,000 miles of mileage in one year! Without her love and sacrifice, I could never have come this far. Wing, I love you.

Contents

List of Figures

Chapter 1

Introduction

Most computer users know the great value of distributed file systems (DFSs). However, seldom of them use these systems with their mobile computers. There are deep reasons behind this phenomenon. The key reason is that most of the DFSs have been designed with the assumption that computers are connected together by *strong networks*, which have high bandwidths and low latencies. This assumption is true in an environment where the computers are stationary, but it is not true in an *mobile computing environment* where many computers are mobile and do not have strong network connectivities most of the time.

Since around 1990, designers have noted the needs to accommodate DFSs to the new environment of mobile computing, and have achieved some successes. Concepts such as *disconnected operation* and *weakly connected operation* have been vividly prototyped and firmly accepted by the designer community. However, as of the time of writing of this thesis, *mobile file systems* still have not been used by the mainstream.

Besides reasons of marketing, implementation, and user training, there are also technical reasons: mobile file systems are still not good enough, and bottlenecks still present. This dissertation addresses an important bottleneck: the update propagation of large files over a weak network from a client to server, and it shows that the bottleneck can be alleviated.

This chapter will set the context of the thesis. Starting with a discussion on distributed file systems, it progressively "zooms in" to the bottleneck problem that this thesis is addressing. It then motivates the key idea of the solution, highlights the major research questions, and then states precisely the thesis claim. It will also mention briefly the other alternative solutions to the same bottleneck problem, and present a roadmap of the whole dissertation.

1.1 Distributed File Systems

If we ask some computer users why their computers need to be networked, they will probably give you four answers: (1) to access the Internet – for exchanging email messages, and for surfing the World Wide Web, etc; (2) to share expensive resources such as laser printers; and (3) to share files with other users; and (4) to access the same set of files from different locations. The first two answers should be self-explanatory, and are beyond the scope of this thesis. This thesis is more concerned with the third and fourth answers.

There are two approaches to sharing files: either the users explicitly transfer their files to and from different computers, or they organize their files and their computer systems around a distributed file system.

The first approach is useful only when the sharing is ad hoc in nature. When the sharing is more frequent and systematic, explicitly transferring files would become too tedious. Moreover, transferring a file over a network actually amounts to creating a new copy of the file on the remote machine. When there are more than one copies of a file, the problem of *versioning* follows. For example, many computer users should have experienced the following: "Which copy of this file is the latest one?" "I edited and added different things on both copies, how can I merge them?" "Have I mistakenly edited the other copy but not this *master* copy? Some of my changes may be missing!"

For frequent and systematic sharing of files, a better approach is to use distributed file systems, such as Sun Microsystems's NFS [42, 37], Carnegie Mellon University's and IBM's AFS [51, 13, 45], or the DCE Distributed File System [17].

DFSs are usually designed using a *client–server* model. Files and directories are physically located in a server (or a group of servers). The server *exports* a file-system service to its clients. The clients thus can *mount* the exported file-system on local *mount points*. Once this is done, applications on the clients access the mounted remote file system and the local file system using the same application programmer's interface (API). In other words, they treat the remote file system as if it is local.

As a result, a DFS decouples the logical view and the physical location of a stable storage (usually a disk). When a user on a client machine instructs an application to "save this file to the *disk* (of this machine)," the file is actually saved to a "disk" that may physically be on a remote server machine.

There are many advantages of using DFSs.

- **Location independency of stable storage.** Many computer users use more than one computers. For example, a user may use both her computer at work in daytime and her computer at home at nighttime. Another example is the case for the students in a university. They normally do not have dedicated machines. Rather, when they go to a computing laboratory, they just use any machines available there. Location independency is a very important advantage to these users, because they do not need to bother with the physical location of their working files and directories. Using DFSs, they can see the same logical view of their files and directories on different machines.

- **Sharing of information.** Sometimes, we want to use a file without having to make an explicit copy of it. One example is that a user Joe want to print a file of another user Maria. If the file is stored in a DFS, with appropriate access permissions, Joe can just go into Maria's directory and print the file. Another more important example is for system administrators. Every computer in an organization needs many software packages. To install these packages, a naive solution would be to make an explicit copy of them to each machine. With modern disk technology, doing so is not too expensive in terms of hardware cost. However, the maintenance of these machines will be a nightmare. This is because whenever a software package is to be upgraded, the system administrators will need to copy the new version explicitly to all machines again. Therefore, they prefer a better solution for installing the software packages. The better solution is to let the machines to share the same logical view of the software packages by using DFSs.

- **Delegation of tedious backup duties to the professionals.** Most computer users lack the time and the professional skills to handle many important maintenance duties, most notably the backing up of their data. With DFSs, the backing up of users' data is easy, because the data are physically located on the server machines, and the users can delegate the backup duties to the system administrators.

1.2 The Challenge of Mobile Computing

Note that all the above advantages apply also to mobile computing environments. It would be nice if mobile users can also enjoy the same advantages. However, they cannot yet. This is because many DFSs are not quite well-adapted to mobile computing environments. As we know, most DFSs were first developed in the late 70's and the early 80's. At that time, the concept of mobile computing was almost unheard of.

The early DFSs were designed with a fundamental assumption that the client machines and the server machines are connected by good networks. These networks have the characteristics of high bandwidth, low latency, and high availability. Another way to express the assumption is that the clients and the servers are *strongly connected.* The assumption matches the reality well in the old computing environments, in which the computers are stationary and are connected by local area networks (LANs), such as the 10-Mbps Ethernet.

However, in the new era of mobile computing, the assumption needs to be changed. Whereas most stationary computers in office environments are strongly connected, many other *mobile computers* are not. These mobile computers include laptops, notebooks, sub-notebooks, and PDAs (personal digital assistants), which are always on the move. Also, they include home computers, which support the mobility of users.[1] Although strong networks are common in many offices, mobile computers are usually *weakly connected*. Typical networking technologies that are available to mobile computers are wireless modem (typically 9.6 Kbps), wired modem (typically 56 Kbps for downlink, and 38.4 Kbps for uplink), ISDN line (typically 64 Kbps), ADSL line (typically 1.5 Mbps for downlink, and 64 Kbps for uplink), and wireless LAN (typically 2 Mbps). Moreover, many mobile computers are occasionally totally disconnected.

Some people may think that with the advances in wireless communication technologies, mobile computers will soon be strongly connected too. I am pessimistic on this prediction. Although the technologies of wireless communication advance rapidly, so do the technologies of wired networks. So, there is always a disparity in the bandwidths available to stationary and mobile computers. For example, when the mainstream modem technology used by home users has recently been upgraded from 28.8 Kbps to 56 Kbps, the Ethernet technology has also undergone a major upgrade – the bandwidth has increased from 10 Mbps to 100 Mbps. In general, in the foreseeable future, I believe that "Mobile elements are resource-poor relative to static elements"[48] – the bandwidth available is one such resources.

At the same time, the average file size is growing with the processing, storage, and communication capacity of computers. In 1981, a study [43] shows that 50% of the

[1] We probably can use other names such as "weakly-connected computers." but then these name are not as well-understood as "mobile computers."

files under study are smaller than 5 blocks – each block being 128 36-bit words – or roughly 2.8 Kbytes. Nowadays, files greater than 64 Kbytes are very common.

The two facts – the disparity in network bandwidths available to stationary and mobile computers, and the growth of file size – combine together and make it painful to use DFSs on weak networks. A traditional DFS would use a weak network as if it is strong, and the traffic that the DFS generate would simply jam the network. To conclude, DFSs need to adapt to the new mobile computing environment for them to be useful there.

1.3 Mobile File Systems

In this thesis, the DFSs that adapt well to mobile computing environments are called *Mobile File Systems (MFSs).* Of course, they are not precluded to be used in traditionally non-mobile environment.

1.3.1 Disconnected Operation

The very first thinking on the adaptation issue is "can we use a DFS when the client is disconnected from the server?" The answer is "yes." Some readers may be surprised by the answer, and the trick is the following. Assuming the network disconnection is only occasional, a DFS can transfer the needed data – they are the files and the directories – *before* or *after* a network disconnection. During a network disconnection, the DFS hides from the users the unpleasant fact of disconnection, and continues to provide file-system service.

This leads to the concept of *disconnected operation*, or *operation in a disconnected mode*. It is a combination of three mechanisms: (1) before a disconnection, a DFS client caches file-system data aggressively; (2) during the disconnection, the client continues to provide file-system services by using the cached data for read requests, and by logging the update operations for write requests; (3) after the disconnection, the client sends the logged updates to the file server.

James Kistler has demonstrated that disconnection operation can indeed be used effectively in a DFS [18]. His work is ground breaking, and has triggered many other research projects using similar concepts, some in the context of DFSs, some in other contexts [15, 59].

1.3.2 Weakly-connected Operation

Disconnected operation is great. However, as Lily Mummert put it, "it is not a panacea." [31, 30] A mobile client operating in a disconnected mode suffers from many limitations. First, although there is already the concept of *hoarding* [18] – or predictive caching – cache misses still have non-zero probability to happen. When they happen, they impede the users' work. Second, updates made by a user on the disconnected client are delayed from propagating to the server. The second limitation manifests in several ways.

First, users on other clients cannot see the updates until the disconnected client reconnects to the server. This delay of seeing the latest updates is acceptable to most users. However, it should be kept small.

Second, the updates may be lost if unfortunately the client is lost, damaged, or stolen before the updates are propagated to the server. This risk is relatively high comparing to the case for stationary clients, because "mobility is inherently hazardous." [48]

Third, if a DFS is using *optimistic replica control* (Section 2.2.2), then a prolonged delay of update propagation implies a higher chance of *update conflicts*, which happen when a user on another client updates the same data item during the delay.

Finally, there is a higher chance of *resource exhaustion* on the disconnected client. For example, the client may run out of local disk space that is used for logging the updates.

To sum up, disconnected operation is useful but has limitations. These limitation can be overcome by *weakly-connected operation*, or *operation in a weakly-connected mode*. Note that many mobile clients are not totally disconnected but rather having certain forms of weak network connections, which can be exploited to provide a better service to the users.

There are several mechanisms that can be used in this venture: a better communication-layer adaptation, a rapid cache validation, a better cache-miss handling, and a better update-propagation mechanism [31, 30, 32].

The last mechanism is the most relevant to this thesis. The idea is the following. A mobile client makes use of the available weak network and propagates its updates to the server earlier. However, it should *not* slow down the client's perceived speed, which in this case is the processing rate for mutating file-system calls. Note that, without a careful design, propagating updates to the server can indeed slow down the processing rate. To understand why. Let us compare the processing of a mutating file-system call on a disconnected client and a weakly-connected client. On the one hand, the disconnected client logs the request locally. On the other hand, the weakly-connected client, if naively designed, processes the call by sending data across the slow network. The processing on the disconnected client can be faster than that on the weakly-connected client, because the former involves a fast local disk but the latter involves a slow network.

Therefore, in a carefully designed update-propagation mechanism for weakly-connected clients, the foreground processing of a mutating file-system call should be *decoupled* from the actual update propagation. That is, when the file system receives a mutating call from an application, it logs the call locally and finishes the foreground processing immediately, and the actually update propagation is done in the background. As a result, the user using the application does not need to wait for the slow update propagation to complete, and perceives that the processing of the call is fast.

This mechanism is called *trickle reintegration* in Mummert's work [31, 30]. It is similar in spirit to *write-back caching*, which is used in Sprite [33] and Echo [27], but there are important differences. On the one hand, write-back caching is intended for a strongly connected environment. It preserves strict Unix write-sharing semantics, aims at the reduction of file-system latency, and only delays updates by order of seconds. On the other hand, trickle reintegration is intended for a weakly connected environment. It exploits optimistic replica control, aims at the reduction of both the latency and the network traffic volume, and can delay updates by order of hours.

Mummert has demonstrated that trickle reintegration is indeed useful. First, the decoupling make the systems perceived to be fast. She has shown that there were no perceivable difference in the speed of processing file-system calls even though the network bandwidth had been varied by four orders of magnitude. Second, by carefully choosing an *age* for updates to be refrained from propagation, some updates cancel each others in a mechanism called *cancellation optimization*, which achieves significant network traffic reduction.

However, updates eventually have to be propagated to the server. They can be delayed but cannot be eliminated. Also, the network traffic needed for the propagation is not small even after cancellation optimization. The next section will examine the problem.

1.4 Update Propagation of Files

Many modern files are big. The shippings of these big files across weak networks are not trivial tasks. Figure 1.1 tabulates the time needed to propagate some files of different sizes on different networks. The different network speeds are selected with the following reasons. 10 Mbps is a typical speed of local area networks found in many office environments, and 64 Kbps, 28.8 Kbps, and 9.6 Kbps are the typical speeds found in many mobile computing environments. The different file sizes are picked to

File size (bytes)	Example (nature; size in bytes)	Network speed (bps)			
		10M	64,000	28,800	9,600
16K	comment.txt (text file; 23K)	0.0	2.0	4.4	13.3
64K	usenix99.tex (latex source file; 57K)	0.1	8.0	17.6	53.3
256K	usenix99.ps (PostScript file; 240K)	0.2	32.0	70.4	213.3
1024K	libvicedep.a (static library file; 1866K)	0.8	128.0	281.6	833.3

Figure 1.1: Propagation Times for Files on Different Networks

This table shows the theoretical time needed, in seconds, to propagate a file of a given size across networks of different speed. The second column gives some example files, found in our environment, that are about the same sizes as the given file sizes. The natures of the files and the real sizes of the files are given in parentheses.

cover the full spectrum of modern usage of DFSs. These are typical sizes of real files, as indicated by the example files listed also in the table.

The figure points out that there is a mismatch between the sizes of modern files and the bandwidths of networks used in mobile computing environments. For example, it takes 213.3 seconds to transfer a 256-Kbyte file across a 9.6-Kbps network. Most users do not accept such a long delay.

Trickle reintegration can alleviate the problem to some extent, because an application issuing a mutating file-system call needs not wait for the long delay when the update propagation is put into the background. Also, cancellation optimization can alleviate the problem by reducing the network traffic volume generated by mutating file-system calls.

However, the propagation of uncancelled updates is still a bottleneck of DFSs. For example, in a study carried out by Mummert et al. [31, page 151 - 154], they measured the performance of trickle reintegration of four 45-minute sessions. At the end of the sessions, the amount of data pending for propagation[2] ranged from 1,060 Kbytes to

[2]They call the amount "End CML" in the paper

2,538 Kbytes. These amounts translate to 833 to 2,115 seconds of shipping times on a 9.6-Kbps modem network.

These long shipping times cause either or both of the following two inconveniences to the users. First, they are visible to the users when the users are concerned of the completions of the update propagations. For example, consider a user who has just finished work and wants to propagate her updates to the server before shutting down the network, she will have to wait for the long delay. Second, the shipping times represent the durations during which the weak networks are congested by the traffic of update propagations. Note that the networks are important resources for many other purposes, such as for electronic mails or for the World Wide Web – the traffic volumes of the DFSs should be kept small to leave rooms for the other purposes.

In other words, the update propagation of big files across a slow network is still a bottleneck problem to be addressed, and this thesis is focused on the alleviation of the bottleneck.

1.5 The Thesis

1.5.1 Motivation

The solution proposed in this thesis is called *operation-based update propagation*, or, for brevity, *operation shipping.* It is motivated by the following two observations:

1. **Big file, small operation.** Many big files are created, modified, or updated by some *user operations* that can be easily intercepted and compactly represented.

2. **CPU cycles are cheap.** The cost of re-executing the user operations to regenerate the files is often much cheaper than that of shipping the files across weak networks.

Let us see a motivating example here.[3] Suppose a file-system client finds that it has the following three newly updated files pending for propagation to the server: `usenix99.dvi`, `usenix99.log`, and `usenix99.aux`. These three files has probably been updated by the execution of a user command [23]: `latex usenix99.tex`.

In this example, the three files together has a total size of 95 Kbytes, but the command string has only 19 bytes. The ratio of the two numbers is approximately 5000:1. In other words, although the three files are quite big to be shipped across a weak network, the command string is very compact and is easy to be shipped. Moreover, the command can be intercepted easily at the level of the interactive shell[4].

Besides, the re-execution of the command is also cheap. It can be finished within five seconds on a modern computer (say, on an Intel Pentium MMX 200MHz machine). In contrast, shipping 95 Kbytes of data across a 9.6-Kbps network needs 79.2 seconds. The ratio of the two numbers is approximately 16:1. To sum up, this example exemplifies the two observations stated previously. In Sections 6.2 and 6.3, we will see more examples of a similar nature.

With the two observations, the idea of a new update-propagation mechanism becomes obvious. In the new mechanism, the compact *operation*, instead of the big updated files, is shipped across a weak network for update propagation. And then, on the other side of the network, the operation is replayed or re-executed so as to re-generate the files that are meant to be propagated. This mechanism is therefore called operation-based update propagation, or operation shipping. In contrast, the traditional mechanism of shipping the files is called *value shipping*, because the updated files can be regarded as the *values* produced by the user operation.

The idea is conceptually simple. However, to really use it in mobile file systems, we need to address a number of research questions:

[3]This example later becomes the test T16 in Chapter 6

[4]such as the "Bourne Again Shell" (`bash`)

1. How are user operations logged? What kind of user operations can be logged? Who are responsible for logging them?

2. How are user operations re-executed? Will server scalability be hampered?

3. How about the correctness of the update propagation? What should the file system do if the re-execution does not re-generate the same file?

This thesis is a study to answer these research questions, so as to firmly establish the following thesis statement.

1.5.2 Thesis Statement

> *Operation-based update propagation is a feasible and beneficial mechanism in a mobile file system. It is feasible because it can be deployed rather easily into existing systems, and it does not compromise correctness nor hamper server scalability. It is beneficial because it can reduce significantly the network traffic.*

1.5.3 Validation of the Thesis

Three realistic prototypes have been designed, implemented, and evaluated to validate the thesis claim. They are extended from three existing systems: the *Coda File System*, the *Bourne Again Shell* (`bash`), and the *GIMP* (an image application). After the extensions, Coda works with `bash` in *application-transparent* operation shipping, and with the GIMP in *application-aware* operation shipping (the meaning of the two types of operation shipping will be explained in Section 3.5).

There are many advantages in building realistic prototypes. First, realistic prototypes are the best demonstration that an idea works. Second, they allows realistic evaluations to be performed. Finally, there are always issues that can be uncovered only by really building and using the systems.

1.6 Alternative Solutions

Before concluding this chapter and going into the main body of the thesis, it is appropriate to mention briefly the other possible solutions to the problem being addressed. There are at least four alternative solutions proposed. First, *delta shipping* reduces network traffic by shipping only the incremental difference between different versions of the file meant to be shipped. Second, *data compression* reduces network traffic by "compressing out" the redundancies in the file meant to be shipped. Third, someone has proposed to log and replay *the keystrokes*. Fourth, someone has proposed to implement operation shipping *without involving the file system*. These four solutions will be discussed in more detail in Chapter 7.

1.7 Document Roadmap

The rest of this document is organized as follows. Chapter 2 presents the background about Coda, which is the basis upon the prototype mobile file system is built. Chapter 3 provides an architectural overview of the new shipping mechanism. It also answers two important questions. The first is how we can avoid hampering server scalability, and the second is to how we can preserve correctness.

As will be explained also in Chapter 3, there are two types of operation-shipping mechanisms: application-transparent and application-aware. These two mechanisms are discussed respectively in Chapters 4 and 5.

Chapter 6 reports on the implementation status of the prototypes that have been built for this work, and it evaluates their performance. It shows that huge performance gains of operation shipping in terms of network traffic reduction and elapsed time reduction.

Chapter 7 discusses related work, and Chapter 8 concludes with a summary of the thesis work and its contributions and a discussion of future work.

Chapter 2

Coda Background

This chapter describes the context of this research: the Coda File System. It provides an overview of the system and hence introduces many architectural and implementation features of the file system that affect the design of the operation shipping mechanisms to be discussed in the following chapters. Coda has been described quite extensively in the literature [11, 49, 53, 19, 18, 52, 20, 31, 30]. Readers are referred to the literature should the brief discussion in the following is not sufficient.

Coda is such a complex system that describing it in one chapter is a daunting task. This chapter looks lengthy, because the discussion is organized along the evolution path of Coda. This approach may put more materials than the absolute minimum that are needed to understand the rest of this thesis, but it provides a better organization for the discussion. The readers are recommended to only briefly skim through this chapter to get a rough idea about Coda, and to come back to this chapter later when necessary. In the rest of this document, back pointers to sections in this chapter will be provided when appropriate.

Coda is a descendant of the AFS-2 file system. [1] The design goals of AFS are

[1] Carnegie Mellon University, together with the IBM Corporation, developed three different versions of AFS. The Coda development team took the code base of the second version and started developing Coda in the late 1980s.

to provide a scalable, high-performance, secure, and easy-to-administer file system. The original design goal of Coda was to enhance the availability of AFS. Two complementary mechanisms for high availability – *server replication* and *disconnected operation* – were thus added to Coda. It turned out that the second mechanism of disconnected operation is good for not only high-availability but also for mobile computing. This is because disconnected operation allows a client to continue operations in the face of not only an *involuntary disconnection* – an availability concern – but also in the face of a *voluntary disconnection* – a concern for mobile computing.

Following the line of supporting mobile computing, Coda's designers found that total disconnection is actually a special and extreme case of *weak connectivity*. Between the two extremes of strong connection and no connection, there is a wide range of weak connection that can be exploited to provide even better support for mobile computing.

The following sections follows the above evolution path, discussing one by one the following four main evolution steps: AFS-2, server replication, disconnected operation, and weakly-connected operation.

2.1 Architectural Features Inherited from AFS-2

To support the design goal of scalability, performance, security, and operability, AFS has several main architectural features, which have been in turn inherited by Coda.

2.1.1 Careful Distinction of Servers and Clients

To support better scalability and security, AFS and Coda make a *careful distinction between servers and clients*. Servers are also collectively known as *Vice*, which is a small nucleus of a large distributed system. Servers are physically separated from the rest of the machines and run only a few trusted software systems. They are looked after by dedicated personnel and are trusted. On the other hand, clients, collectively known as *Virtue*, are at the peripheral of the distributed system. They may be located anywhere on the network and run arbitrary software systems. They are looked after by the users or by some decentralized staff, such as the support staff in individual departments of the organization. They have a high chance of being subverted by malicious users, and the network segments connecting them may be eavesdropped. Therefore, they are not trusted from the security point of view.

For better performance seen by the users of a client, most *end-user computations* are performed on the client. The users see a better performance because the client machine is shared by only a few users, or is even dedicated to only one user.

For better sharing, easier administration, and location-independency of users, Vice exports a *distributed file-system service* to the clients.

On each client, there is a process called *Venus* that communicates with Vice and exposes the file-system service to applications running on the client, using a close approximation of the BSD 4.3 UNIX [2] interface (Figure 2.1). Applications make use of the service through some *file-system requests*, which are also referred to as *file-system calls*. As with other UNIX file systems, there are three kinds of *file-system objects*: regular files, directories, and symbolic links. The shared objects have a distinctive name space: every AFS's object has a pathname prefix of `/afs`, and every Coda's object has a pathname prefix of `/coda`.

The *primary replicas* of these objects are physically stored in Vice, but they are cached extensively on the clients for better performance. Cached replicas on the clients are regarded as *secondary replicas*. The cache structure and the cache-coherence scheme are to be discussed in the next two sub-sections.

AFS and Coda use the following three measures to enhance the system security [44], which is an important concern for a large scale system. First, servers and clients are treated very differently in terms of their security levels and the trusts they have. Second, because the clients and the network segments connecting them cannot be

[2] BSD refers to the family of Berkeley Software Distributions of UNIX.

File-System Call	Description
`access`	Determine access permissions of an object.
`chmod`	Change mode bits of an object.
`chown`	Change owner of an object.
`close`	Close an previously `open`'ed file.
`creat`	Create a new file.
`fsync`	Synchronize a file's in-core state with that on disk.
`ioctl`	Perform a control function on an open descriptor.
`link`	Make a hard link to an object.
`lseek`	Move the read/write pointer for an open descriptor.
`mkdir`	Make a directory with the specified path.
`mknod`	Make a special file.
`mount`	Mount a file system.
`open`	Open a file for reading or writing, or create a new file.
`read, readv`	Read input from an open descriptor.
`readlink`	Read value of a symbolic link.
`rename`	Change the name of an object.
`rmdir`	Remove a directory.
`stat`	Get object status.
`statfs`	Get file-system statistics.
`symlink`	Make a symbolic link to an object.
`sync`	Synchronize a file system's in-core state with that on disk.
`truncate`	Truncate a file to a specified length.
`umount`	Remove a file system.
`unlink`	Remove a directory entry.
`utimes`	Set access and modification times for an object.
`write, writev`	Write output to an open descriptor.

Figure 2.1: 4.3 BSD File System Interface

In the description, the term “object” refers to a file, a directory, or a symbolic link. Source: Adapted from Mummert [30], Table 2.3, page 15 trusted, secure channels are provided for the communications between clients and servers. AFS and Coda use a special RPC (remote procedure call) package – RPC2 [47] – that supports both the authentication scheme and a RPC-packet-level private-key encryption.[3] Third, AFS and Coda use *access-control list (ACL)* as a finer grain protection specification scheme for various access rights on directories.

2.1.2 Cache Structure

Coda inherits the cache structure and cache-coherence scheme of AFS. Both of them are crucial to the performance of the file system in a large scale. This sub-section describes the cache structure, and the next sub-section describes the cache-coherence scheme. The cache structure has four distinctive characteristics.

First, Venus uses *whole-file* caching but not *block-oriented* caching. This simplifies the design of the cache structure and the cache-coherence scheme, and it demands a smaller file-transfer overhead per bytes. It was motivated by the observation that most files were small, and the whole files will be needed anyway in most file accesses [35, 7, 5]. [4]

Second, Venus caches file-system objects, both their status and data, in *non-volatile memory*. This has two implications. First, the cache can be much larger, and therefore more effective, since the local disk – a common form of non-volatile memory – on a client machine is usually one to two order-of-magnitude larger than the virtual memory (RAM plus swap space). Second, when a client reboots, it does not need to reload the cache from the server. Note in particular that a disconnected client does not have

[3]In the package, hooks are provided for adapting different private-key encryption. Currently, only a simple encryption scheme – the XOR scheme – is available. Furthermore, in Coda, the packet encryption is usually turned off. This is because Coda is still largely used as research prototypes, and the security concern is less eminent. As more and more production system are installed, the secure-communication infrastructure will become more and more important.

[4]Modern files are no longer small. Also, more and more Coda clients are being used with weak networks. Therefore, the design decision of whole-file caching may worth a revisit in the near future.

contact with any servers. In terms of implementation, the data part of file-system objects are kept in local Unix files called *container files*. These files are allocated in a cache directory, such as `/usr/coda/etc/venus.cache`. However, the status part is being treated differently in AFS-2 and in Coda. In AFS-2, status are also stored in local Unix files, but in Coda, they are instead stored in *recoverable virtual memory* using the RVM package [55, 28]. Kistler provided a detailed explanation on the change[18].

Third, Venus caches both the status and the data of file-system objects. In particular, the *naming information* is also cached. This allows the clients to perform *pathname translation*, which is the translation of a globally unique pathname to a globally unique 96-bit identifier – *the file identifier* or *fid*. The moving of the responsibility of pathname translation from the server to the client helps reduce the client–server communication and the server workload.

Fourth, Coda treats the *status* and *data* caches *separately*. A client can cache the status of a file-system object without its data. However, it always caches the data of an object together with its status. This differential treatment allows a client to process the `stat` system call – a frequent event – with less communication with a server.

2.1.3 Callback Cache-Coherence Scheme

Coda maintains the coherence of its clients using the notion of *callback*. A callback scheme is an *invalidation-based* scheme rather than a *validate-on-use* scheme, and it goes as follows. When a client caches an object from a server, the server establishes a *callback promise*. The promise is that the server will invalidate the object by sending the client a *callback break* when the cached copy is no longer valid (this happens when another client has updated the object). With this promise, the client is assured that its cached copy is fresh so long as it does not hear otherwise from the server. Therefore, it needs not re-validate the freshness on every use. This scheme reduces significantly the client–server communications as well as the server workload. However, it complicates the server's structure, since the server needs to keep states about the callback promises that it has made. The possibility of network disconnection complicates the matter further, as we shall see in the subsequent sections.

Coherence is maintained at the temporal granularity of *sessions*. Most file-system requests form their own *simple sessions*, but the `open` and the `close` requests are the exceptions, and a matching pair of them form a *open–close compound session*. The advantage of this scheme is that it reduces the frequency of client–server communications, and it is in line with the whole-file caching scheme. However, it has a disadvantage that it has relaxed the UNIX file-sharing semantics, as discussed in the following. Although processes on the same machine still see the precise UNIX semantics (since they are both accessing the same local cached copy of a file), processes on different machines do not. For example, if a process A is mutating a file, another process B, which runs on another machine, will see the mutation of A only after A has closed the file and B has re-opened the file. Fortunately, usage experience has shown that such a relaxation of semantics is acceptable.

2.1.4 Volumes

One of the key concerns of AFS and Coda is the ease of administration in a large scale. To this end, they both use a structuring primitive called *volume*. A volume is a collection of files forming a partial subtree in the Vice name space. Volumes are like mountable Unix file systems, but are considerably more flexible.

Volumes are glued together by *mounting* one on another. A leaf node of a volume can be a *mount point* on which another volume can be mounted. By mounting volumes on volumes, the whole Vice name space is formed. The *root volume* is special, as it is not mounted anywhere but forming the root of the name space. A mount point is implemented as a symbolic link with a special mode bit. Venus recognizes mount points and crosses them transparently during name translations.

A volume is an administrative unit. Typically, a volume is assigned for a user or a project. Volumes are physically *in custodian* by some servers, and they can be moved from one server to another in a way transparent to the users. The physical location of a volume is not explicitly stated in neither the high-level identifier – pathname – nor the low-level identifier – fid. Rather, a client looks up the actual location of a volume from a *Volume Location Database (VLDB)*.

2.2 Server Replication

Server replication is an mechanism to increase the availability of the Coda file system. By replicating data on multiple servers, the probability that the data are inaccessible due to server or network failures is reduced. However, the challenge is how to maintain the consistency of the replicated data after partitioned updates, and how to resolve the situation when some data items are inconsistent.

2.2.1 VSG and AVSG

Replication is done at the granularity of volumes. The replication factor of a volume is defined at the volume creation time, and can vary from one to eight. Typically, volumes are doubly or triply replicated. The set of servers that are the current *custodians* of a volume is called a *Volume Storage Group (VSG)*. Due to server or network failures, the number of servers that a client can contact may be reduced. The smaller set of servers that can be reached for a given volume is called an *Accessible Volume Storage Group (AVSG)*.

2.2.2 Optimistic Replication

When there is a network partition, a client cannot talk to some of the servers of a give volume, nor can it obtain an exclusive lock on all servers. Therefore, when it is trying to update an object, another client in the other partition may be updating the replica of the same object concurrently, and a *write–write* conflict may occurs. Facing this possibility of conflict, there are two different approaches.

A *pessimistic* approach assumes that conflicts are likely, so partitioned updates are either forbidden totally, or are allowed only under very restricted conditions, such as the successful obtaining of a quorum, or the successful obtaining of a token. However this approach severely limits the availability of the system. In contrast, an *optimistic* approach always allows partitioned updates, and provides a higher availability.

Coda uses the optimistic approach. To maintain consistency, the system keeps enough information of the partitioned updates. Upon the removal of the partitioning condition, it *detects* all objects that are in an inconsistent state, and *resolves* as many as possible of them. For unresolvable inconsistent objects, the system marks them for manual repair. Usage experience has confirmed that write-sharing is indeed rare in a distributed file system [20] and justifies the optimistic approach used by Coda.

2.2.3 Version Vector

Coda uses the notion of *Version Vectors* for the concurrency control of replica.[5] The dimension of a version vector is the same as the replication factor of an object. So a n-way replicated object has a version vector $V =< V_1, V_2, \ldots, V_n >$. Each server keeps a version vector with the replica of an object. In the version vector, V_i represents an *estimate* of the number updates that the object has received on the i-th server. Note that, because of possible network failures, the server can only estimate the number of updates on the other servers. The maintenance procedure for version vectors will be

[5] Version Vector was first used in the Locus File System[39].

described in the following sub-section.

Updates are always allowed even when there is a network partition. When the partitioning condition is removed, the version vectors kept on each server help to determine what had happened on the object. Suppose there are two version vectors from two servers A and B, there are three possible outcomes:

- If every component of the version vector of A is equal to the corresponding component of that of B, then the object must have received an equal number of updates on both servers, and they must be identical.

- If every component of the version vector of A is greater than or equal to the corresponding component of that of B, then the object must have been updated in A's partition but not in B's. In other words, B has missed the updates made on A due to the network partition. The replica in A is said to be a *dominant* copy. The remedy of this inconsistent state is simple: B's replica should be replaced with A's.

- If some components of the version vector of A is greater than the corresponding component of that of B, but some other components is smaller, then the object must have been updated in both A's and B's partitions. A missed B's updates, and B missed A's updates. In this case, the two replicas are *diverging*. It is not right to either replace A's replica by B's, or replace B's replica by A's. This is a true write–write conflict, and need to be resolved.

2.2.4 Replica-Control Algorithm

When a client accesses an object, it uses the approach of *read-status-from-all, read-data-from-one, write-all*. When serving a file-system request, Venus reads the status from all accessible servers. The status contains the version vector. If the version vectors are the same, Venus can fetch the data from a *preferred server*. If the version vectors differ, Venus will block the request and trigger a *resolution* on the servers. Only if the resolution succeeds will Venus proceed to service the request. In the unfortunate case that the resolution fails, the object is marked as inconsistent. It is no longer accessible until the user *repairs* it manually with the aid of a *repair tool*.

When serving an update requests, Venus writes simultaneously the new version of the object to all accessible servers. It also generates a *store ID* and ask the servers to stamp the object with the store ID. A store ID has two components: `<host_ID, seq>`, where `seq` is a monotonically increasing number. A store ID tells which client is the last updater of a replica of an object. A protocol called the *Coda Optimistic Protocol (COP)* is used for the updating of objects, and it proceeds in two phases.

In the first phase (COP1), Venus sends the update request simultaneously to all servers, and each server will proceed with the update independently. The update may succeed in some servers but fail on the others. By collating the responses from all servers, the client constructs a list of servers who made the update successfully. This list is called the *update set*. The set can also be represented as a vector of the same dimension as the version vectors, but the components only takes the value of either 0 or 1. For the i-th component, a 0 means the update failed on server i, a 1 means the update succeeded on server i.

In the second phase (COP2), Venus sends the update set to all servers. This allows the i-th server to update all the j-th components, $j \neq i$, on the version vector that it keeps. The RPC2 package used by Coda supports MultiRPC, which allows a parallel communication from one client to multiple servers [47].

2.2.5 Resolution and Repair

When Venus detects that the version vectors kept on different servers for an object differ, it triggers a resolution. Coda uses different resolution mechanisms for *files* and *directories*, because the two types of objects are different in nature. Directories are structured objects, and the file system understands the update operations performed on them. In contrast, files are unstructured byte streams, and the file system has no knowledge as to how an application may update them. The following two paragraphs describe the two resolution mechanisms respectively.

Because update operations on directories are understood by the file system, resolving a directory amounts to finding the missed update operations and replaying them. In Coda, every server maintains for each volume a *resolution log*, which remembers all the directory-update operations performed on the volume. The resolution protocol proceeds as follows. One of the servers is elected as the *co-ordinator*. It collects the resolution logs from all servers, compares the logs, constructs a partitioned-update log, and sends the log to all the servers. Each server then compares its own resolution log with the partitioned-update log, determines the missed operations, and replays the missed operations.

For the resolution of files, Coda uses the following approach. If a replica dominates all the other replicas (Section 2.2.3), Coda will resolve the files by replacing all replicas with the dominant replica. If the replicas are diverging and there is no dominant replica, Coda may still be able to resolve the inconsistent file replicas with the aid of an *application-specific resolver (ASR)*. An ASR is provided by someone who understands the file format used by an application. For example, a calendar resolver can merge diverging calendar-file replicas by knowing how to interpret the calendar file format.

Unresolvable objects are marked as inconsistent. They are represented as dangling symbolic links. A repair tool is provided to assist the manual repairing of the inconsistencies. It has two components: one for repairing *server-server conflicts*, which arise due to server replication; and the other for repairing *client-server conflicts*, which arise due to disconnection operation (Section 2.3).

2.2.6 Replicated Server and Callbacks

The fact that objects are replicated on multiple servers complicates the original callback-based cache-coherence scheme. Callbacks are now kept on each replicated server. For each volume, when the AVSG shrinks, Venus may miss some callback breaks sent from an unreachable server. When the AVSG grows again, Venus must drop the callbacks for all objects in the volume, and re-establish them with the servers in the enlarged AVSG.

2.3 Disconnected Operation

In a traditional distributed file system, a client stops working when it loses contact with the server. Coda pioneered the concept of *disconnected operation*, which masks as much as possible the unpleasant fact of network disconnection and allows the client to continue working. A client enters into disconnected operation for a volume when the AVSG of the volume becomes an empty set. Network disconnections can be *involuntary*, such as the case due to a server or network failure, or they can be *voluntary*, such as the case when a user unplugs her mobile computer from the network.

The key to disconnected operation is aggressive client-side caching. This is because if the client has already cached the data and/or status of the needed file-system objects, then it can continue to service requests using these cached objects even when it is disconnected from the server. However, to make this possible, there are three challenges:

1. Before a network disconnection, the client must cache file-system objects in preparation of the disconnection. The cache-management scheme must reduce the probability of *cache misses*, which impede progress and reveal the network disconnection.

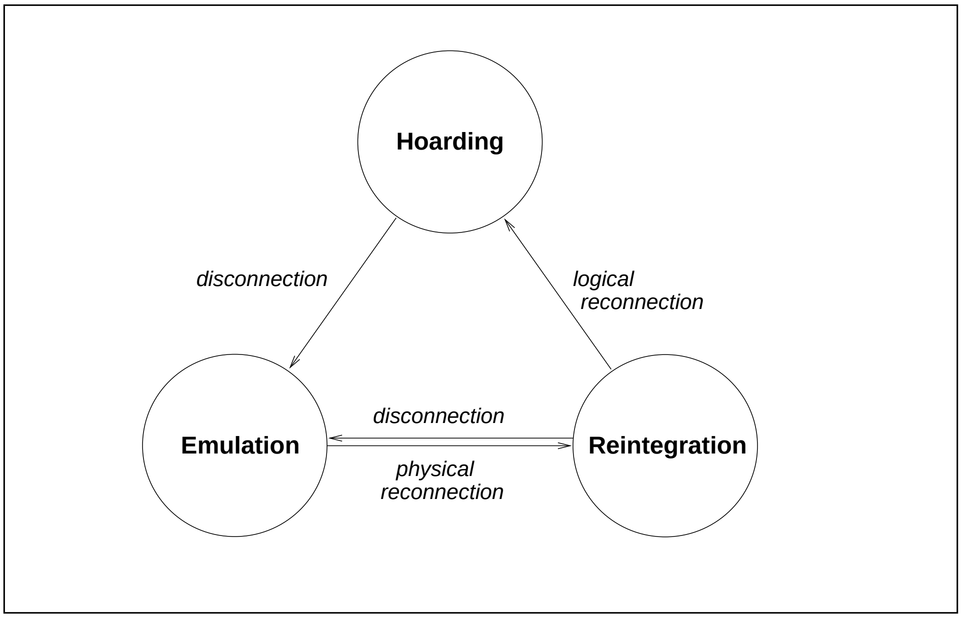

Figure 2.2: Three Volume States for the Support of Disconnected Operation

Source: Adapted from Kistler [18], Figure 3.1, page 54.

2. During the network disconnection, the client continues to service file-system requests using cached objects. Both read-only and mutating requests are serviced. For the latter, the client *logs* the requests so that they can be *replayed* later on the server upon reconnection.

3. After the network disconnection, the client must merge the disconnected updates with the server in a correct way. Conflicts must also be detected and handled.

The three challenges happens respectively in the three possible states of a volume: *hoarding*, *emulation*, and *reintegration*. Figure 2.2 shows the three states and the events that trigger the state transition. Note that, since each volume has its own volume storage group and accessible volume storage group (Section 2.2.1), it has it own volume state too. That is, different volumes on the same client can be in different states.

The following three sub-sections discuss respectively the three states as discussed in [18, 19]. Note that, since then the transient reintegration state has been changed to a long-lasting *write-disconnected* state. The change is needed for the support of weakly-connected operation, and will be discussed in the Section 2.4.2.

2.3.1 Hoarding

When some servers in the VSG are accessible, a volume is in the *hoarding* state. In this state, Venus has all the connected-mode responsibilities for the volume: servicing file-system requests, fetching objects from the servers and storing them back to the servers, etc. Mutating requests, in particular, are *written* directly to the server through the cache. Venus also needs to participate in the callback cache-coherence scheme and the Coda Optimistic Protocol.

On top of these activities, Venus has one more duty: a careful management of the client-side cache. Unlike ordinary cache, the cache must cater also for the long term need of the user to prepare for possible disconnections. Coda allows users to specify their long term needs in *hoard profiles*. The hoarding algorithm considers both the reference patterns as well as the hoard profiles to strike a balance between short and long term needs.

2.3.2 Emulation

A volume enters into the *emulation* state when its accessible volume storage group (AVSG, Section 2.2.1) becomes empty. In this state, Venus emulates the unreachable servers and services file-system requests using the cached objects.

File-system requests come in two flavors: read-only and mutating. The servicing of read-only requests is simple. Venus just presents the cache objects to the requesting applications. The servicing of mutating requests, on the other hand, is more complex. Besides performing the requests on the locally cached objects, Venus must also *log* enough information about the requests so that they can later be replayed on the server when the network connection is resumed.

To admit more partitioned updates, Coda uses an *inferred-transaction* model of access. In this model, file-system calls are internally mapped to transactions based on their semantics and how they use the accessed data items. Figure 2.3 shows the different types of update transactions, and the mapping from Unix file-system calls. A transaction model admits more partitioned update than a shared-memory model, because it exposes the structure of computation. Coda infers transactions from the the type of file-system calls, instead of adding new transactional facility to be invoked explicitly by the applications. This approach allows existing applications to be benefited from the transaction model.

2.3.2.1 Client-Modify Log (CML)

The key data structure for keeping the disconnected update information is a per-volume *client-modify log (CML)*. Each record in the log corresponds to a logged update transaction (Figure 2.3). A log records is appended to the log when a local update transaction commits. To simplify reintegration and cancellation optimization (will be discussed respectively in Sections 2.3.3 and 2.3.2.2), each non-empty CML is owned by exactly one user. So, other users are not allowed to perform mutating file-system calls on the volume during a disconnected session. This restriction seldom imposes serious problems, since each Coda client is typically used exclusively by one user.

Figures 2.4 and 2.5 show respectively the type-independent and type-specific parts of a CML record. Among the different type of CML records, `store` is special. This is because the new value of a `store` transaction, `new_contents`, can be very large. The new value contains all the bits of the regular files being stored, and can be as large as thousands or millions of bytes. In contrast, the new values of all other types of records are compact, and they are usually less than a few hundreds bytes.

Therefore, `store` records are treated differently. Instead of storing all the bits of the `new_contents` field, Venus only stores a pointer to a *container file*, which is

`chown` (*object, user*)
    `chown`
`chmod` (*object, user*)
    `chmod`
`utimes` (*object, user*)
    `utimes`
`store` (*file, user*)
    `[ [create | open] [read | write]* close] | truncate`
`create` (*directory, name, file, user*)
    `creat | open`
`mkdir` (*directory1, name, directory2, user*)
    `mkdir`
`symlink` (*directory, name, symlink, user*)
    `symlink`
`remove` (directory, name, file, user)
`remove` (directory, name, symlink, user)
    `rename | unlink`
`rmdir` (*directory1, name, directory2, user*)
    `rename | rmdir`
`link` (*directory, name, file, user*)
    `link`
`rename` (*directory1, name1, directory2, name2, object, user*)
    `rename`
`repair` (*file, user*)
    (no mapping)

Figure 2.3: Coda Updates and System-Call Mapping

The leftmost lines show the different types of Coda mutating transactions. The indented lines show the mapping from UNIX system calls, in which the syntax has the following meanings: juxtaposition represents succession, `*` represents repetition, and | represents selection. Source: Adapted from Mummert [30], Table 2.4, page 16.

```
ClientModiyLog *log;
rec_dlink       handle;

ViceStoreId     sid;
Date_t          time;
UserId          uid;
int             tid;
CmlFlags        flags;

< type specific fields >

dlist  *fid_bindings;
dlist  *pred;
dlist  *succ;
```

Figure 2.4: Type-independent Fields of CML Records

This figure shows the fields that are common to all different types of CML records. Each record contains a back pointer to the CML itself (`log`) and to its successor `handle`. The modify-time of the update is in `time`, the author of the update is indicated by `uid`. Two fields are used as "transaction identifiers"; they are the store id `sid` and `tid`. The `fid_bindings` field contains pointers to the descriptor of the file system object that the record references. Pointers to lists of preceding and succeeding records are contained in `pred` and `succ`, respectively. Source: Adapted from Mummert [30], Figure 2.5, page 17.

already in the cache. Recall that Coda uses whole-file caching, and the data part of a regular file is stored in local UNIX files called the container file (Section 2.1.2). The re-using of the container files for the `new_contents` field of the CML is called the *store-record optimization*. It is possible only when the following two conditions are true: (1) there is at most one `store` record per file in the log, otherwise, the referent of a non-final `store` is invalid; and (2) a container file referenced by a `store` record must not be replaced until after reintegration. The two conditions are indeed true, because the first condition is assured by the procedure of cancellation optimization (Section 2.3.2.2), and the second condition is assured by marking as *dirty* all container files that are referenced by any `store` records. The flag prevents the files from being replaced from the cache, and is cleared only after reintegration.

Record Type	Type-specific Fields of Record
`chown`	fid, new owner, version id
`chmod`	fid, new mode, version id
`utimes`	fid, new modify time, version id
`store`	fid, new length, new contents, version id, offset, server handles
`create`	parent fid, name, child fid, mode, version id
`mkdir`	parent fid, name, child fid, mode, parent version id
`symlink`	parent fid, old name, new name, child fid, mode, parent version id
`remove`	parent fid, name, child fid, link count, parent version id, child version id
`rmdir`	parent fid, name, child fid, parent version id, child version id
`link`	parent fid, name, child fid, parent version id, child version id
`rename`	from parent fid, from name, to parent fid, to name, from fid, from parent version id, to parent version id, from version id
`repair`	fid, length, modify time, owner, mode, version id

Figure 2.5: Type-specific Fields of CML Records

This table shows the type-specific contents of CML records. The `store` record, which represents the close of a file opened for write, includes new data by reference from a separate local cache *container* file. Source: Adapted from Mummert [30], Table 2.6, page 18.

In other words, although logically CML is one entity, it comprises two parts. In this thesis, the two parts are called the *thin* and *thick part* respectively. The thin part comprises the fields proper as listed in Figures 2.4 and 2.5, including the pointers to the container files; and the thick part comprises the pointed-to container files. In the following chapters, we shall see that the main purpose of operation shipping is indeed the avoidance of the shipping of the thick part of a CML, because it generates too much traffic for a weak network.

In the implementation, the thin part of a CML is placed in *recoverable virtual memory* using the RVM package [28, 55]. The thick part of a CML is stored as local Unix files. Both the recoverable virtual memory and local Unix files are persistent, so Venus can reintegrate even after a reboot.

Overwritten Subsequence	Overwriter
[`store` (f, u) | `utimes` (f, u)]+	`store` (f, u)
`chown` (f, u)	`chown` (f, u)
`chmod` (f, u)	`chmod` (f, u)
`utimes` (f, u)	`utimes` (f, u)
[`store`(f, u) | `chown`(f, u) | `chmod`(f, u) | `utimes`(f, u)]+	`remove`(χ, χ, f, u)
[`chown`(s, u) | `chmod`(s, u) | `utimes`(s, u)]+	`remove`(χ, χ, s, u)
[`chown`(d, u) | `chmod`(d, u) | `utimes`(d, u)]+	`remove`(χ, χ, d, u)

(a) Overwrite Optimizations

	Identity Subsequence	
Initiator	Intermediaries	Terminator
`create` (χ, χ, f, χ)	[`store` (f, χ) | `chown` (f, χ) | `chmod` (f, χ) | `utimes` (f, χ) | `link` (χ, χ, f, χ) | `remove` (χ, χ, f, χ) | `rename` $(\chi, \chi, \chi, \chi, f, \chi)$]*	`remove` (χ, χ, f, χ)
`symlink` (χ, χ, s, χ)	[`chown` (s, χ) | `chmod` (s, χ) | `utimes` (s, χ) | `rename` $(\chi, \chi, \chi, \chi, s, \chi)$]*	`remove` (χ, χ, s, χ)
`mkdir` (χ, χ, d, χ)	[`chown` (d, χ) | `chmod` (d, χ) | `utimes` (d, χ) | `rename` $(\chi, \chi, \chi, \chi, d, \chi)$]*	`rmdir` (χ, χ, d, χ)

(b) Identity Subsequence Optimizations

Figure 2.6: CML Optimization Templates

This figure shows optimizations that may be performed on CML records. Overwrite optimization replaces a sequence of log records (the overwritten subsequence) with a single update (the overwriter). Identity-subsequence optimization removes a sequence of log records, beginning with the initiator, including intermediaries, and ending with the terminator. In the table above, f is a file, s is a symbolic link, d is a directory, u is a user ID, and χ means the value of the argument is not relevant for the cancellation. All updates must be authored by the same user; this condition is satisfied trivially by CML ownership. Source: Adapted from Mummert [30], Table 2.7, page 19.

2.3.2.2 Cancellation Optimization

File-system activities often exhibit *canceling* behaviors. Here are two examples: (1) if a file is repeatedly stored, the later store operations will cancel the effect of the earlier store operations; and (2) if a file is created and then subsequently removed, interspersed perhaps with other updates to it, the remove operation will cancel the create operation and the intervening update operations.

By exploiting these behaviors, Venus can transform a CML into a smaller equivalent one by taking out canceled records. This procedure is called *cancellation optimization*. The two logs are equivalent in the sense that the results of merging the two logs with the server are the same. [6] A smaller log means less consumption on the local storage space, a smaller reintegration traffic volume, and a smaller server workload for reintegration. Trace-driven simulation and usage experience have both confirmed the effectiveness of the optimization [54, 34].

Coda uses the following on-line approach for cancellation optimization. When a new CML record is logged, Venus lookups the previous records in the log and see whether they match one of the specified templates (Figure 2.6), and proceeds to canceling if appropriate. The two preceding examples represent the two types of optimization: *overwrite cancellation* and *identity cancellation*. In overwrite cancellation, an overwriter cancels the overwritten subsequence; in identity-subsequence cancellation, a whole identity subsequence – including the initiator, the intermediaries, and the terminator – is canceled altogether. A `store` on a file is always overwritten by a subsequent `store` on the same file; this guarantees that each file can have at most one `store` record in a CML.

[6] Strictly speaking, there are two definitions of correctness: *reintegration-transparent* and *1SR-preserving*. Interested readers are referred to [18] for more details.

2.3.3 Reintegration

A once emulating volume goes into the *reintegration* state when the AVSG goes from empty to non-empty. This state is a transient state in the original model proposed by Kistler [18], and the volume is expected to return to the hoarding state after this state. Reintegration proceeds in three phases: *prelude*, *interlude*, and *postlude*. Venus dispatches a *reintegrator* thread to manage the reintegration procedures.

The prelude phase happens on the client. It encompasses the tasks necessary to start reintegration, and is triggered by a triggering event, such as a reference to an object in the volume. The most important task in this phase is the preparation of a *reintegration log*, which contains all information related to the disconnected updates. Two other tasks worth mentioning here. First, if there are some files open for update, any `store` records referring to the files will be canceled "early." Second, any *temporary fids* generated by the client during a disconnected session will be replaced by globally unique *permanent fids*, which are generated by servers. Details can be found in [18]. The client finally invokes a `ViceReintegrate` RPC with the server, marking the end of prelude phase and the begin of interlude phase.

The interlude phase happens on the server. The server retrieves the reintegration log, and unmarshalls the records from it. It then write-locks the updated objects. Transactions logged on the reintegration log are thus replayed on the server side. While replaying the transactions, the server makes a series of *concurrency*, *integrity*, and *protection* checks. The concurrency check is performed by a combination of value-based and version-based *certification* procedure. If all of the checks succeeds, the server will tentatively commit the transaction. If the transaction being replayed is a `store` transaction, the server will also *backfetches* the container file from the client. Finally, if all transactions on the log can be tentatively committed, the server will atomically commit them altogether, release the locked objects, and reply the RPC.

The reply of the RPC kicks off the postlude phase on the client. If the reintegration succeeds, the client marks dirty objects as clean and discards the CML records. It then finalizes the reintegration by transiting the volume from the reintegration state to the hoarding state.

The above description is for the cases when reintegrations succeed. In fact, reintegration may fail. The early Coda and the current Coda use two different failure models. The early Coda used a *coarse-grain* model, and the current Coda uses a *fine-grain* model. In the coarse-grain model, a single failure in one transaction renders the whole reintegration failed. Also, Venus aborts the local updates by spiriting involved objects away to a local *closure* file, purging them from the cache, and discarding the CML records.

In contrast, in the fine-grain model, if a failure occurs, the server will abort the reintegration but also returns an index containing the position of the offending record. The client may then retry with non-offending records. Also, objects that have failed to be reintegrated are marked as inconsistent *in situ*, like the cases for resolution failures. The users are again provided with a manual repair tool to repair the objects.

2.4 Weakly-connected Operation

One more step ahead from disconnected operation is the support of *weakly-connected operation*. The underlying theme is to exploit the available weak networks to deliver services better than that of the totally disconnected situations.

Four different mechanisms have been incorporated into Coda to support the uses in weak networks. Among them, *trickle reintegration* is the most relevant to this thesis, and is discussed in this section. The other three – *adaptation in the communication layer*, *rapid cache validation*, and *better handling of cache misses* – are less relevant and are skipped for discussion here. The readers are referred to [30] for the discussions of the three mechanisms.

2.4.1 Overview of Trickle Reintegration

There are two key related ideas in trickle reintegration. First, the available bandwidth, albeit small, should be exploited to reintegrate a client's updates, a small piece at a time. An earlier update propagation has many advantages: updates can be visible to other clients earlier, there is a smaller chance of lost of updates due to damage or threat of the client, the time window opened for write–write conflict will be smaller, and local resources held pending for reintegration can be released earlier.

Second, the trickling of updates should not make users' life worse on the weakly-connected client. In particular, the processing of file-system requests should not be tied to the slow speed of update propagation. To this end, update propagation is done *asynchronously*. It is decoupled from the request-processing and is put into the background. When Venus receives a mutating file-system request, the request is performed locally and is also logged for a later propagation to the server. The processing of the request can finish quickly without blocking the requesting process for the slow update propagation.

The mechanism of trickle reintegration introduces several architectural features that deserve our attention.

2.4.2 Structural Modification

As discussed in Section 2.3, the early Coda designed before adding the considerations for weakly-connected operation has a transient *reintegration* volume state (Figure 2.2). Now this state has to become a stable *write-disconnected* state (Figure 2.7). There are three reasons for this change. First, reintegration can no longer be assumed to finish quickly, as tricklings of updates across slow networks may take minutes or even hours to finish. Second, because update propagation is made a background activity, new updates should be allowed to be made on the volume. In other words, while old updates are being trickled back to the servers, new updates can be made to the volume. Third,

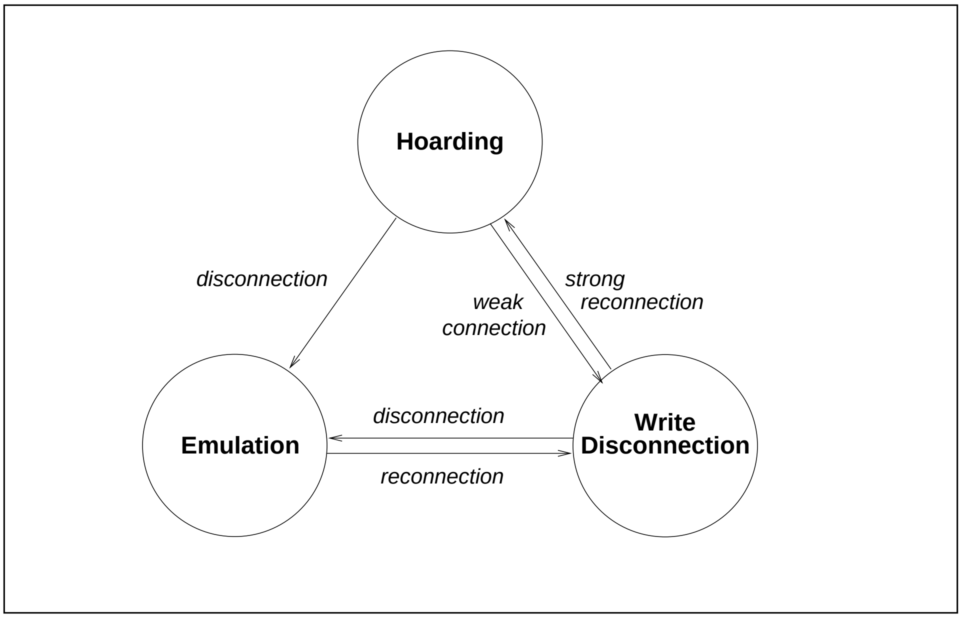

Figure 2.7: Three Volume States for the Support of Weakly-connected Operation

Note that the reintegration state in Figure 2.2 has been replaced by the write-disconnection state in the new volume state-transition diagram. Source: Adapted from Mummert [30], Figure 5.1, page 71.

as long as the network connectivity is weak, for speedier processing, mutating file-system calls are logged for asynchronous propagation. Therefore, the volume should stay in the write-disconnected state so that new updates can be logged.

The new state is so named because it is a blend of a strongly connected state and a disconnected state. As if the volume is strongly connected, Venus services cache misses, and maintains cache coherence using callbacks. As if the volume is disconnected, Venus performs updates locally and logs them in the CML – these updates are propagated later in the background.

2.4.3 Aging Window of CML

As discussed in Section 2.3.2, cancellation optimization is an effective means to reduce the size of CMLs, reintegration traffic, and server work load.

However, trickle reintegration reduces the effectiveness of log optimization. In fact,

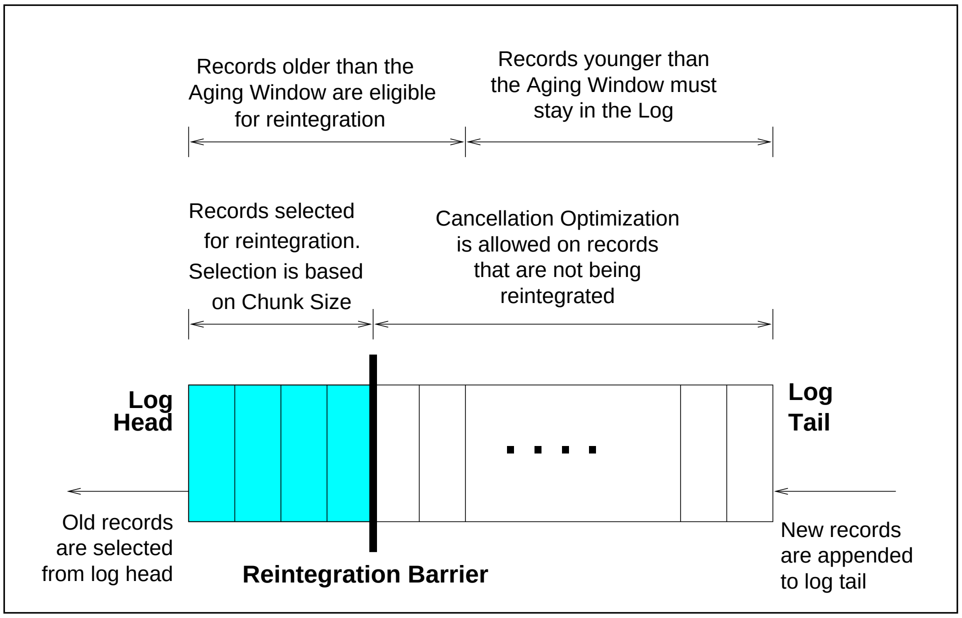

Figure 2.8: CML During Trickle Reintegration

An aging window is the minimum amount of time that a record must stay in a CML. Only records older than the aging window are eligible for reintegration. The actual number of records selected for a particular round may be smaller because of the limit of the chunk size. A reintegration barrier is placed to prevent records that are being reintegrated from being subjected to cancellation optimization.

we have two conflicting wishes. On the one hand, we want records to leave the CML and be propagated earlier, for the reasons discussed in the beginning of this section. On the other hand, we want them to stay longer in the CML so that they have a higher chance to be canceled by subsequent records.

To balance the two wishes, Coda uses a simple strategy based on aging. The age of a CML record is the amount of time that it has spent on the log. An *aging window* defines the minimum age a record must have before it is eligible for reintegration. Figure 2.8 shows the use of aging window. If probably chosen, an aging window can cover the “hot spots” in which repeated updates on the same objects are likely. Coda currently uses a fix aging window of 600 seconds. Mummert explains in detail how the parameter was chosen in [30].

2.4.4 Object-level Concurrency Control

In the early Coda, Venus locks the whole volume that is being reintegrated. This coarse-grain concurrency control is considered acceptable since reintegrations are expected to be done on a strong network and to have low latencies. However, the expectation is no longer true when Coda supports reintegration over weak networks. To avoid preventing the others from using the volume during the potentially slow reintegration, Venus now uses a *finer-grain* concurrency control: instead of locking the whole volume, Venus now locks only the individual objects. There are two new issues that need to be addressed: *file contention* and *dead-lock*. More details about how they are addressed by Coda can be found in [30].

2.4.5 Chunks and Fragments

Since update propagations are now made asynchronous to the foreground processing of the file-system requests, there is no pressing need to propagate updates all at once – doing so would have a large impact on the weak networks since reintegration latencies are no longer small now. Coda uses two measures to make reintegration traffic less intrusive to the weak networks: *chunks* and *fragment*.

Instead of sending all CML records older than a given age (Section 2.4.3), a reintegrator send them in chunks. It selects records from the prefix of a CML, in the order of the log, until the aggregate size of the records reaches a limit defined by the parameter *chunk size*. The parameter is calculated from a specified limit on reintegration time, and varies with the current network bandwidths. The default of reintegration time limit is set to 30 seconds, which translates to 36 Kbytes in a 9.6-Kbps network, and to 7.7 Mbytes in a 2-Mbps network.

Most records will fit into a chunk, excepts some `store` records. Recall that the container files can be large and may exceed chunk size by themselves. For these records, reintegrator will send the data in the container files in fragments, each of it has a size of the chunk size. Shipments of the fragments of a container file may be done in many rounds of reintegration; they may even be done across a Venus restart. The partial data of the container file are stored in a *shadow inode* on the server. Venus refers to a shadow inode using a *handle*, which is exported from the server. The server now exports four more RPCs for reintegrating a files in fragments: `ViceOpenReintHandle`, `ViceSendReintFragment`, `ViceQueryReintHandle`, and `ViceCloseReintHandle`.

2.4.6 Other Changes

Coda has two other changes added for the support of weakly-connected operation. They are only mentioned briefly here. More details can be found in [30].

First, while in strong networks Venus still uses a "*reintegrate-with-all*" strategy for replicated volumes, it now uses a "*reintegrate-with-one-and-then-resolve*" strategy in weak networks. That is, instead of reintegrating with all servers in the AVSG, Venus reintegrates with only one server, and then triggers a resolution to bring the other servers up-to-date. This change is needed to save the precious bandwidth of weak networks.

Second, a server breaks a callback whenever an object is updated, no matter whether the update is on the data or the status, or both, of the object. It would be quite unwise for a weakly-connected client to fetch unchanged data again if only the status is updated. Note that status-only updates are quite common because resolution updates status, and because weakly-connected clients now use a reintegrate-with-one-then-resolve strategy to save reintegration traffic. These unnecessary data fetches are suppressed in the new Venus.

2.5 Chapter Summary

In this chapter, some background knowledge about the Coda File System is presented. The discussion is organized along the evolution path of Coda: from AFS-2, through server replication and disconnection operation, to weakly-connected operation. Key Coda architectural or implementation features are identified. Equipped with the background knowledge, we can go into the following chapters, which will discuss the design and implementation of an extension of Coda supporting operation-based update propagation.

Chapter 3

Architecture

This chapter presents the high-level view of operation shipping to prepare the readers for the more detailed discussion in the coming chapters. First of all, an overview of the operation-shipping mechanism is presented in Section 3.1. A new party, called the surrogate, is needed in the mechanism, and the need is justified in Section 3.2. The terms "operation" and "value" have been used in the preceding discussion, and Section 3.3 presents a concrete picture of what they are. A key concern about operation shipping is the correctness of the new propagation mechanism, and Section 3.4 presents the four-step approach designed for preserving the correctness. In practice, there are two different types of operation-shipping mechanisms: application-transparent and application-aware; Section 3.5 explains why there are two different mechanisms needed.

3.1 Overview of Operation Shipping

Operation-based update propagation comprises two stages. The first one is the *logging* stage, and the second one is the *shipping* stage. The purpose of the logging stage is to establish an association between some high-level *user operations* and some low-level values that they generate, and it is a preparation stage for the shipping stage.

The shipping stage itself can be further divided into five phases. Figure 3.1 presents an overview. In the figure, a new party called the *surrogate* appears. It will be discussed in more detail in Section 3.2. For the moment, just take it as a special client strongly connected to the server.

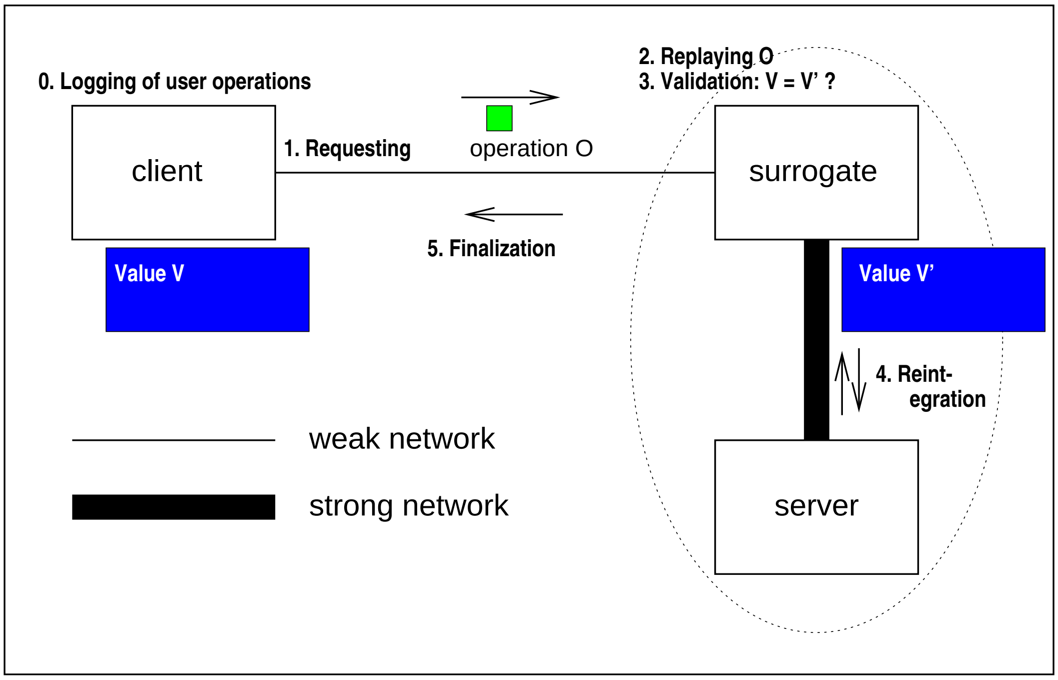

Figure 3.1: Overview of Operation Shipping

This figures presents a high-level view of operation shipping. The concept of the surrogate will be explained in Section 3.2. For completeness, the zeroth step, which is the logging of user operations, is also shown.

The five phases of the shipping stage are:

1. **Requesting.** When propagating a value V to the server, instead of sending V directly to the server, the client sends O, the user operation that generates V, to the surrogate.

2. **Replaying.** The surrogate replays O, which re-generates a value V'.

3. **Validation.** The surrogate checks whether V' is equal to V.

4. **Reintegration/aborting.** If V' is equal to V, the surrogate will reintegrate V' with the server on behalf of the client; otherwise, it will abort the attempt of operation shipping and discard V'.

5. **Finalization.** The surrogate reports to the client whether the attempt of operation shipping has succeeded, and the client proceeds to finalize the shipping state as follows. If the attempt has succeeded, the client will locally commit V; otherwise, it will *fall back* to value shipping – that is, it will ship V directly to the server.

This overview is largely simplified. The subsequent two chapters will provide more details.

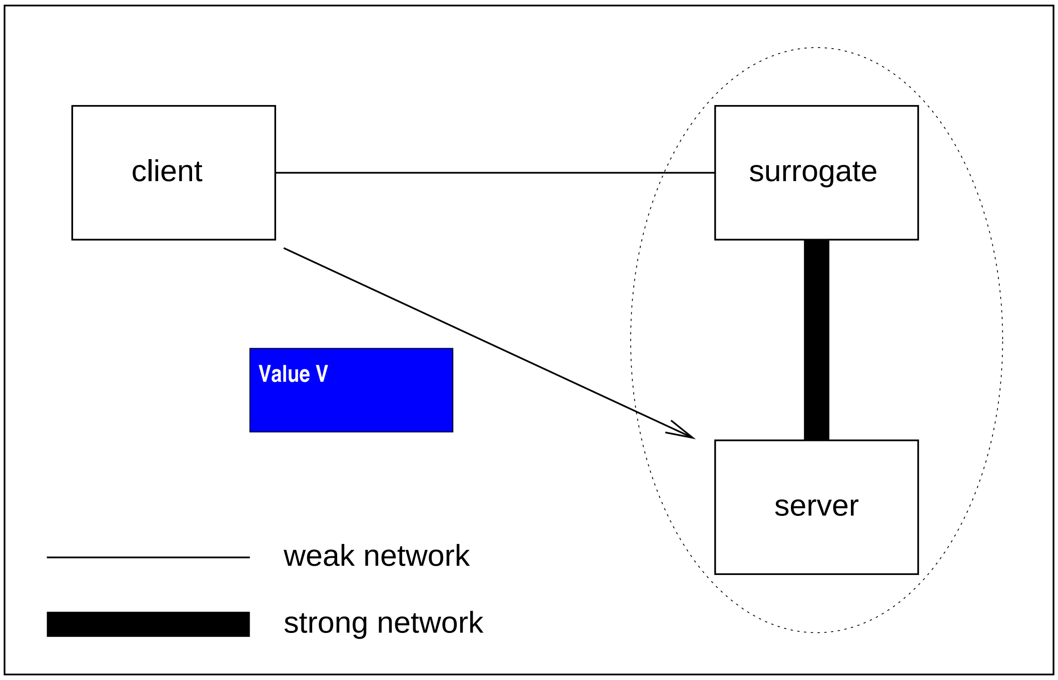

Figure 3.2: Fallback Mechanism: Value Shipping

When shipping by operation is not possible, the client will fall back to shipping by value. Note that the surrogate is not longer involved.

Note that operation shipping is always done in a "best-effort" manner. The file system is always well prepared for the cases when operation shipping does not work, and it just falls back to the traditional way of value shipping. During value shipping, the surrogate is not involved, and the client reintegrates directly with the server (Figure 3.2). The followings are some examples of the cases when operation shipping does not work.

- The surrogate cannot be contacted.
- The user does not have a valid token on the surrogate.
- The surrogate cannot re-execute the user operation.
- The re-generated value on the surrogate is different to the original.
- The surrogate fails to reintegrate with the server.

3.2 Surrogate

> "High level [user] operation logging places a greater burden on the server, because operations require greater computational resources. To preserve scalability, the system must take server load into account before agreeing to perform these functions. The client should negotiate operation logging rights upon connection, and the server should be able to revoke these rights if load becomes too heavy. [30, page 172]"

When the idea of high-level operation logging was first proposed. The main concern in people's mind is its adverse effect on server scalability, as we can see from the above discussion. This was also my main concern when I first looked at the problem. However, a deeper thinking into the issue led me to the conclusion that it should not be the server who replays the operations, rather, a new party called the surrogate is needed. This section details my thinking on this issue.

3.2.1 Location of Re-executions

A key component of operation shipping is that user operations need to be re-executed so that they can re-generate the files and other file-system modifications. Obviously, re-executions incur computational workload. So the first question is where the re-execution workload should be placed. A natural candidate seems to be the server. However, there are several drawbacks rendering re-executions on the servers impractical.

First, the server may be overloaded. In a typical installation of distributed file system, the file server serves a large number of clients. The number may be in the range of tens to hundreds, or even thousands. The file server is already very likely the bottleneck of the system [46]. At the same time, the aggregate workload of all the re-executions requested by the many clients is not trivial at all to the server, even though the workload of each individual re-execution may look trivial. Therefore, adding more workload to the bottleneck is not a wise thing to do.

Second, it is difficult to ask one server to instantiate the execution environments of the numerous clients that it services. As we know, the *execution environment* affects the result of an execution. It comprises many aspects, including the CPU, the operating system, the application software, the system libraries, the system header files, and others. For example, if we run the same compilation command on two machines, it may not produce the same result if the two machines are loaded with different versions of system header files. In general, the execution environment of the re-executing machine should be as similar to the client as possible, otherwise the result of the re-execution may not be the same as the original, and is thus useless.

Thirdly, re-executing on the server is a potential security threat. This is because the server has to run whatever software that a user claims that he uses on his client. A user with bad intention may supply a malicious program, not really for the purpose of operation shipping, but for the purpose of attacking the availability or integrity of the server. Because of this reason, and also because of the scalability concern discussed previously, many file servers prohibit the running of normal user applications but only a small selected set of applications solely for the purpose of providing file-system services. In many installations, normal users are even prohibited to login to the file servers.

The previous arguments suggest that the re-execution workload should not be placed on the server, but rather on another machine. This re-execution machine is called the *surrogate*.

3.2.2 Properties of Surrogate

The word "surrogate" means proxy or agent in the English language. The term surrogate is chosen, instead of its synonyms, because the latter are already used heavily with other computer concepts, such as web proxy and Internet search agent.

For the purpose of operation shipping, a surrogate is performing the reintegration *on behalf* of a weakly-connected client. Therefore, it needs all the machinery for reintegration with server. The easiest way is to modify Venus, the Cache Manager, so that it can support both the replaying of user operations and the reintegration of the the result with the server. Venus on the surrogate machine is thus run in a special mode `--isSurrogate`.

The following are the desired properties of a surrogate machine:

1. It should be strongly connected to the server, so that the shipping of re-execution result to the server can be done cheaply.

2. It should provide an execution environment as similar as possible to the weakly-connected client that it services.

3. It should be at an adequate level of security, and processes suitable authentication tokens for the user requesting services.

3.2.3 Dedicated Surrogate

Should a surrogate machine be dedicated to only one weakly-connected client? Can it be shared by multiple weakly-connected clients?

The first approach has many advantages. First, the surrogate can be identically configured as the weakly-connected client. Second, it simplifies the system administration. A re-configuration of a client, such as a software upgrade, can simply be matched by a re-configuration of its dedicated surrogate without worrying that so doing may break services to other clients. Third, it will be much easier to track down the problem if a user operation does not produce an result that is the same as the original.

Additionally, many computer users already own more than one personal computers. For them, dedicated surrogates can be set up without additional hardware investment. For example, a user Joe owns one computer at work, and another computer at home. He can set up his computer at work, which is strongly connected to server, as the surrogate machine of this weakly-connected computer at home. Another example is a user Mary, who owns one desktop computer at work, and another notebook computer for business trips. She can set up her desktop computer, which again is strongly connected to server, as the surrogate machine of her weakly-connected laptop computer when she is on a business trip.

This thesis assumes that the surrogate machine is dedicated to one client. The use of shared surrogate machines is an interesting future work.

3.3 User Operations and Values

3.3.1 User Operations

What exactly are user operations? For the purpose of operation shipping, they are some high-level commands of computer users that can be intercepted and logged on the client, and later be replayed on the surrogate.

There are two types of applications: *non-interactive* and *interactive*. A computer user issues commands to them in different ways. Therefore, there are two categories of user operations:

Type 1: invocation command of non-interactive application

Type 2: interactive and application-specific command

Figure 3.3 lists several examples of user operations in each category, and Figure 3.4 lists several examples of non-interactive and interactive applications. As discussed in Section 3.5, these two different categories of user operations need two different mechanisms of operation shipping.

User operations are usually compact. This is because user operations are commands issued by human users, and the updates that they causes are the results of some computations. Human users type or click much more slowly than computers can process or generate data; therefore, user operations are much more compact than the potential sizes of updates that they cause. The ratio may be as small as one in several thousands. For example, an `invert` command specific to an image application may need only a few bytes to represent, but it can potentially cause an update of an image file of several tens of kilobytes.

However, user operations should not be confused with *key strokes* and *mouse clicks*. Key strokes and mouse clicks are *raw* input. They are not yet *interpreted*, and have little meaning if they are taken out of context. In contrast, user operations are commands that have already been interpreted.

For example, in interactive shells that supports the *history mechanism* (such as the C Shell [38]), the key strokes `!!` means "repeating the last command." What exactly is the last command really depends on the context. Suppose the command `latex usenix` was the last command issued by the user, then the meaning of the key strokes `!!` is the command. Another example is a mouse click. The exact meaning of a mouse

User Operation	Application that receives the user operation	Application that performs the user operation
Type 1: invocation command of non-interactive application		
latex usenix99.tex	shell	latex
rp2gen callback.rpc2	shell	rp2gen
yacc parsepdb.yacc	shell	yacc
c++ -c counters.cc -o counters.o	shell	c++
ar rv libdir.a ...	shell	ar
tar xzvf coda-doc-4.6.5-2.ppt.tgz	shell	tar
make	shell	make
sgml2latex guide.sgml	shell	sgml2latex
Type 2: interactive and application-specific command		
load an image	GIMP	GIMP
invert an image	GIMP	GIMP
change brightness/contrast of an image	GIMP	GIMP
change color balance of an image	GIMP	GIMP
save an image	GIMP	GIMP
load a text file	emacs	emacs
insert some text	emacs	emacs
reformat a paragraph	emacs	emacs
global replace a string	emacs	emacs
save a text file	emacs	emacs

Figure 3.3: Some Examples of User Operations

Shell means an interactive shell such as the `bash`. Figure 3.4 will explain the nature of the different applications mentioned in this table.

click at a certain co-ordinates really depends on what application windows happens to be located at that co-ordinates.

The effects of user operations also depends on the context, but to much lesser extents. For example, the interpreted command `latex usenix` may have different effects if it is issued in different working directories. Therefore, the logging shell also needs to log a few context information to prepare for a successful replay of a user operation – the current working directory is one of those context information.

Name	Nature
Non-interactive applications	
ar	software library archive builder
gcc/g++	compiler and linker
dvips	converter: from TeX dvi format to PostScript format
$\LaTeX$	text formatter and typesetter
ld	linker
make	tool for managing software building processes
rp2gen	stub generator for use with the RPC2 package
sgml2latex	converter: from SGML format to $\LaTeX$ format
tar	file or tape archive packager
troff	text formatter
yacc	parser generator
Interactive applications	
Applix Presents	presentation software
Applix Word	word processor
AutoCAD	computer-aided design software
CorelDRAW	drawing package
Emacs	text editor
GIMP	image manipulation program
magic layout system	computer-aided-design software
Microsoft Word	word processor
Microsoft PowerPoint	presentation software
vi	text editor
xfig	drawing package
xv	image viewer

Figure 3.4: Some Examples of Interactive and Non-interactive Applications

3.3.2 Values

What exactly are values? For the purpose of discussion in this thesis, they are actually the records in the CML (client-modify log). The following is the explanation of why we label CML records as the values.

Recall that, for delayed update propagations, such as those in the disconnected or weakly-connected mode, a mutating file-system calls are mapped to a Coda update, and is logged in a CML. The log has different types of records as listed in Figure 2.3. Also recall that the `store` records are special, because they logically include the new contents of the files being `store`d. The new contents of a file is indeed the value of the `store` operation.

Now, the traditional way of update propagation required the shipping of the CML from a client to a server, including the new contents of the files. Therefore, it is labelled as *value shipping*,[1] and the CML records are labelled as values. Such a labelling is justified because most of the traffic volume is due to the new contents of files.

3.4 Preserving Correctness

In the context of this thesis, the correctness of an update propagation means that a server receives exactly the same values (CML records), no matter whether they are value-shipped or operation-shipped from a client. For the case of operation shipping, the values are re-generated by replaying the user operation on the surrogate. Intuitively, a correct update propagation can be achieved when the user operation is *repeating on the surrogate*. A user operation is repeating on the surrogate (or simply repeating if there is no confusion) if it produces exactly the same values as its original execution. However, the actual story is slightly more complex.

To preserve the correctness, there are four steps involved. First, several measures

[1] Despite the fact that some records, such as `chmod` or `utimes`, actually contains only *directory operations.*

can be taken to *increase the likelihood* that a user operation will be repeating on the surrogate. Second, sometimes an operation is not repeating. However, if it produces a set of values that is close enough to that of the original execution, we may be able to *fix the re-execution discrepancies*. Third, we need to *adjust the status information* of the values re-generated by the replayed operations. An example of the status information is the modification time of the mutating file-system operations. Fourth, we need a *validation procedure* as the final step. The first, third, and fourth steps will be discussed in the following three sub-sections. The discussion of the second step will be deferred to Section 4.3.

3.4.1 Increasing the Likelihood of Repeating Operations

Let us first analyze what makes an operation repeating. There are four factors. Whenever it is possible, measures are taken to make an operation more likely to be repeating.

1. **Identical Configuration.** The configurations on the surrogate and the client should be identical. In general, a configuration comprises the machine architecture, the operating system, the system header files, and system libraries, etc. As discussed in Section 3.2, each client is already assumed to have a dedicated surrogate; so it is easy to make them to have an identical configuration.

2. **Application be Deterministic and Non-time-dependent.** The application should be deterministic and is not time-dependent. This condition rules out pseudo-random applications and applications that depend on the time of execution. The domains of interest of this thesis are program-development and document-production environments. In these domains, pseudo-random applications play a minor role, so the fact that their operations are not repeating does not impose severe limitation. However, some important applications, such as the text formatter LaTeX, do put execution time stamps into the files that they

produce. The operations of these applications are indeed not repeating, and they will be handled in the step of fixing re-execution discrepancies (Section 4.3).

3. **Identical Contexts.** The contexts in which the operations are executed and replayed are identical. For non-interactive applications, the context is the UNIX process environment, which comprises four attributes: (1) the working directory, (2) the environment list, (3) the command-line arguments, and (4) the file-creation mask. For interactive applications, the context must also include the in-memory state of the running process.

4. **Identical Input Files.** If an operation requires an input file stored in Coda, we can rely on Coda to ensure that the client and the surrogate will use an identical version of the input file. Coda can ensure this because a client ships updates to its server in temporal order, and the surrogate will always retrieve the latest version of a file from the servers. For example, consider a user issuing three user operations successively on a client machine:

   - Op_1: update a source file using an editor
   - Op_2: compile the source file into an object file using a compiler
   - Op_3: update the source file again.

   When the surrogate re-executes Op_2, the client must have shipped Op_1 but not Op_3, and the replayed operation will be reading exactly the version updated by Op_1.

It must be emphasized that the file system does not guarantee that all operations be repeating on the surrogate – it just increases the probability of that happening. If an operation is not repeating, the client will simply fall back to value shipping (Section 3.1).

3.4.2 Adjusting the Status Information

In addition to updating data, user operations also update some meta-data. The meta-data are the status information of the CML records. Some of the meta-data are internal to file system and are invisible to the users (e.g., the Store ID for concurrency control); some are external and are visible to the users (e.g., the modification time). Venus on the surrogate must reset these meta-data to those of the original execution, which are packed by the clients as a part of the operation log.

3.4.3 Validation

Validation is the final step. It is done after the fixing of re-execution discrepancy and the adjusting of status information. The key question is whether the portion of CML on the surrogate, resulted from the replaying of a user operation, is identical to its counterpart on the client. If it is, the surrogate will *accept* the replayed operation, and will proceed to reintegrate the CML with the server. If it is not, the surrogate will *reject* the replayed operation and report an error to the client, which will respond by falling back to value shipping.

Recall from Section 2.3.2 that a CML physically comprises two parts: the thick part comprises the container files, and the rest is the thin part. The thick part is stored as local Unix files, and the thin part is stored in recoverable virtual memory. The thick part is much more bulky than the thin part. The main purpose of operation shipping is indeed the avoidance of the shipping of the think part of a CML, since it generates too much traffic for a weak network.

Therefore, the validation uses two different approaches regarding the two parts. First, the thin part is packed as a part of the operation log by the client. It is called the *reference CML* on the surrogate. Validation is done by comparing a reference CML to its counterpart re-generated on the surrogate.

Second, for the thick part. The client packs only a *fingerprint* of each container file.

A fingerprint is a quick summary of a file, and it will be discussed later. Validation is done by comparing the fingerprints of a container file on the client to its counterpart on the surrogate.

A fingerprint function produces a fixed-length fingerprint $f(M)$ for a given arbitrary-length message M. A good fingerprint function should have two properties:

1. computing $f(M)$ from M is easy,
2. the probability that another message M' gives the same fingerprint is small.

Here, the messages for which we compute the fingerprints are the contents of the container files.

Our prototype uses MD5 (Message Digest 5) fingerprints [41, 56]. Each fingerprint has 128 bits, so the overhead is very small. Also, the probability that two different container files give the same fingerprints is very small: it is in the order of $1/2^{64}$.

The fact that the probability is non-zero, albeit very small, may worry some readers. However, even value shipping is vulnerable to a small but non-zero probability of error. That is, there is a small probability that a communication error has occurred but is not detected by the error-correction subsystem of the communication channel. People probably can tolerate the small probabilities of errors of both operation shipping and value shipping.

3.5 Application-transparent Versus Application-aware Operation Shipping

Recall from Section 3.3 that there are two kinds of applications: *non-interactive* and *interactive*. These two types of applications are so fundamentally different that they demands two different mechanisms of operation shipping. For the former, the mechanism is called *application-transparent*; for the latter, the mechanism is called *application-aware*.

Let us first consider non-interactive applications. For this type of applications, the user operations are the invocation commands of the applications. Let us consider what happen on the client and the surrogate. On the client, a non-interactive application itself has no knowledge about the happening of the user operations, because it is just passively invoked by an interactive shell. Therefore, the logging entity in this case is the interactive shell. On the surrogate, again, the application is passively re-invoked upon replaying. Therefore, Venus on the surrogate can serve as the replaying entity (the task of re-invoking the application is so simple that we do not need another interactive shell to serve as the replaying entity on the surrogate). To sum up, the happening of user operations is totally *transparent* to the application. This transparency is important since the application does not need any modification for participating in operation shipping.

On the other hand, for interactive applications, the user operations are some interactive and application-specific commands. The knowledge about the happening of some user operations rests with the application being used. Therefore, only the application itself can be the logging entity. Similarly, it is also the replaying entity on the surrogate. In other words, the application must be *aware* of the operation-logging and operation-replaying activities. To use operation shipping for an application, it must be extended to add the logging and replaying capabilities.

Figure 3.5 summarizes the differences between the two types of operation shipping mechanisms, which will be discussed in more detail in the next two chapters.

3.6 Chapter Summary

A overview of operation shipping is presented in this chapter. The concept of surrogate, operation, and value are discussed. This chapter has answered two of the research questions posed in Chapter 1. The first is how we can avoid hampering server scalability; the answer is in Section 3.2, which discusses about the concept of surrogate. The second is how we can preserve correctness; the answer is in Section 3.4,

	Application-transparent Operation Shipping	Application-aware Operation Shipping
Application Nature	Non-interactive	Interactive
User Operation	Type 1: invocation command of non-interactive application	Type 2: interactive and application-specific command
Application has knowledge about operation shipping?	no	yes
Logging Entity	shell	application
Replaying Entity	Venus	application
Application needs modification?	no	yes

Figure 3.5: Two Different Types of Operation shipping

which discuss the four-step approach for ensuring correctness. The last section of this chapter explains that there are two types of operation-shipping mechanisms: application-transparent and application-aware. The two mechanisms will be discussed in more detail in the following two chapters.

Chapter 4

Application-transparent Operation Shipping

Application-transparent operation shipping is the simpler of the two forms of operation shipping, since a user operation is just an invocation command of a non-interactive application. It is also the form of operation shipping that people probably will accept earlier. The main reason is that it is backward compatible with respect to existing applications. In other words, it can be deployed without requiring additional efforts of modifying existing applications.

In this chapter, the design and implementation of the prototype operation-shipping file system, based on Coda, are described. [1] The discussion is organized around the two main stages: logging and shipping stages. The two stages are discussed in Sections 4.1 and 4.2 respectively.

Besides, in the course of developing the prototype file system, it has been found that some important applications exhibit non-repeating side-effects, or re-execution discrepancies. Without careful thoughts, the surrogate would have had to reject any replayed user operations that use these applications. Section 4.3 shows how we can avoid rejecting naively these user operations. Furthermore, it has also been found that

[1] The source code of the prototype can be downloaded from [2].

cancellation optimization, a proven technique, may force many user operations to be value shipped. Section 4.4 explains what the issue is, and presents a new procedure of cancellation optimization that works harmoniously with operation shipping.

In this chapter, the term “a user operation” means “a Type-1 user operation” – that is, an invocation command of a non-interactive application. The term “replaying of user operation” and “re-execution of application” mean the same thing. Without further qualification, the term “client” means “weakly-connected client.”

4.1 Logging

Logging means the logging of user operations. It is the activity that happens while users are performing their high-level commands. In this section, the modeling of the problem is presented, which is followed by a discussion of the design alternatives in two aspects. The actual mechanism used in the prototype file system is presented in the last sub-section.

4.1.1 Modeling the Problem

When a user performs a user operation O, the execution of O causes a number of *file-system calls* on a Coda client. Some of the calls are *mutating*. That is, they make some changes on the state of the file system. For example, they may create a file (`creat`), modify a directory entry (`chmod`), or update a file (`write` followed by `close`). A full list of file-system calls can be found in Figure 2.1. Mutating file-system calls are mapped to *Coda updates* using an *inferred-transaction model* (Section 2.3.2). The full list of Coda updates can be found in Figure 2.3.

When the client is strongly connected, these Coda updates are written directly to the server; when the client is weakly connected or disconnected, these updates are *logged* as records in a *client modify log (CML, Section 2.3.2)* so that they can be propagated to the server later. As explained in Section 3.3, the set of records $V = \{V_1, V_2, \ldots, V_m\}$

are regarded as the value of the user operation.

Traditionally, Venus, the client cache manager, does not know about the user operation. Therefore, the *logging of the user operation* is to pass the following three pieces of knowledge to Venus:

1. O has happened,
2. O has been executed with a context C,
3. O has generated V.

With the knowledge, Venus can later choose to ship O instead of V. It needs to know about the execution context C, because O should be replayed with the same context on the surrogate.

Now, there are two design questions:

- What is the best model to pass to the file system the association of O and V?
- Who should be the *logging entity*? That is, who should communicate the above three pieces of knowledge to Venus?

Next sub-section is going to examine these two questions.

4.1.2 Design Alternatives

4.1.2.1 Associating a User Operation with a Value

Recall that Venus receives the value when it receives some mutating file-system calls. Therefore, one approach to tell Venus that a call is related to a user operation would be to add one more argument to every file system call. For example, the `write` system call interface could have been changed as shown in Figure 4.1.

This approach would work. However, it would also imply that all existing applications would have to be modified to make use of the new interface. Therefore, it is very unattractive in practice.

```
#include <unistd.h>
ssize_t write(int fd, const void *buf,
              size_t count);           /* old */
ssize_t write(int fd, const void *buf,
              size_t count, int op_id); /* new */
```

Figure 4.1: An Hypothetical Change of the System-Call Interface

Some may suggest to add one more argument `op_id`, which designate to which user operation the system call belongs.

Instead of using the above-mentioned approach, the operation-logging file system makes use the concept of *process groups* to establish the association between a user operation and the value that it generates.[2] The concept is originally used in UNIX for job control. When a shell spawns a child and executes an application, it puts the child into its own process group. The child may spawn some more children, but they will be in the same group. This way, the shell can control the whole group as one unit.

Venus has already been built in such a way that it knows some basic information about the calling process, such as the user ID, the process ID, the process group ID, etc, when it receives a file-system call. Furthermore, as described in the preceding paragraph, the shell puts all the processes created for a user operation into exactly one process group. Therefore, Venus can associate a call with a user operation by knowing the process group ID of the call.

Note that the file system identifies a user operation with its process group but not with the individual process. This is because many applications may spawn children to carry out some of the sub-tasks. For example, Figure 4.2 traces the process-creation activities of an execution of the "`make`." It shows that the execution has forked four children, each of it was for an execution of the compiler "`cc`." (The compiler in turn might fork some more children.) The whole group of processes, but not only the `make` process, are working for the requested user operation.

[2] This method is inspired by Qi Lu [24].

```
[clement@localhost oplog_applix]$ strace -e trace=process make
execve("/usr/bin/make", ["make"], [/* 24 vars */]) = 0
cc -g -I/usr/X11R6/include   -c main.c -o main.o
fork()                                  = 2401
wait4(-1, [WIFEXITED(s) && WEXITSTATUS(s) == 0], 0, NULL) = 2401
--- SIGCHLD (Child exited) ---
cc -g -I/usr/X11R6/include   -c reparent.c -o reparent.o
fork()                                  = 2405
wait4(-1, [WIFEXITED(s) && WEXITSTATUS(s) == 0], 0, NULL) = 2405
--- SIGCHLD (Child exited) ---
cc -g -I/usr/X11R6/include   -c xutils.c -o xutils.o
fork()                                  = 2409
wait4(-1, [WIFEXITED(s) && WEXITSTATUS(s) == 0], 0, NULL) = 2409
--- SIGCHLD (Child exited) ---
cc -g -I/usr/X11R6/include -o oplog_Presents main.o reparent.o
xutils.o -L/usr/X11R6/lib -lXm -lXt -lX11
fork()                                  = 2413
wait4(-1, [WIFEXITED(s) && WEXITSTATUS(s) == 0], 0, NULL) = 2413
--- SIGCHLD (Child exited) ---
_exit(0)                                = ?
```

Figure 4.2: Process-creation Activities of an Execution of the User Operation "`make`"

> This trace was produced by "`strace`," a utility for tracing system calls and signals. It shows that "`make`" forked four children in this execution.

4.1.2.2 Logging Entity

Now, the file system needs an operation-logging entity to identify progress groups to be logged and to get the details about the user operations. There are two alternatives for the entity. The first alternative is the kernel; the second alternative is an interactive shell.

The first alternative is possible because every application is executed using the `exec` system call, and all the information pertaining to the user operation are available to the kernel when the system call is invoked. This alternative has an advantage that once it is done, operation shipping can be used no matter which interactive shell the user is using. However, it has two more serious disadvantages. First, in this approach, all `exec`'ed applications are considered useful user operations. However, this position is too strong in practice, because not really all `exec`'ed applications are useful user operations. For example, for an interactive application, the application-specific commands performed with the application, but not the invocation command of the application, are useful user operations. Second, it is contrary to the philosophy that functionalities that are not absolutely needed in the kernel space should be moved to the user space. This philosophy is commonly adopted, as it helps to keep the kernel lean and allows more flexibility in designing the functionalities.

Therefore, the logging file system adopts the second alternative. That is, the logging entity should be an interactive shell, which is a user-space program. There are several advantages. First, operation logging is not a functionality that is absolutely needed in the kernel space, so it is more appropriate to be placed in the user space. Second, this approach allows the users to have more control on the scope of operation logging. When they use an operation-logging shell, all the "magical" things of operation logging and shipping will happen; when they do not want to use operation logging and shipping, they can simply switch to use an ordinary shell. Third, this approach allows a separation of *policy* from *mechanism*. The file system provides the mechanism of operation logging, and the logging shell can choose to use different policies for operation logging. For example, the logging shell may allow the user to selectively enable or disable some applications from being logged.

4.1.3 Logging Mechanism: Using the `bash` Shell as an Example

The logging mechanism is now presented. It can be described in three aspects: interface, usage, and implementation.

4.1.3.1 Interface

First of all, the file system supports operation logging by extending its interface. The extended interface is expected to be used by an operation-logging shell. It

```
VIOC_BEGIN_OP                fs_begin_op(char *command,
                                         char **args,
                                         char **env,
                                         char *cwd,
                                         mode_t umask);
VIOC_END_OP                  fs_end_op(pgid_t pgid);
```

The left column shows the two new `ioctl` commands added to support operation logging. The right column shows the two corresponding convenient functions and the arguments that they accept.

Figure 4.3: Extended Interface for Logging User Operations

comprises two new commands of the `ioctl` system call. The two new commands are `VIOC_BEGIN_OP` and `VIOC_END_OP`. They are used by the logging shell to indicate the begin and end of a *logging session.* The logging session is associated with the process group of the caller of the "begin" command. During the logging session, the effect of every mutating file-system call from the same process group seen by Venus is regarded as a part of the value. Information about the user operation and the execution context are also passed to file system with the "begin" command. Figure 4.3 shows the extended interface. For each command, a convenient function is provided. It simplifies the task of packing various arguments needed by the command, and is implemented as a static library routine to be linked into the logging shell.

4.1.3.2 Usage

How should a logging shell make use of the extended interface? Let us first examine how a Unix shell executes an application-invocation command.

Figure 4.4a shows the pseudo code of a Unix shell dealing with the execution. Figure 4.5a depicts the execution in graphical form. When the shell recognizes the application-invocation command O, it `fork`'s a copy of itself. On the child side, the forked child creates a new process group itself by making use of the `setpgid` system call, and then it `exec`'s the application. The application is thus executed, and it may cause some mutating file-system calls $V_1, V_2, \ldots, V_m$ while executing. On the parent side, The parent waits for the termination of the child by using the `waitpid` system call, and proceeds for other tasks when the child terminates.

Now, to log the user operation, the shell makes use of the two new `ioctl` commands. First, the forked child issues `VIOC_BEGIN_OP` after the `setpgid` call and before it `exec`'s the application. Second, the parent issues the `VIOC_END_OP` command when the forked child terminates. Figure 4.4b shows the pseudo code, and Figure 4.5b depicts the scenario. These two commands thus delineate the logging session for the user operation.

In this project, a popular Unix shell – the GNU Project's Bourne Again Shell (`bash`) [10] – is extended to add the operation-logging capability. The source code of `bash` is publicly available, and my extension is based on version 1.14.7. The modification involves only a few lines of code. The source code of the extended `bash` shell is available for download from [2]. The current version implements the most straightforward policy, under which all user operations are logged. A future version may implement a more flexible policy.

4.1.3.3 Implementation

In this sub-section, several implementation details are discussed. First, the activities of the logging file system during a logging session are explained. Second, the data structures for logging user operations are described.

Logging Session. Recall from Section 2.4.2 that a Coda volume has three different states: hoarding, emulation, and write-disconnection. When the Venus is in a logging session, all volumes that are being carried by Venus are put into the write-disconnection state. All volumes are affected since Venus does not know yet which volumes will be mutated by the coming file-system calls of the user operation. At the end of the logging session, these volumes may transit to other states using the existing state-transition rules. For example, a volume may return to the hoarding state when all of the following three conditions are met: (1) there is no on-going logging session, (2)

```
...
if ( (pid=fork()) == 0 ) { /* child */
    setpgid(0,0);           /* put myself in a pgrp */
    execve(pathname, argv, envp); /* exec the app. */
} else if ( pid > 0 ) {     /* parent */
    waitpid(pid, &status, option);
} ...
...
```

(a)

```
...
if ( (pid=fork()) == 0 ) { /* child */
    setpgid(0,0);           /* put myself in a pgrp */
    fs_begin_op(pathname, argv, envp, cwd, umask);
                            /* start the logging session */
    execve(pathname, argv, envp); /* exec the app. */
} else if ( pid > 0 ) {     /* parent */
    waitpid(pid, &status, option);
    fs_end_op(pid);         /* end the logging session */
} ...
...
```

(b)

Figure 4.4: Pseudo Code Showing the Sequence of Executing an Application

(a) and (b) are the pseudo code for an ordinary and an logging shell respectively. The application being executed is specified by `pathname`. The following are the meanings of the other arguments. `argv` is the command-line argument list, `envp` is the environment-variable list, `cwd` is the current working directory, `umask` is the file-creation mask, `status` is used for getting the exit status of the terminated child, and `option` specifies the exact behavior of the `waitpid` system call.

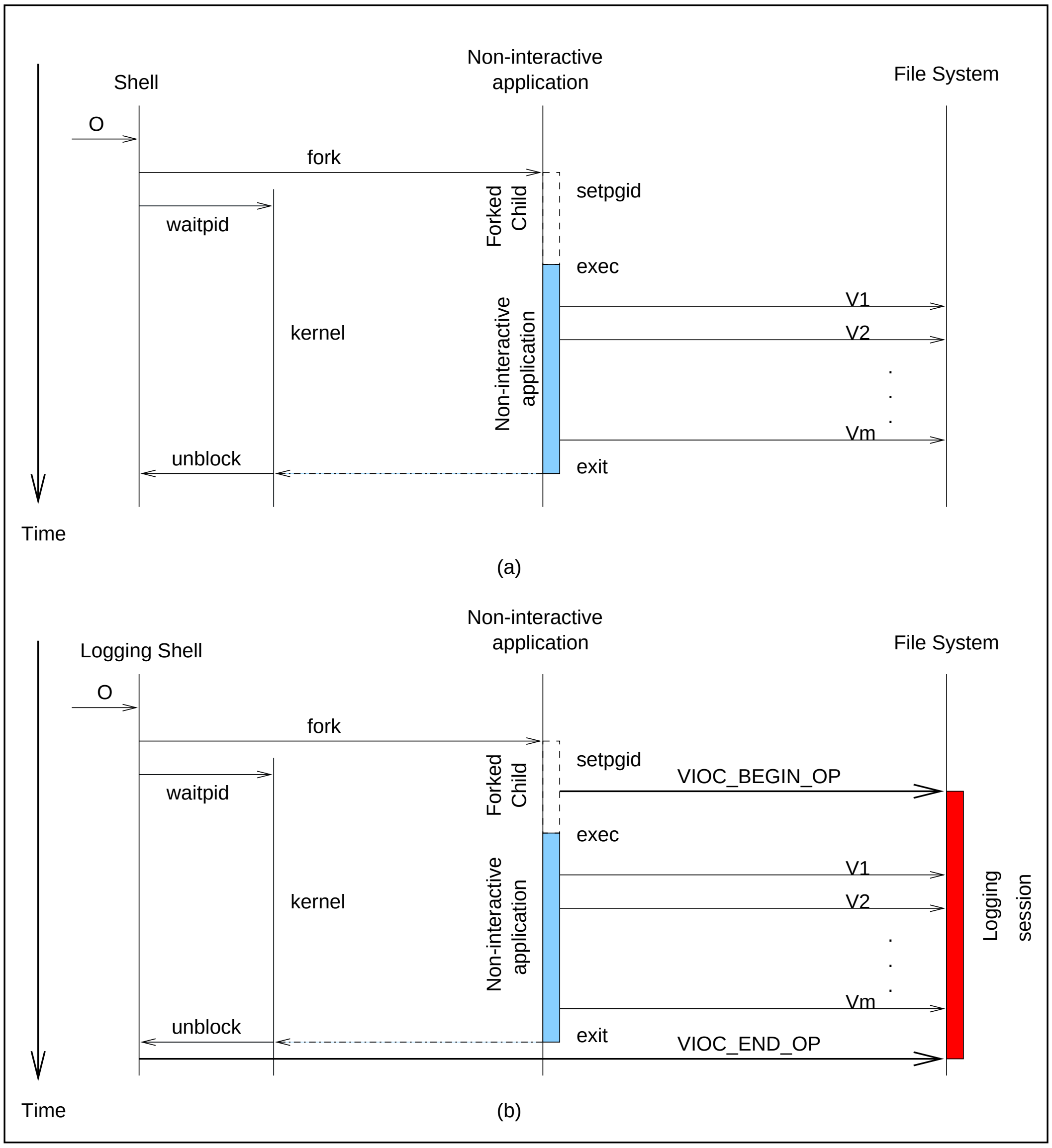

Figure 4.5: Logging of User Operations

(a) The typical sequence of a shell executing an application. (b) A logging shell logs the user operation by inserting the `VIOC_BEGIN_OP` and `VIOC_END_OP` commands before and after the execution of the non-interactive application. $V_1, V_2, \ldots, V_m$ are the effects of the mutating file-system calls that the application generates.

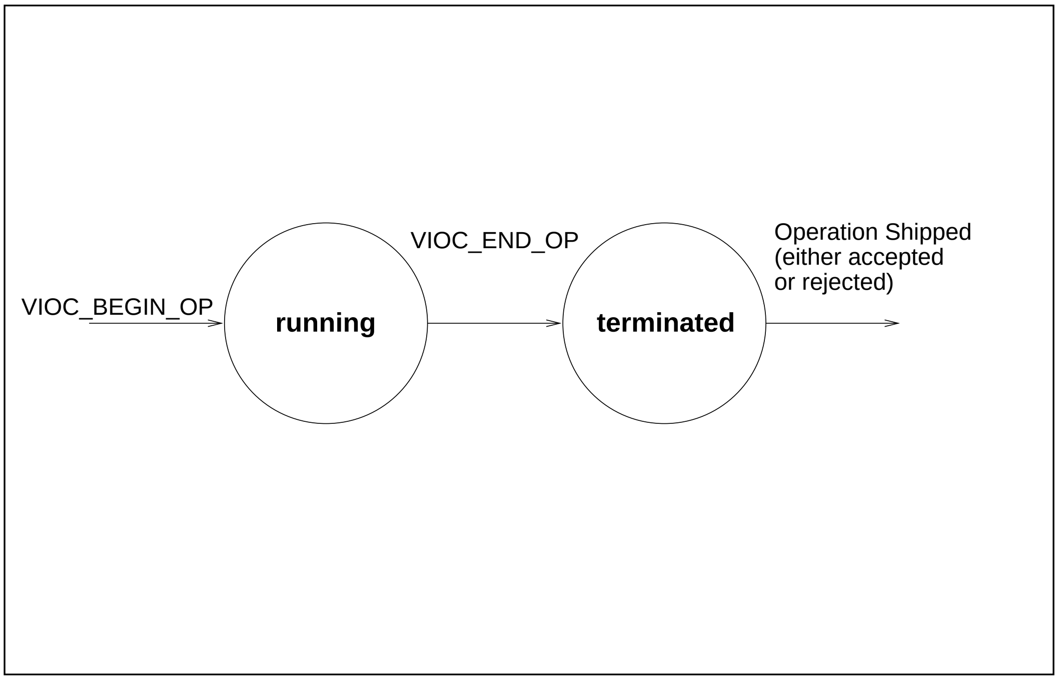

Figure 4.6: Two Possible States of User Operations

there is a strong physical network connection, and (3) all log records on the CML have been reintegrated with the server.

A user operation has two different states. When its logging session is started by a `VIOC_BEGIN_OP` command, it is in the *running* state. When its logging session is ended by a `VIOC_END_OP` command, it is in the *terminated* state. Information about a user operation is kept in the logging file system until the operation is shipped for propagation. The file system attempts at most once operation-based update propagation per user operation. If the attempt is rejected by the surrogate, the file system will fall back to value shipping, so it will no longer need the information about the user operation. If the attempt is accepted by the surrogate, it will not need the information either. Figure 4.6 shows the various transitions of the states.

The CML records of a user operation are eligible for update propagation only after the operation is in the terminated state. This is because the set of CML records is considered an indivisible unit when attempting operation shipping, and the file system can know about all the records for a given user operation only when the operation has terminated.

An erroneous shell may forget to end a logging session after starting it. However, Venus does not allow any logging sessions to stay forever in the running state, otherwise the records of the session will block all other records in the same CML from being reintegrated. All logging session is bounded by a parameter `UserOpTimeBound`. A periodic daemon proactively ends all logging sessions that have exceeded the time bound.[3] The default value of the time bound is 600 seconds.

The Structure of the Operation Log. User operations and the file-system updates that they generate are concepts at two different level of abstract. File-system updates are logged in temporal order in the CML (Section 2.3.2). So, how should user operations be logged?

One may attempt to add another log for user operation records. However, doing so has two problems. First, Venus then needs to synchronize the two logs, since some updates are shipped by value and some by operation. Second, this structure makes it difficult to merge the operation-shipping mechanism into the existing value-shipping mechanism.

A better solution is to *augment* CMLs so that they also carry the information about user operations. Specifically, it is to *tag* CML records with the user operation, if any, that causes them. Figure 4.7 shows a high-level view of the new CML structure. Note that CML is a per-volume log. In the figure, it can be seen that records of different user operations and records of no user operation can be kept in the same log.

Records of different CMLs, that is, records that mutate different volumes, may be tagged with the same user operations. However, in the current implementation, they will not be operation shipped but only be value shipped. There are two reasons for this design decision. First, the existing reintegration mechanism is done on a per-volume

[3]This daemon is not yet implemented in the current version of the prototype, but it will be implemented by the next release.

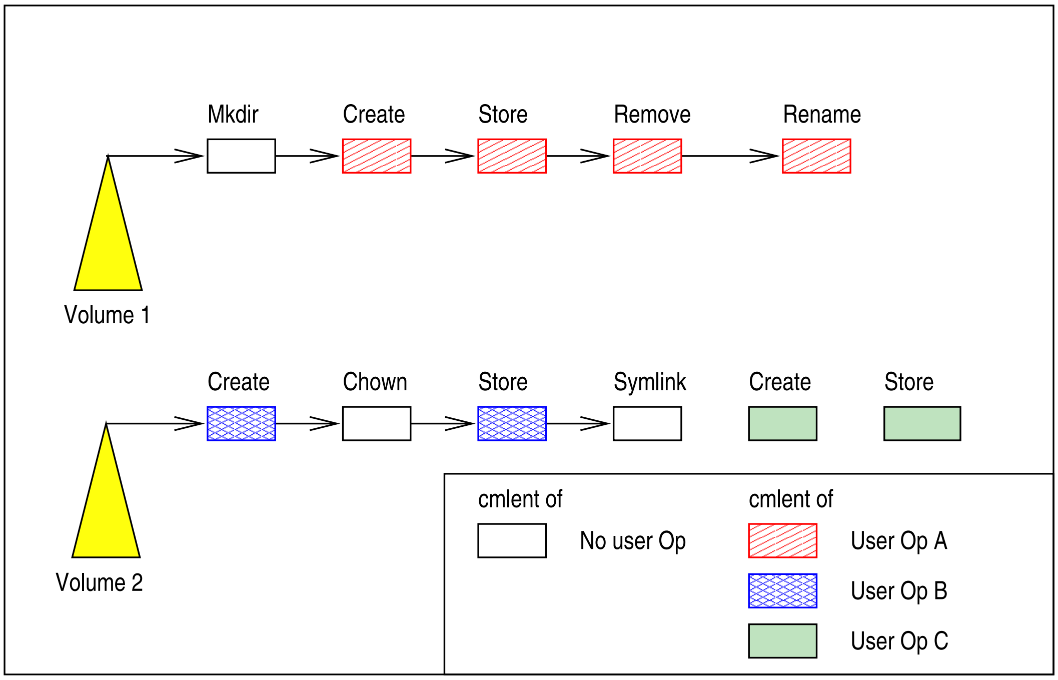

Figure 4.7: High-level View of the New CML Structure

This figure illustrates two client-modify logs of two volumes. Note that some records in the logs are tagged with user-operation information, some are not tagged.

basis. Operation shipping records of different CMLs simultaneously amounts to a multi-volume reintegration, and it is possible only after an overhaul of the reintegration mechanism. Second, the reintegration of a CML involves locking the volume involved on the server. If multiple volumes are involved, then the server need to deal with a number of complicated issues such as deadlocks. The design of the logging file system favors simplicity. Complicated mechanism is used only when it is fully justified. So far the need for supporting the shipping of multi-volume user operations is not eminent. This is true because most users' tasks are done within a volume.

Figure 4.8 shows the detailed structure of the augmented CML. Records that are generated by a known user operation are tagged with non-zero value in the fields `vbt` and `userOpId`. The two fields are used as indices for the User Operation Database (OPDB), which will be explained below. The field `userOpId` is the process group

```
ClientModiyLog *log;
rec_dlink       handle;

ViceStoreId     sid;
Date_t          time;
UserId          uid;
int             tid;
CmlFlags        flags;

time_t          vbt;            /* for operation shipping */
pid_t           userOpId;       /* for operation shipping */
unsigned        srgRejected:1;  /* for operation shipping */
unsigned        ghost:1;        /* for operation shipping */

< type specific fields >

dlist  *fid_bindings;
dlist  *pred;
dlist  *succ;
```

Figure 4.8: Type-independent Fields of Augmented CML Records

This figure shows how the CML record is augmented for supporting operation shipping. Four more fields are needed: `vbt` (Venus Birth Time), `userOpId`, `srgRejected` (this operation was rejected before by the surrogate), and `ghost` (ghost record). The other fields are already explained in Figure 2.4.

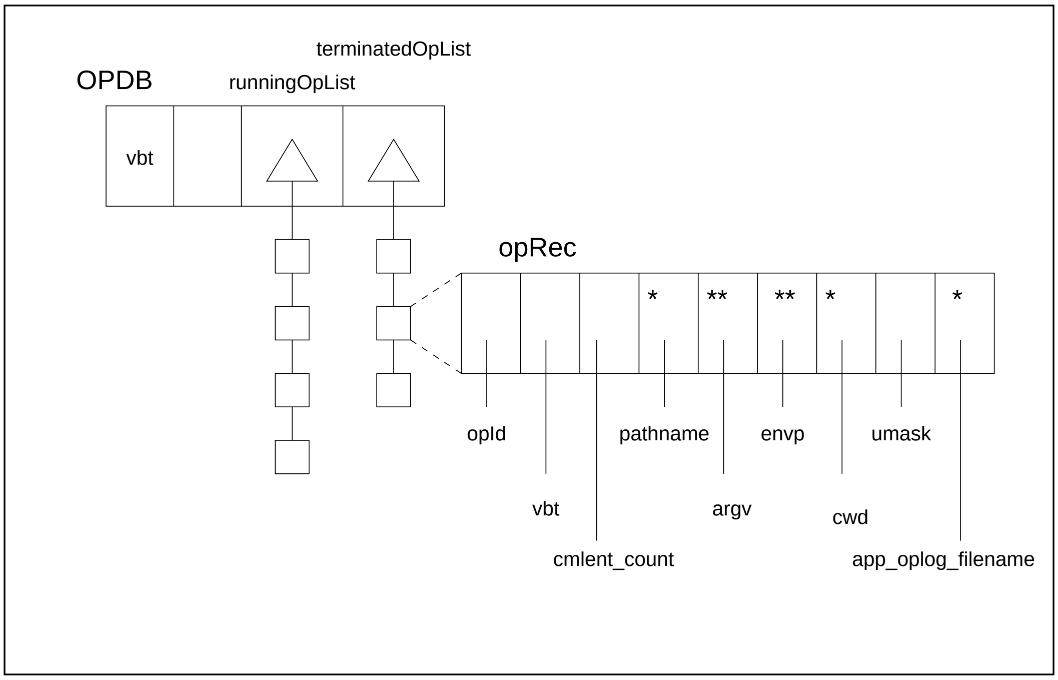

Figure 4.9: User Operation Database and User Operation Records

This figure illustrates the User Operation Database (OPDB) and the User Operation Records (OpRec). Every entry in the two list of the OPDB is a OpRec. In the OpRec, a field with a asterisk (*) is a field that needs a separate string for physical storage; a field with two asterisks (**) is a field that needs a separate array of string for physical storage.

ID of the user operation. The field `vbt` is the "Venus Birth Time." It is needed since different runs of Venus may see different process groups using the same process group ID (the machine recycle the process group ID after reboot). A zero value on both the two fields indicates that there is no user operation information provided for the record. The field `srgRejected` indicates whether or not the user operation was rejected by the surrogate before. The field `ghost` is to be explained in Section 4.4.2.

Figure 4.9 shows the structure of the User Operation Database (OPDB). The database contains two lists of User Operation Records (opRec), one for the running operations, and the other for the terminated operations. A newly created operation record is added to the running list. It is later transferred to the terminated list when its logging session terminates. It is discarded from the terminated list when the user operation is shipped. Each record stores the relevant information for replaying the user operation. The field `app_oplog_filename` is needed for an *application-specific* operation log, and is only for application-aware operation shipping (Section 5.2.1).

4.2 Shipping

The main stage of operation-based update propagation is the shipping stage. It is integrated into the existing reintegration mechanism. That is, when Venus is triggered to start a reintegration, it selects some records from the prefix of the CML. Depending on whether the records that it selects are associated with or without user operation information, it proceeds with operation shipping or value shipping respectively. Operation shipping is performed in a best-effort manner, so if some records cannot be operation shipped, they will just be value shipped in the next round of the reintegration.

There are three parties involved in the shipping stage: the client, the surrogate, and the server. Figure 4.10 shows an overview of the interaction between the three parties. It also shows that the shipping stage comprises five phases: (1) requesting, (2) replaying, (3) validation, (4) reintegration/aborting, and (5) finalization.

The client performs the requesting phases. The surrogate performs the replaying, validation, and reintegration/aborting phases. The server is also involved in the reintegration/aborting phases. Both the surrogate and the client need to perform some duties in the finalization phase. The five phases are discussed in five sub-sections following the next sub-section, which presents the problem model.

This section begins with a discussion on the modeling of the problem. It then describes the five phases of the shipping stage in the next five sub-sections.

4.2.1 Modeling the Problem

Recall from Section 4.1 that Venus on a weakly connected client gets the knowledge of O (a user operation), C (the execution context of O), and an association that a number of records in the CML $V_1, V_2, \ldots, V_m$ are resulted from the execution of O. To complete the background processing of those file-system calls that generate the records, Venus must eventually ship the records to the server for reintegration (Recall from Section 2.1.1 that the server keeps the primary replicas of file system objects, and the cached copies on the client are only secondary replicas.)

However, shipping the records to the server using the weak network is an expensive operation. It imposes too much traffic for the network, and incur a long latency for finishing the reintegration procedure. Therefore, Venus attempts to send O to a surrogate for a re-execution of O, under the context C, which re-generates a set of CML records $V_1', V_2', \ldots, V_m'$. It hopes that the re-generated set of records has the property that $V_i = V_i', \forall i \in \{1, 2, \ldots, m\}$. If this can be done, then it is no different whether the server receives V_i from the client, or V_i' from the surrogate. The surrogate is strongly connected to the server, so the reintegration of V_i' from the surrogate can be done easily: the impact on the network is small, and the latency for the reintegration procedure is low.

4.2.2 Requesting

A record in the CML is *eligible* for operation shipping if it is associated with a user operation, and neither of the following two *breaking conditions* happen to the user operation: (1) the user operation has been rejected before by the surrogate, (2) the records of the user operation are interleaved by some records that are not associated with the same user operation. A record that is not eligible for operation shipping can only be value shipped. The first condition ensures that operation shipping is attempted at most once. The second condition ensures that records are always shipped in temporal order.

Note that Venus uses a static policy regarding operation shipping. That is, an eligible user operation is always operation shipped regardless of the bandwidth

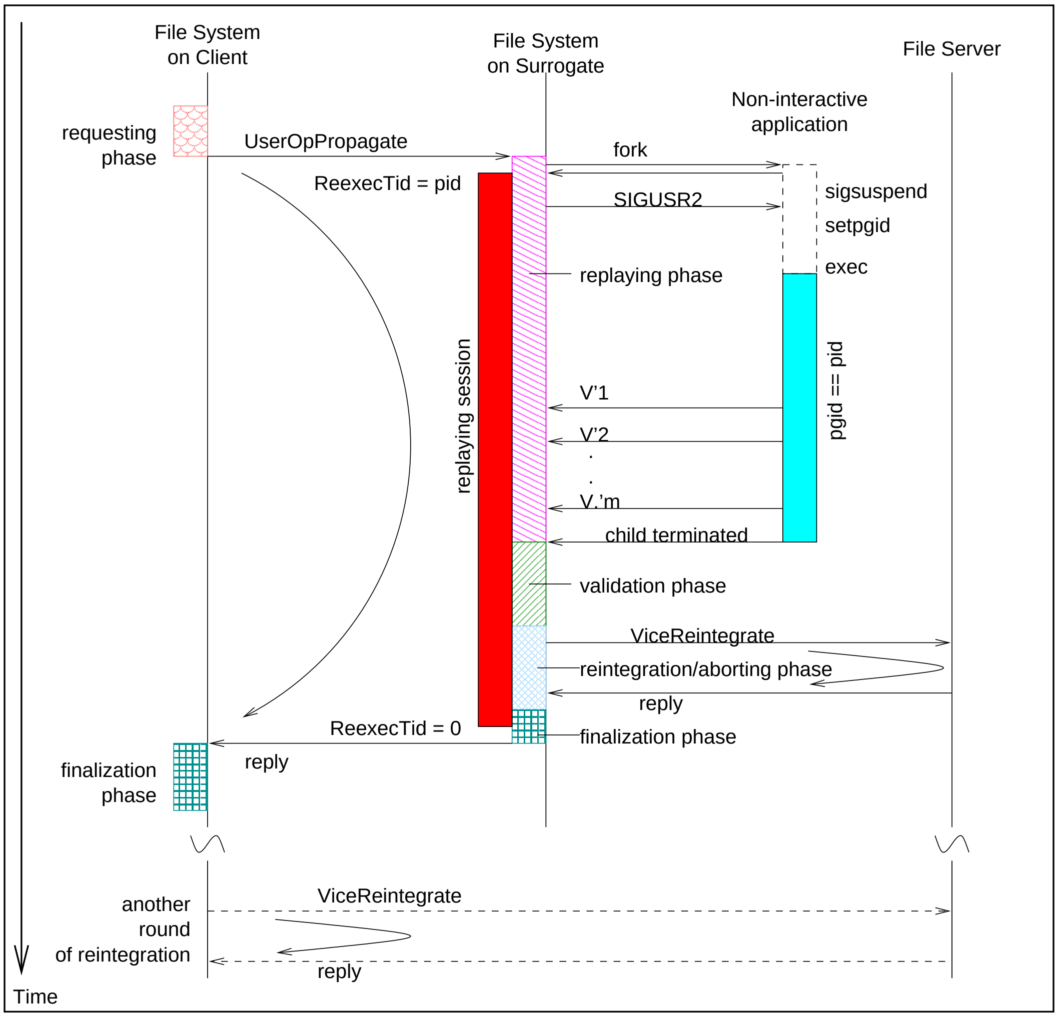

Figure 4.10: Shipping Stage

This figure shows the five phases of operation shipping: requesting, replaying, validation, reintegration/aborting, and finalization. It also shows how the replaying session spans across the last four of the above five phases. Shown in the lower part of figure is the fallback mechanism, which is needed only when the shipping of a user operation is not successful.

condition. In Section 8.2.3 we will see that another approach is that Venus makes a *dynamic decision* regarding whether an eligible user operation should be propagated using operation or value shipping.

The shipping stage for operation shipping is an integrated part of the existing reintegration mechanism. It is implemented in a procedure called `IncReintegrateViaSurrogate`, and is an addition to the two existing value-shipping procedures: reintegration by chunks (`IncReintegrate`) and reintegration by fragments (`PartialReintegrate`) (Section 2.4.5).

Reintegration is triggered by some triggering events, such as the referencing of an object in the volume by either a worker thread or the hoarding thread, or the periodic volume daemon detects that the volume is pending for reintegration. Once triggered, Venus will dispatch a *reintegrator* thread to oversee the reintegration process. Reintegration proceeds in rounds. In each round, the reintegrator decides which procedures to use depending on what records it can select from the prefix of the CML. Note that the records are always selected for reintegration in temporal order.

If the first record is eligible for operation shipping, then the reintegrator checks to see if the user operation is *ready* for shipping. A user operation is ready for shipping when both of the following two conditions are true: (1) the user operation is in the terminated state, and (2) all the records of the user operation have an age larger than the age window. The first condition ensures that all records of a logging session will be shipped in the same attempt of operation shipping so that the validation phase will have all the needed information (Section 4.2.4); the second condition preserves the effectiveness of cancellation optimization (Section 2.3.2.2).

If the user operation is ready, the reintegrator will proceed to use operation shipping; if not, the reintegrator will terminate the current round of reintegration, awaiting both of the two conditions to become true. The awaiting period is bound, since the first condition is bound by the parameter `UserOpTimeBound`, and the second condition is bound by the age window.

If the first record is not eligible for operation shipping, then the reintegrator will proceed to use value shipping for the current round. Depending on the size of the first record, it may proceeds with reintegration by chunks or reintegration by fragments. That is, if the first record exceeds the chunk size, it will be reintegrated by fragments; otherwise, all the subsequent non-eligible records fitting into a chunk will be selected for the current round, and they will be reintegrated by chunk.

The requesting phase starts when the reintegrator decides to use operation shipping and invokes `IncReintegrateViaSurrogate`. There are three main steps in this phases:

- **Permanent-Fid Allocation** - If some newly created objects are having locally generated temporary fids (file identifiers), the client needs to translate them permanent fids, which are generated by the server and are globally unique The need of this step is explained in Section 2.3.3. Note that the client needs to communicate with the server using the `ViceAllocFids` RPC to allocate the needed permanent fids.

- **Preparing information for replay** - Five groups of information are needed for the replay of the user operation:

  – Information about the user operation and the execution context: the pathname of the invoked application, the command-line arguments, the environment variables, the current working directory, and the file-creation mask.

  – The identifier of the user who has made this operation, and the identifier of the volume in which the mutating file-system calls have happened.

  – Checksum information. For each `store` record, the reintegrator need to compute two types of checksum information: a MD-5 fingerprint, and a Reed-Solomon forward-error-correction code. The former is for validating

```
UserOpPropagate  (IN RPC2_CountedBS     packedOpRec,
                  IN RPC2_Integer       buf2size,
                  OUT ViceVersionVector updateSet);
                  IN OUT SE_Descriptor  BD);
```

Figure 4.11: RPC Interface for Operation Shipping

Most arguments needed for the RPC are packed into `packedOpRec`. The reference CML and the application-specific operation log are sent using the RPC2 side-effect mechanism [47]. Their total size is `buf2size`.

the new contents of the re-generated `store` record, and the latter is for fixing any minor re-execution discrepancies possible. The need of them are explained in Sections 4.2.4 and 4.3.1.3 respectively.

- Reference CML. The thin part of all CML records of the user operation is packed as the *reference CML*. Recall from Section 2.3.2.1 that the thin part of a CML record includes everything of the CML except the contents of the container file, if any. The reference CML is used in the validation phase (Section 4.2.4).
- Application-specific Operation Log. This is needed only for application-aware operation shipping. It is to be discussed in Section 5.2.1.

The first three groups are packed into a RPC argument called `packedOpRec`. The last two groups are packed into an in-memory buffer call `buf2`, which is to be retrieved by the surrogate using the RPC2 side-effect mechanism [47].

- **Invoking the `UserOpPropagate` RPC.** Finally, the reintegrator sends the RPC request to the surrogate. The interface of the RPC is shown in Figure 4.11.

4.2.3 Replaying

On the surrogate, the receiving of the `UserOpPropagate` marks the beginning of the replaying, validation, reintegration/aborting, and finalization phases. The four phases

```
...
if ( (pid=fork()) > 0 ) {     /* parent */
    put the volume in write-disconnected mode;
    disable the volume from reintegration and ASR;
    entering the volume as a mutator;
    ReexecTid = pid;          /* replay session begins */
    kill(pid, SIGUSR2);       /* child may proceed now */
    wait for the child to finish, but not indefinitely;
} else if ( pid==0 )    {     /* child */
    prepare childmask so that only SIGUSR2 is unmasked;
    sigsuspend(&childmask);  /* do not proceed until parent
                              * is in replay session (SIGUSR2) */
    chdir(cwd);               /* restore current working dir */
    umask(_umask);            /* restore file-
creation mask */
    setuid(user);             /* restore effective user id */
    setpgid(0,0);             /* put myself in a process group */
    execve(pathname, argv, envp); /* really executing the app. */
}
...
```

Figure 4.12: Pseudo code of the Re-execution of an Non-interactive Application by Venus

This pseudo code shows how Venus `fork`'s and `exec`'s the re-executor, and how they are synchronized. The re-executor uses the parameters logged by the weakly-connected client for restoring the re-execution context and re-execute the application. These parameters are `cwd`, `_umask`, `user`, `pathname`, `argv`, and `envp`.

are all controlled by a *surrogate-server thread* in Venus.

In the replaying phase, there are three preparatory steps before the user operation is replayed:

- **Entering** - The surrogate checks to make sure that Venus is not already engaged in other replaying activities. If Venus is already engaged, it will refuses to participate in replaying, and will reply an error to the client (`ESURR_FAIL`).
- **Retrieving information for replay** - The surrogate retrieves the information needed for the replay of the user operation. The RPC argument `packedOpRec` is unpacked, and `buf2` is retrieved from the client using the RPC2 side-effect mechanism.

- **Token checking** - The user performing the user operation must have a valid authentication token on the surrogate machine. Otherwise, the surrogate will refuses to participate in replaying, and it will reply an error to the client (`ESURR_NOTOKEN`).

- **Replaying the user operation** - Since the user operation is an invocation command of a non-interactive application, the replaying of the user operation is the re-execution of the application. Venus spawns a child process, called the *re-executor*, which re-executes the application. The pseudo code of this step is shown in Figure 4.12. There are several remarks about this step:

  - **Write-Disconnecting the volume.** Although most re-executions are anticipated to be successful (execute completely and pass the validation), the file system must prepare for the possibility that they may fail (cannot execute completely or fail the validation). Therefore, re-executions must be *abortable transactions* such that failed re-executions will have no lasting effect. This is done by putting the affected volume in *write-disconnected* mode during the re-executions. When a volume is in the write disconnected mode, input files can be retrieved from the server, but file-system updates are not written immediate through to the server. The volume will be re-connected to the server in the finalization phase.

  - **Disabling the volume from reintegration and application-specific resolver.** There may be some other concurrent activities going on in Venus. Two of such activities – reintegration and application-specific resolver (ASR) [21, 20] – may interfere with the activities of the replaying phases, so they are disabled. These activities will be re-enabled in the finalization phase.

  - **Entering the volume.** The surrogate makes the user "enter" the volume as a "mutator." This means that the user will acquire a write lock of the volume, after a possible duration of being blocked awaiting other active users to "leave" the volume. Note that it is the user, but not the surrogate-server thread, who enters the volume. This distinction is important because the same user can have multiple threads active in the volume. The user will leave the volume in the finalization phase.

- **Marking the beginning of a replay session.** The surrogate marks Venus to be in a *replaying session* by setting the global variable `ReexecTid` to be the process group ID of the re-executor. The replaying session will be ended in the finalization phase.

  When in a replaying session, whenever Venus receives a mutating file-system call, it will examines the process group ID of the caller. If the ID equal to the value of `ReexecTid`, the call is an effect of the replayed user operation, and the effect should be tagged with the replaying session.

- **Synchronization between the parent and the child.** The re-executor should proceed to re-executing the application only after Venus has begun the replaying session. Venus can begin the replaying session only after `fork`, since it knows the process group ID of the re-executor only after that. However, `fork` does not guarantee whether the parent process (Venus) or the child process (the re-executor) runs first. Therefore, an explicit synchronization is needed between the two processes. This is done by the following arrangement. Immediately after `fork`, the re-executor suspends itself awaiting a signal from Venus. Venus wakes up the re-executor by sending it a signal after it has begun the session.

- **Waiting for the re-executor to finish.** Venus must wait for the termination of the re-executor before it can proceed to the next validation phase. It uses a loop combined with the non-blocking `waitpid` system call to poll for the termination of the re-executor.

The waiting period is bounded by a parameter. Venus cannot wait indefinitely since the re-executor may erroneously involve in an infinite loop. Venus will kill the re-executor if it runs for too long. The parameter is a constant in the current implementation, but it may be changed to a client-specified variable in the future.

– **Restoring of execution context.** As shown in the pseudo code, the re-executor restores the execution context before really executing the application. The current working directory is set by using the `chdir` system call, the file-creation mask is set by using the `umask` system call, the environment-variable list and the command-line argument list are set when invoking the `execve` system call, and the user identity is set by using the `setuid` system call. The re-executor also puts itself into its own process group by using the `setpgid` system call.

4.2.4 Validation

In the validation phase, the portion of CML that is resulted from the replaying of the user operation (the new CML) is *adjusted* and *checked* against the reference CML. The surrogate adjusts the status information of the records in the new CML to their counterparts in the reference CML. It also checks that the records in the new CML is identical to their counterparts in the reference CML. The two tasks are performed in one routine (AdjustAndCheck). The return value of the routine indicates whether the replayed user operation is *accepted* or *rejected*. The former indicates that the new CML is identical to the reference CML, so the surrogate can reintegrate the new CML with the server in the next phase. The latter indicates that the two CMLs are different, so the surrogate should abort the new CML in the next phase. The routine contains a loop in which CML records are adjusted and checked one by one, but before entering the loop, the surrogate performs the following two tasks:

1. **Marking the records of the new CML and counting their number.** Records of the new CML are tagged with a common reintegration identifier, so that they can be atomically committed or aborted in the next reintegration/aborting phase. The number of the records in the new CML is also counted. If the number does not match with its counterpart of the reference CML, the surrogate rejects the replayed user operation immediately, as the two CMLs will never be the same.

2. **Temporary Filename Renaming.** This is a step to fix any possible non-repeating side effects. It will be discussed in more detail in Section 4.3.2.

After that, the routine will enter the main loop. For each record, the following three steps are performed:

1. **Locating the corresponding reference CML record.** The corresponding record in the reference CML is located. In the following discussion, the term "two records" refers to the record in the new CML and the corresponding record in the reference CML.

2. **Checking the contents.** Except the status information that will be adjusted in the next step, the contents of the record is compared to that of the reference record. For example, for a `chown` record, the value of the `owner` field of the two records are compared. Any mis-matches in the comparison cause the user operation to be rejected.

   However, there is one important exception for the container file of the `store` record. For this field, the MD-5 fingerprints of the two records are compared. Recall that the client does not ship the container files of the `store` records, but only their fingerprints, to the surrogate. Otherwise, the network traffic required will be amounting to value shipping.

   If the fingerprints of the two records differ, a naive approach would be to reject the replayed user operation immediately. However, the re-generated file of the replayed user operation may already be very similar to the original file. They may be so similar that the former can be transformed to the latter by using a procedure of error correction. Therefore, instead of rejecting the replayed user operation immediately, the surrogate will make an attempt to error-correct the re-generated file using the parity blocks sent in by the surrogate. The details of this error-correction procedure will be given in Section 4.3.1. Another fingerprint comparison will follow the error-correction procedure to see if the procedure succeeds in correcting the re-generated file.

3. **Adjusting status information.** The following three fields on the two records will always differ, because they are generated in a time-dependent manner. The value of the new CML record is restored to take the value of the reference record.

   (a) **Modification time.** The local system time when the mutation happens.

   (b) **The fids of child objects in the three object-creation records:** `create`**,** `mkdir`**,** `symlink`**.** Note that the fids will appear in both the CML record and the file-system object descriptors of the created object.

   (c) **Store ID.** It is the ordered pair `<host,uniquifier>`. It is globally unique and is locally generated by Venus at the time of local mutation. It is used for concurrency control by Coda (Sections 2.2.3 and 2.3.3).

4.2.5 Reintegration/Aborting

After having made a conclusion on whether the replayed user operation is accepted or rejected in the validation phase, the surrogate proceeds to the reintegration/aborting phase. The two possible outcomes are handled differently: the surrogate reintegrates the new CML with the servers if the operation is accepted, but it aborts the current attempt of operation shipping if the operation is rejected. The details are provided in the following.

Reintegration If the replayed user operation is accepted, the surrogate will proceed to reintegrate the new CML with the servers on behalf of the client. Like an ordinary reintegration, the surrogate reintegrates the new CML with all servers in the volume's AVSG (accessible volume storage group, Section 2.2.1). It also collates the responses from the servers, and construct an *update set* (Section 2.2.4), which is used by both the servers and the surrogate for keeping track of which servers have performed the reintegration successfully. Note, however, that Venus on the client also needs the update set for keeping track the same information. Therefore, the surrogate will forward the update set to the client in the reply of the `UserOpPropagate` RPC in the subsequent finalization phase.

Aborting If the replayed user operation is rejected, the surrogate will proceed to abort the current attempt of operation shipping. This includes discarding all the records in the new CML and discarding from the local cache all the file-system objects that have been mutated by the rejected user operation. Note that the surrogate does not need to undo any mutating operations that have performed on these objects, because it can fetch the unmodified versions of them from the server when these objects are next referenced (remember that the surrogate replays user operations in the write-disconnected mode). After the aborting of the current attempt of operation shipping, the rejected user operation does not have any lasting effects.

4.2.6 Finalization

After the reintegration/aborting phase, the surrogate enters into the finalization phase. Some steps of this phase are performed on the surrogate, and some other steps are performed on the client. The surrogate performs the following steps:

```
ESURR_TIMEDOUT         /* no reply from the surrogate */
ESURR_REEXEC_REJECTED  /* re-exec did not produce same mutations */
ESURR_INVAL            /* invalid value specified */
ESURR_FAIL             /* re-execution failed */
ESURR_NOTOKEN          /* user has no token on the surrogate */
ESURR_REEXEC_TIMEDOUT  /* replaying process took too much time */
```

Figure 4.13: Error Returns of the `UserOpPropagate` RPC

- **Marking the end of the replaying session.** The global variable `ReexecTid` is reset to null, indicating the end of the replaying session.
- **Re-connecting the volume to the server.**
- **Re-enabling the volume for reintegration and application-specific resolver.**
- **Leaving the volume.** The surrogate make the user leave the volume by releasing the write lock that it holds.
- **Replying the `UserOpPropagate` RPC.**

The reply of the `UserOpPropagate` RPC starts the finalization phase on the client. The actions taken by the client depends on the result of the RPC:

1. **The RPC reply indicates a success.** This happens when all phases of the operation shipping stage have completed successfully. In this case, the client locally commits the shipping stage. Local commitment includes updating objects' states, such as version vectors, clearing the dirty bits for objects, and discarding all CML records of this round.
2. **The RPC reply indicates an failure, or there is no reply.** An error in any phase of the shipping stage causes the stage to fail. Figure 4.13 lists the various possible error returns of the RPC. The client thus sets the flag `SrgRejected` for all the CML records of the current round. These records will therefore be not eligible for operation shipping – they will be value shipped in the next round of reintegration.

It is possible that a reintegration completes successfully at the servers, but the response of the `UserOpPropagate` fails to arrive at the client in spite of retransmission. This can happen when there is an untimely failure of the surrogate or the communication channels. The existing Coda mechanism of ensuring atomicity of reintegrations is used here to handle this kind of failures [30, page 87], and it proceeds as follows. The client presumes a reintegration has failed if it does not receive a successful response from the surrogate, and it will retry the reintegration. At the same time, the server retains the information necessary to detect whether a record has already been reintegrated earlier. If a client attempts such an improper retry, the server will reply with the error code `EALREADY` (operation already in progress). From the error code, the client knows that the records have already been successfully reintegrated in a previous attempt, and it will simply locally commit the records.

4.3 Non-repeating Side Effects

In an early stage of this project, we expected the re-executions of most non-interactive applications can produce an identical result as the original execution. We had such an expectation because we are focusing on applications that perform deterministic tasks, such as compiling binaries or formatting texts. It was also because we had excluded applications such as games and probabilistic search that are randomized in nature. However, to our surprise, many common applications exhibit *non-repeating side effects*. They do not produce an identical result on re-execution, instead, their results are different to the originals in some non-critical ways. Non-repeating side effects are also called *re-execution discrepancies* in this thesis.

So far two types of such side effects have been observed:

Applications that put time stamps in output files	
rp2gen	stub generator for use with the RPC2 package
ar	software library archive builder
latex	text formatter and typesetter
Applications that use time-dependent names for temporary files	
ar	software library archive builder

Figure 4.14: Applications that Exhibit Non-repeating Side Effects

Different applications that exhibit non-repeating side effects.

1. time stamps in output files
2. time-dependent names for temporary files

Figure 4.14 lists the applications that are known to exhibit these side effects. A naive approach is to reject the re-execution of these applications. Unfortunately, this implies that many common applications cannot be used with operation shipping, and severely limits the usefulness of operation shipping. A better approach is to find some ways to suppress the side effects. The following two sub-sections explain the handling of the two side effects.

However, note that not all side effects can be suppressed. For example, if an application has a side effect outside of the system, such as sending an email message during each execution, then its side effects probably cannot be suppressed. In this case, the best solution is to let the users to decide whether they still want to use the operation shipping application given the non-repeating side effects. This can be done by the policy module of the logging shell. It can allow the users to selectively enable or disable applications for logging.

```
0000000 367 002 001 203 222 300 034   ;  \0  \0  \0  \0 003 350 033
0000020   T   e   X       o   u   t   p   u   t       1   9   9   9   .
0000040   1   2   .   0   2   :   0   7   0   2 213  \0  \0  \0 001  \0
0000060  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
```

(a)

```
0000000 367 002 001 203 222 300 034   ;  \0  \0  \0  \0 003 350 033
0000020   T   e   X       o   u   t   p   u   t       1   9   9   9   .
0000040   1   2   .   0   2   :   0   7   0   3 213  \0  \0  \0 001  \0
0000060  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
```

(b)

Figure 4.15: Dumping Two DVI files in Octal Format

The "octal dump" of the first 64 bytes of the output DVI files of two different executions of the LaTeX program. Each line shows 16 bytes, displayed in either a printable form if the byte is printable, in its ASCII code in octal if the byte is not printable, or as \0 if its ASCII code is zero. The leftmost column is the offset of the first byte in the line. Note that two time-stamp strings "`1999 12 02:0702`" and "`1999 12 02:0703`" appear in the two files respectively (located on 27th to 41th bytes).

4.3.1 Side Effects Due to Time Stamps

4.3.1.1 Time Stamps

LaTeX, a text formatter, for example, puts time stamps in the output file. Figure 4.15 shows the "octal dump" of the first 64 bytes of two output files, which are produced by two executions of the LaTeX program. The output files are in the DVI format. In each file, the 27th to the 41th bytes is a time-stamp string. For example, the strings for the two files are "`1999 12 02:0702`" and "`1999 12 02:0703`" respectively. Note that the two 179-Kbyte files are identical except the 15-byte time stamps.

Therefore, the file produced by a re-execution is slightly different to the file produce by the original execution, and the two files have different fingerprints. Our first implementation detected such difference and rejected all re-executions of the LaTeX program.

The time stamps are seldom used by users. Many users are not even aware of the existence of the timestamps. To them, the slight discrepancy is just an artifact of re-execution and can be tolerated. However, the design philosophy of our system is that the file system must not compromise correctness. It assumes that the time stamps are very important to some users for some reasons. Therefore, without a better solution, rejecting all re-execution of the LaTeX program is necessary though undesirable.

4.3.1.2 Design Alternatives

Several solutions have thus been proposed so that our file system does not need to reject re-executions unnecessarily because of the minor re-execution discrepancies.

The first solution is to advice users to run the application in a hypothetical special mode if they want the updates to be shipped by operation. In the special mode, the application does not put a time stamp in output files. However, this solution is not favorable as it sacrifices transparency in two fronts: (1) the user needs to remember to use the special mode if operation shipping is desired; and (2) the application must have such a special mode, otherwise it has to be modified to add the special mode. Incidentally, none of the three applications listed in Figure 4.14 has the hypothetical mode.

The second solution observes that applications get the time-stamp values from the `gettimeofday` system call. It proposes that the kernel on the client could remember the time value that an application made the system call, and Venus ships the time as a part of the operation log. On the surrogate, the kernel could reply with the same time value when the re-executing application made the system call again. Obviously, this requires modification of the behavior of the kernel, either by directly modifying the kernel proper, or by modifying the shared library, which serves as the intermediate layer between applications and the kernel. This idea is not very elegant and looks scary to most computer users.

The third proposed solution uses some application-specific post-processors. A post-processor is a small program that knows the behavior of an application; and it can extract the time stamp from the output file, either from a fixed offset, or by recognizing some patterns that surround the time stamp (some applications, such as rp2gen, do not put time stamps in fixed offsets.). It then tells Venus the value of the time stamp. Venus ships the value as a part of the operation log. The surrogate, after re-execution, invokes a counterpart of the post-processor and replaces the time stamp in the re-generated output file. In this approach, the logging shell will be slightly more complex. It needs to keep a table of applications and application-specific post-processors, and invokes the latter when the former finishes execution. The major disadvantage of this solution is that each application needs to have a specific post-processor.

None of the proposed solutions are as elegant and practical as the finally adopted solution, which is presented in the next sub-section.

4.3.1.3 Fixing Time Stamps by Error Correction

We can view the slight discrepancies in the original file and the re-generated file as if they are “noises”. They are minor differences, and the exact location of the differences are not known to the file system. This view point suggests that we can use the technique of *forward error correction* [12, 16] to *correct* the noises.[4]

The file system, therefore, does the following. Venus on the client computes an error-correction code for each updated file that is to be shipped by operation. The *Reed-Solomon code* is used in the current implementation (Section 4.3.1.4). Venus then packs the code as a part of the operation log. This is done in the requesting phase of shipping stage. The Reed-Solomon code operates on a fixed length data block, and produces a parity block for each data block. A file bigger than the data-block size is chopped into multiple data blocks, and has multiple parity blocks.

In the validation phase, Venus on the surrogate uses MD-5 fingerprints to check if a re-generated file is the same as that of the original. If they are different, it invokes the error-correction routine. The routine uses the parity blocks sent in from the client to correct the “errors” in the data blocks. Note that unlike the solution of

[4]This idea was first suggested by Matt Mathis of Pittsburgh Supercomputer Center.

application-specific post-processor, the routine does not know the exact locations of the errors. After the error-correction step, Venus makes another comparison on the MD-5 fingerprints to determines if the corrected file is the same as the original.

Note that here the technique of forward-error correction is used in a novel way. Conventionally, mostly in communication systems, a sender sends both the data blocks and the parity blocks; whereas in this case, the client sends only the parity blocks but not the data blocks – the data blocks are instead re-generated by an re-execution on the surrogate. Conventionally, forward-error correction code is used to correct communication errors, whereas in this case, the code is used to correct re-execution discrepancies. Figure 4.16 illustrates the differences.

4.3.1.4 Reed-Solomon Codes and the Chosen Parameters

The current implementation uses the Reed-Solomon code (R-S code) for forward error correction. R-S codes are block codes (other examples of block codes are Cyclic Redundancy Check codes, Hamming codes, etc), and they operates on symbols alphabets of more than two values. The alphabet sizes are usually powers of 2 and are written as $GF(2^m)$. A R-S block has a size one less than the alphabet size. For example, a R-S code over $GF(256)$ will have a block size of 255 symbols, or 2040 bits. The number of data symbols in a R-S codeword can be chosen according to the desired error correction ability. To correct up to E errors in a block, $2E$ parity symbols will be needed, and the rest in the block will be the data symbols. [16]

The prototype file system uses the following parameters for the Reed-Solomon code. The symbol-alphabet size is 16 bits, so the block size is 65,535 symbols (or 131,070 bytes). We desire that our code can correct up to 16 errors in each block, so 32 parity symbols (64 bytes) are needed, and there are 65,503 data symbols (131,006 bytes) in the block. It is common to denote the code that we choose as a $(65535, 65503)$ block code over $GF(2^{16})$. With these parameters, the additional network traffic due to the parity block is quite small ($\frac{32}{65503} = 0.049\%$).

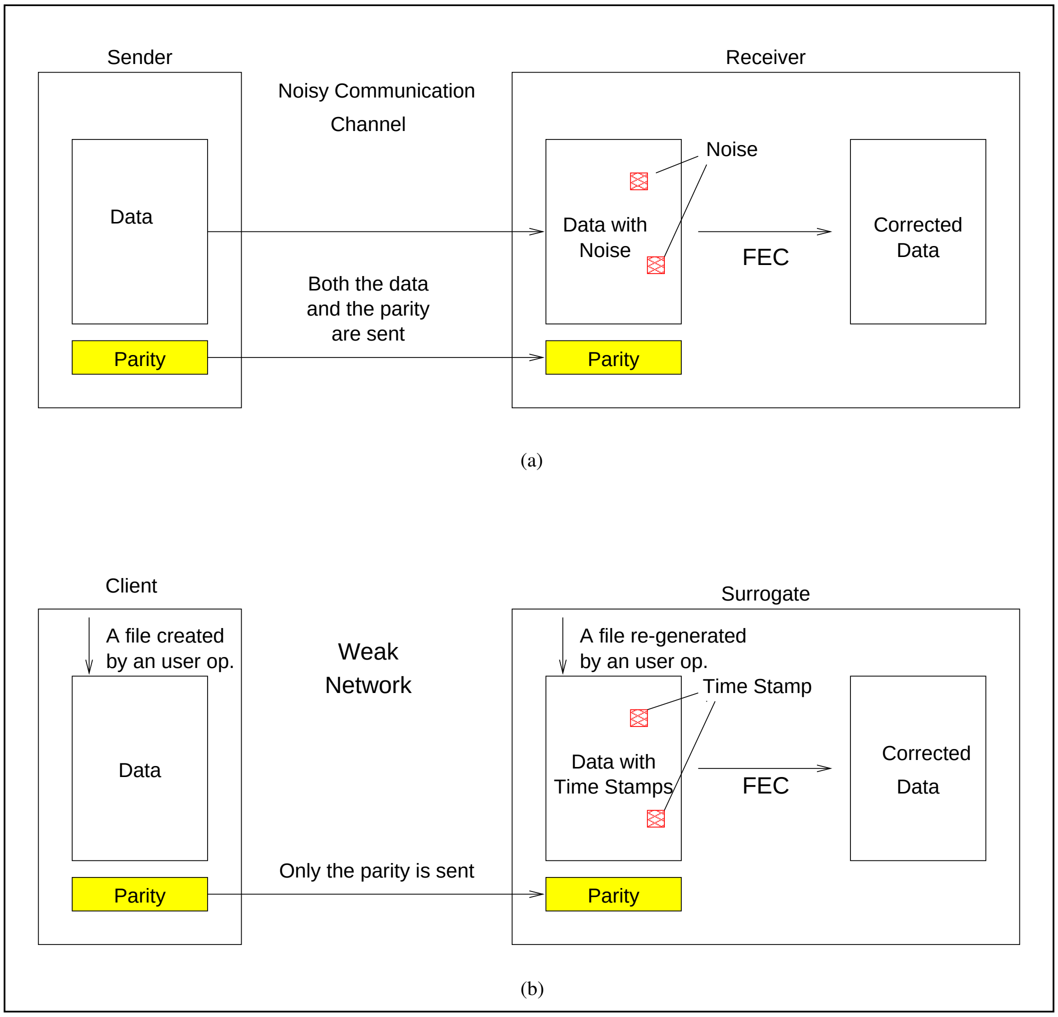

Figure 4.16: Use of Forward Error Correction

(a) The conventional use of forward error correction. (b) The use of forward error correction in fixing re-execution time stamps. Note that a file may actually be chopped into multiple data blocks. In that case, each data block has a corresponding parity block.

4.3.1.5 Limitation and Future Extensions

Some re-execution discrepancies may involve not only byte changes but also length changes. For example, consider the case when an original execution and the re-execution writes a 4-byte time stamp “`9:17`” and a 5-byte time stamp “`14:49`”respectively. The Reed-Solomon code does not correct error with length changes. This is one of its limitations.

However, the idea of using forward error correction can be generalized to the use of any *application-independent post-processors*, which are some small programs that can fix re-execution discrepancies. For example, it is possible to use a binary delta algorithm like the “rsync” algorithm [60]. The “rsync” algorithm‘ can handle length changes, but it needs a much larger overhead in network traffic. A possible future extension of the system is to allow multiple post-processors to be used at the end of re-execution.

4.3.2 Side Effects Due to Temporary Files

4.3.2.1 The problem

The program `ar` is an example of an application that uses temporary files. Figure 4.17 shows the two CMLs on the client and the surrogate after the execution and re-execution of the following user operations:

```
ar rv libsth.a foo.o bar.o.
```

The user operation builds a library file `libsth.a` from two object modules `foo.o` and `bar.o`.

Note that `ar` used two temporary files `sta09395` and `sta16294` in the two executions. The names of the temporary files are generated based on the identity numbers of processes executing the application, and hence they are time dependent. The validation procedure might naively reject the re-execution because the `name` fields

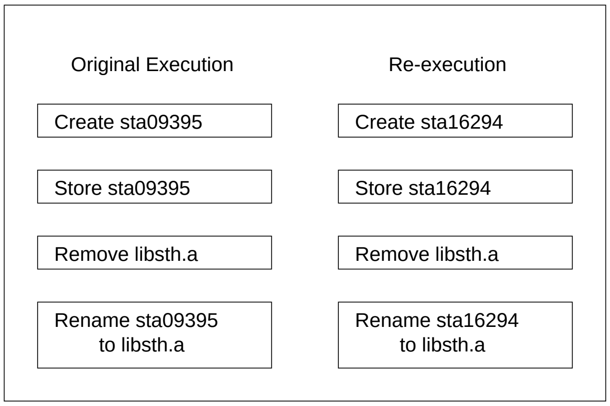

Figure 4.17: CMLs of Two Executions of `ar`

of the two records are different.

4.3.2.2 Solution: Temporary-file Renaming

Temporary files appear only in the intermediate steps of the execution. They will either be deleted or renamed at the end, so their names do not affect the final file-system state. An application uses temporary files to provide execution atomicity. For example, `ar` writes intermediate computation results to a temporary file, and it will rename the file to the target filename only if the execution succeeds. This measure is to ensure that the target file will not be destroyed accidentally by a futile execution.

If a temporary file is created and subsequently *deleted* during the execution of a user operation, its modification records will be deleted by the existing *identity cancellation* procedure (Section 2.3.2). They will not appear in the two CMLs and will not cause naive rejections of re-execution.

However, if a temporary file is created and subsequently *renamed* during the execution of a user operation, its modifications records will be present in the CMLs, and will cause our validation to reject the re-execution.

The solution adopted in the operation-shipping file system is to use a procedure of *temporary-file renaming* to compensate for the side effect. This procedure is done in the early validation phase (Section 4.2.4).

The idea of the temporary-file renaming is to scan the two CMLs and identify all the temporary files as well as their final names. We identify temporary files by the fact that they are created and subsequently renamed in a user operation. For each temporary file used by the surrogate, our file system determines the temporary file name N used by the client in the original execution. It thus renames the temporary file to N. In our `ar` example, the temporary file `sta16294` will be renamed to `sta09395`.

4.4 Complication with Cancellation Optimization

The technique of *cancellation optimization* performed on the CML has been proven to be a very useful technique. It can save log space on a client. The log space includes the persistent storage in both the recoverable virtual memory and the local Unix file. Both of them are scarce resource particularly during a long disconnection period. It can also save network traffic and reintegration work load. Section 2.3.2 gives a brief description of the techniques. More details can be found from [18, page 123–136].

Unfortunately, the need of validation for operation shipping is in conflict with cancellation optimization. Section 4.4.1 explains why there is a conflict, and Section 4.4.2 gives a solution to the problem.

4.4.1 Dilemma: to cancel or not to cancel?

Cancellation optimization is a log transformation. It shortens a CML by canceling some records. Without special consideration about operation shipping, it could cancel any records including those records that are associated with a user operation. However, canceling these records present a difficulty for validating the user operation when it is being replayed. The reason is the following.

Recall from Section 4.2.4 that there are two comparisons in the validation phase. First, there is a comparison on two CMLs: (1) the new CML, which is generated by a replayed user operation on the surrogate, and (2) the reference CML, which is sent from the client to the surrogate. The client prepares the reference CML in the requesting phase by packing all CML records of the user operation. Now, cancellation optimization may have canceled some of these records, so the client cannot prepare a complete reference CML. The result is that the surrogate detects that the two CMLs differ, and rejects the replayed user operation. Second, there is a comparison on the fingerprints of the container files of the `store` records. The fingerprints for the client's container files are computed also in the requesting phase. Again, the container files of those canceled `store` records are gone with the records, so the client cannot computes their fingerprints nor their Reed-Solomon parity blocks. The result is that the surrogate will not have enough information for validation.

Rejecting a replayed user operation just because some of its records were canceled is an artifact introduced by the optimization. The replayed user operation is indeed repeating, just that some information is lost, and it cannot be accepted in the validation phases. The user operation has to use the fall-back mechanism of value shipping. In other words, cancellation optimization is in conflict with operation shipping. Being careless, there seems to be a dilemma: if we want the advantage of the optimization, then we lost the advantage of operation shipping; if we do not want the interference caused by the optimization, then we lose all its advantages.

Before going further into the discussion, a clarification must be made here. There are indeed two types of cancellation optimization in this context: *intra-user-operation* and *inter-user-operation* cancellations. Only the latter is in conflict with operation shipping. When a *canceled* record and the *canceling* record are from the same user operation, it is an intra-user-operation cancellation; when they are not from the same user operation, it is an inter-user-operation cancellation. Intra-user-operation cancellation does not cause problems for operation shipping since it happens on both the original execution and the re-execution. Inter-user-operation cancellation causes problems since it may have happened on the client but may have not yet happened on the surrogate.

In the following discussion, without other qualifier, the term "cancellation optimization" refers to inter-user-operation cancellation optimization. Also, the record being canceled is a record that is eligible for operation shipping. Other records are canceled in the ordinary way as they do not need the special handling.

4.4.2 Solution: Keeping Information in Ghost Records

The solution to the dilemma is the following new procedure of cancellation optimization. On the one hand, Venus allows cancellation optimization to happen, but, on the other hand, it keeps the necessary information around so that a successful validation is possible. The validation phase need three types of information: (1) the reference CML, which contains the thin part of the original CML; (2) the fingerprints of container files; and (3) the parity blocks of the container files. Note in particular that the container files themselves are not needed.

Therefore, when canceling a CML record, Venus marks the record as a *ghost* record rather than throwing it away. Ghost records are used to keep the three pieces of information stated above. They are totally ignored by the value-shipping reintegration mechanism. On the other hand, they are important for the operation shipping mechanism. In the requesting phase, the reintegrator treats them as if they are ordinary records.

Ghost records are not kept forever. They are only kept until the user operation is shipped and finalized. They are thrown away no matter the operation shipping is successful or not. If the shipping is successful, all the CML records, ghost and ordinary, are thrown away as a final step of local commitment. If the shipping is not successful, Venus throws away any ghost records but marks the `SrgRejected` flags for other ordinary records. The ghost records are discarded since there will be no further attempt of operation shipping for the user operation. The ordinary records are not discarded yet, as they will be value shipped in the next round of reintegration.

Now lets study the effect of the new procedure. First, in terms of the space saving for the log storage. Keeping ghost records reduces the effectiveness of cancellation optimization. Recall that a CML has a thin part and a think part. The thin part is stored in the recoverable virtual memory, and comprises everything of the log except the container files; the thick part comprises the container files, which are kept as local Unix files. Now the new optimization procedure saves space in the thick part but not in the thin part. There are space saving in the thick part since, as before, the container files of a canceled `store` record need not be kept. There are no space saving in the thin part since a canceled record is still physically kept in the log – it is just marked as a ghost. In fact, there are even new overhead in the thin part, since additional fields must be added to keep the fingerprint and the parity blocks of a ghost `store` record. In the current implementation, this additional overhead is from 88 bytes to 664 bytes per `store` record. [5]

Second, recall that log-storage spacing is not the only benefits of cancellation optimization, which allows saving also in other aspects: network traffic, reintegration latency, and file-server workload. These savings are all preserved by the new optimization procedure. This is because for the value-shipping reintegration mechanism, the reintegrator ignores all the ghost records. Therefore, they do not incur reintegration traffic, nor do they increase the reintegration latency or server workload.

Third, most importantly, the new procedure expands the set of useful replayings of user operations. That is, more updates can be shipped by operation rather by value, and this implies a huge saving in both network traffic and reintegration latency.

In summary, the new procedure preserves both useful techniques: cancellation

[5] Sixteen bytes for the MD-5 fingerprints, one to ten 64-byte parity blocks, four bytes for a pointer pointing to the separately stored parity blocks, and four bytes for the number of parity blocks. Note that, to bound the size of the additional overhead, Venus keeps only at most ten parity blocks. For files larger than ten data blocks, the eleventh blocks onwards will not be error corrected. This bound does not impose much problem in practice, since re-execution discrepancies (such as time stamps) usually appears in the early part of a file.

optimization and operation shipping. There is a small price to pay for the slight overhead in the thin part of the CML, but the gain is that more updates can be sent by operation shipping.

4.5 Chapter Summary

This chapter presents the design and implementation of the file system, which is an extension of Coda, that supports application-aware operation shipping. There are two main stages involved: logging and shipping. The logging stage happens on a client, and it encompasses the activities that establish an association between a high-level user operation and the low-level file-system updates that it generates. The shipping stage involves three parties: the client, the surrogate, and the server. The client ships the user operation to the surrogate, which then attempts to re-generate the same set of low-level file-system updates by replaying the user operation. If the surrogate succeeds, it can reintegrate the updates with the file server on behalf of the client; otherwise, the client falls back to reintegrate the updates with the server directly, using the existing value shipping mechanism.

Besides, there are two interesting lessons learned in the course of developing the file system. The first lesson is that even though a replayed user operation does not re-generate the same set of updates as the original execution, it is possible to do something to fix the re-execution discrepancies. Two techniques have been proposed: a novel use of forward error correction for fixing discrepancies due to time stamps, and the technique of temporary-file renaming for fixing discrepancies due to temporary files. The second lesson is that, if not paying special attention to the records associated with user operations, cancellation optimization would render many user operations not shippable by operation. A new procedure, using the concept of ghost CML record, is thus designed and implemented.

Section 6.2 will show the performance gain of operation shipping.

Chapter 5

Application-aware Operation Shipping

The previous chapter has studied *application-transparent operation shipping*; this chapter takes the idea further and discusses *application-aware operation shipping*. The following are the two main research questions in this discussion:

1. What is the right architecture for application-aware operation shipping?
2. How difficult is it to enable an existing interactive application to log and replay user operations?

The answer to the first question is that there are indeed two design alternatives. The first alternative is simpler and it supports *re-execution in the one-shot style*; the second alternative is more complex and it supports *re-execution in the iterative style*. They imply different *granularities of update propagation*. The meanings of the three terms will be discussed in this chapter.

In this thesis work, simplicity is favored. So the design and implementation of the prototype are built around the first alternative. However, usage experience accumulated in the long run may suggest that the second alternative is better. For completeness, this chapter will cover both design alternatives. Of course, it will first discuss why there are the two alternatives.

For the second question, a popular image application – the GNU Image Manipulation Program (the GIMP) – is used as a case study. It has been extended to add the capabilities of operation logging and replaying. The needed modification on the program has been found to be moderate. This answer is favorable as it indicates that operation shipping will have a higher chance of being accepted by the users.

This chapter begins with an analysis of the problem model. In Section 5.1, it presents several basic concepts and then explains why there are two design alternatives for the architecture. It then discusses the two alternatives one by one in Section 5.2 and Section 5.3. The design and implementation of the prototype is based on the first design alternative, and they are discussed also in Section 5.2. The extension of the GIMP is also presented in the same section.

5.1 Analyzing the Problem

In this section, the alternatives for designing a mechanism for application-aware operation shipping will be presented. However, before we can go into the discussion, we should first understand the nature of interactive applications and their application-specific commands.

5.1.1 Logging and Replaying Processes

When an interactive application is involved in application-aware operation shipping, there are two processes executing the application. The first one is the process of the original execution of the application on the client, and is called the *logging process*. Its role in operation shipping is to log the user operations while the user is working with the application. The second one is the process for re-executing the application on the surrogate, and is called the *replaying process*. Its role in operation shipping is to replay the logged user operations. (In fact, as explained in Section 5.1.4.2, there can be more than one replaying processes for one logging process.)

5.1.2 One-shot and Iterative Execution Styles

The lifetime of a process which executes an application is called *an application session*. It is the duration between the moment the application is started by the `exec` system call to the moment the application terminates using the `exit` system call.

There is a fundamental difference between the *execution styles* of non-interactive applications and interactive applications. In this thesis, the two styles are called the *one-shot style* and the *iterative style*. They are defined in the following.

In the one-shot execution style, there is exactly one user operation per application session. Non-interactive applications execute in this style – the user operation is the invocation command of the applications. A non-interactive application terminates with the finishing of the performing of the user operation.

On the other hand, in the iterative execution style, there can be more than one user operations per application. Interactive applications execute in this style. After an interactive application is invoked, it is sitting in an idle loop for most of the time, waiting for some user operations to be issued. These user operations are some application-specific commands. When a command is issued, the application performs the task specified by the command, and returns to the idle loop when it finishes performing the task. It remains in this idle-perform-idle loop until the user explicitly instructs it to exit.

The logging process of an interactive application always executes in the iterative style, but the replaying process may executes in either the one-shot or the iterative style. The latter is a design decision to be made by the designers of the mobile file system and the application. Section 5.1.4.2 will explain that the design decision is tightly coupled with the design decision regarding propagation granularity.

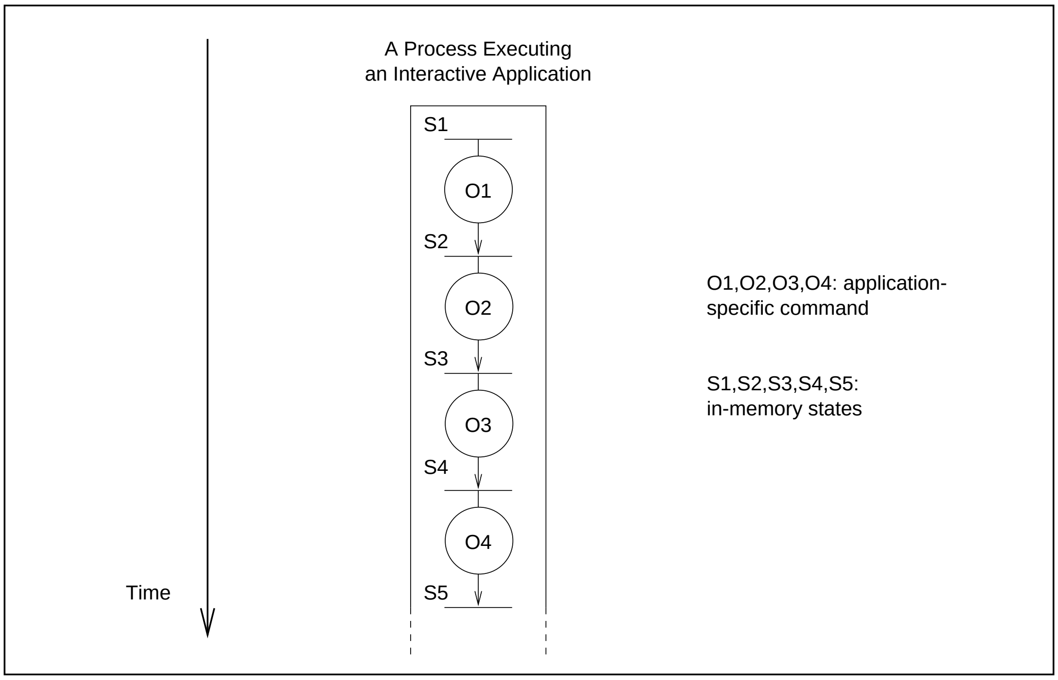

Figure 5.1: Application-specific Commands and In-memory States of a Process

This figure illustrates the interdependency between application-specific commands and in-memory states of a process. The result of executing a command depends on the in-memory state, but at the same time, the command changes the in-memory state.

5.1.3 Application-specific Commands

Recall from Section 3.3 that there are two different types of user operations for the two operation-shipping mechanisms. For application-transparent operation shipping, the user operations that are useful are invocation commands of non-interactive applications. For application-aware operation shipping, the user operations that are useful are application-specific commands.

To prepare for the coming discussion, two concepts related to application-specific commands are introduced here:

In-memory state of a process. The execution context of an application-specific command includes not only the process environment, as that of in the case for an invocation command of a non-interactive application, but also the *in-memory state* of the process executing the application. Also, the in-memory state and the command are interdependent on each other. That is, the result of executing a command depends on the in-memory state, but at the same time, the command changes the in-memory states. This is illustrated in Figure 5.1. For a replaying command to be repeating, it is critical to restore the in-memory state before executing the command. We will see why this issue is important in Section 5.1.4.2,

Two types of application-specific commands. Application-specific commands can be further classified into two types: *editing* and *saving* commands. An editing command changes only the in-memory state of the running process, it does not induce any mutating file-system operations. On the contrary, a saving command does induce file-system mutations. The distinction of whether or not a command induce mutations is important, as will be explained in Section 5.1.4.1.

5.1.4 Design Considerations

5.1.4.1 Granularity of Update Propagation

For interactive applications, there will be more than one application-specific commands per application session. For the purpose of operation shipping, these commands are put into *command groups* that will be used as the basic unit of update propagation. Now, there is a design question on the granularity of command groups. Two different approaches are possible. The first approach puts all commands into a single command group; the second approach puts the commands into multiple command groups.

The question on grouping commands is also a question on the *granularity of update propagation*. In the first approach, update propagation will happen *only after* the application has exited; whereas in the second approach, update propagation may happen much earlier than the moment of exiting the application. Therefore, the former is called *propagation-after-exit*, and the latter is called *early-propagation*.

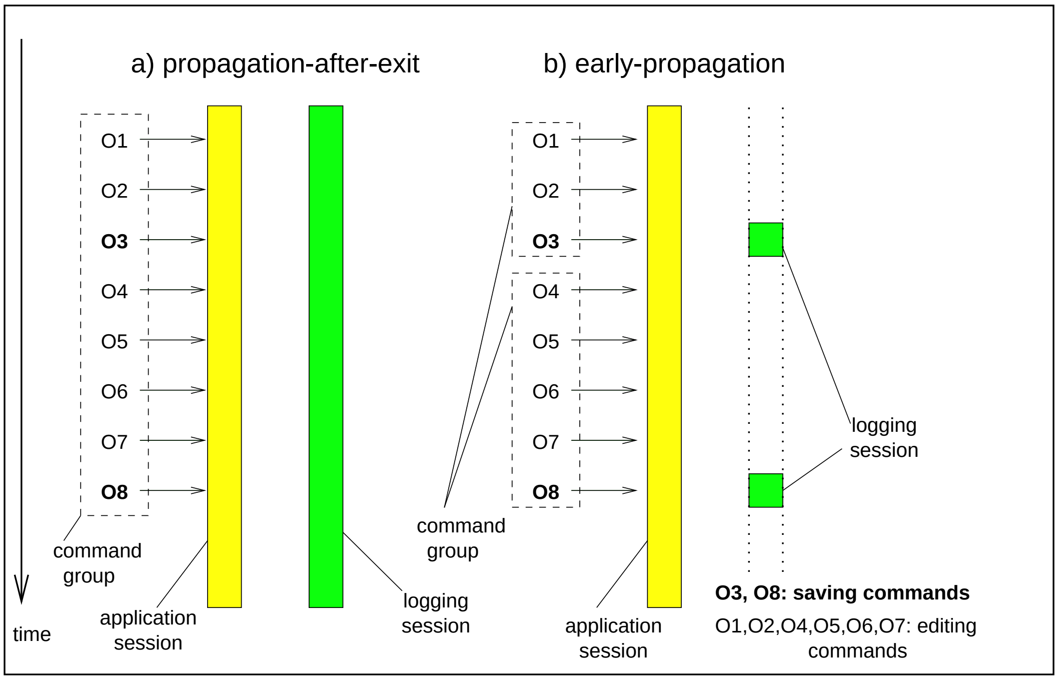

Figure 5.2: Two Different Granularities of Command Grouping

This figure shows the two approaches of grouping application-specific commands of an application session: (a) In the propagation-after-exit approach, all commands are put into one command group. (b) In the early-propagation approach, the commands are put into multiple command groups.

The two approaches will be discussed in turn in the following. They demand two different re-execution styles, as discussed in Section 5.1.4.2.

Propagation after exit. In this approach, all commands in an application session are put into one command group. That means we can make the whole application session as one logging session (Figure 5.2a). There are two main advantages for this approach.

First, the logging and shipping mechanisms for this approach is simpler, because the *restoration of replaying contexts* for individual commands is easy, and the *one-shot re-execution style* can be used. The meaning of the two concepts will be discussed in Section 5.1.4.2.

Second, cancellation optimization is more effective that the early-propagation approach. This is because more commands can cancel log records of previous commands as intra-operation cancellations rather than inter-operation cancellations. Recall from Section 4.4.2 that the cancellation optimization proceeds in two flavors: intra-operation cancellations proceed in the traditional way, whereas inter-operation cancellations proceed by keeping ghost CML (client-modify log) records. The latter is less effective than the former since there is no saving in the storage space for the thin part of the CML.

On the other hand, this approach suffers from several disadvantages. First, the semantics of saving commands of an application is changed. The mutating effects of them will be propagated *only after* the application exits, since, as discussed in Section 4.1.3, individual updates of a logging session will be eligible for updates only after the session terminates. In other words, the users are required to change their working habits – at the moment that they want their updates be eligible for propagation, they have to explicitly exit the application.

Second, in this approach, the logging sessions tend to be of longer duration, and the chance that they will be rejected is higher. This is because when more commands are involved, the chance is higher that some of them are non-repeating.

Third, the longer the duration of a logging session, the higher the chance that a breaking condition may happen and make the user operation not eligible for operation shipping. Recall from Section 4.2.2 that one of the breaking conditions is that the records of a user operation in the CML are interleaved by some other records that are not associated with the same user operation.

Despite the disadvantages, the approach of propagation-after-exit fits very well with the simpler mechanism of *one-shot re-execution*. So it is important in practice. Indeed, the prototype system designed and implemented in this thesis uses this approach.

Early propagation. Unlike the approach of propagation-after-exit, this approach puts commands of an application session into multiple command groups, that means update propagation can be done earlier than the moment of exiting the application. The way of grouping is to recognize those saving commands, which will serve as the dividers. The grouping algorithm proceeds as follows. Suppose an application sees a sequence of commands, $O_1, O_2, \ldots, O_m$, the application examines, in the same order, the commands one by one. If the current command O_j is a saving command, then O_j, together with all the previous O_i, $i < j$, that are not yet marked for any command groups, will be marked for group G_j. The logging session of a command group is the duration encompassing the execution of the saving command, so that all the mutating file-system call induced by the command will happen within the logging session. Figure 5.2b illustrates how commands are grouped, and how each saving command is encompassed by a logging session.

A user issues the saving command when he or she wants to "write something to the file system." The logging application recognizes the command as a divider and forms a new command group. The preceding editing commands are also needed for the group since they help to transform the in-memory state to one that is written to the file system by the saving command.

There are several advantages for the approach of early-propagation. First, the semantics of saving commands are honored. Second, the chance that a command group is rejected is smaller. Third, the chance that a breaking condition may happen to a user operation is smaller.

However, in this approach, there are more than one logging sessions per application session. This demands a more complicated mechanism of operation logging and shipping. The difficulty will be discussed in the next sub-section.

5.1.4.2 Re-execution Styles

Recall that non-interactive applications and interactive application run in two very different styles. A non-interactive application runs in the one-shot style, and there is only one user operation per application session – the invocation of the application;

whereas an interactive application runs in the iterative style, and there are many user operations per application session.

Now, given the fact that a logging instance of an interactive application executes in the iterative style, should the replaying counterpart run in the iterative style too? In fact, there are two alternatives for the re-execution style: one-shot and iterative. The two alternatives are strongly coupled with the two design alternatives regarding propagation granularity, and they will be examined one by one in the following.

One-shot Re-execution. This is a simpler style. When Venus on the surrogate receives a user-operation log, it spawns a replaying process. The replaying process replays the user operations stated in the log and terminates upon finishing the replaying. When Venus receives another user-operation log, it spawns another replaying process.

This style matches pretty well with the propagation-after-exit approach described in the previous sub-section. In Section 5.2, a prototype designed and implemented using this style will be presented.

However, this style does not match well with the early-propagation approach. The reason is due to the problem of *restoration of replaying contexts*, as explained in the following.

Figure 5.3 illustrates the problem. On the left there is a logging instance of an application in which a user performed two groups of commands: G_1 and G_2. To support early-propagation, the client ship the two groups to the surrogate in two batches. Suppose the surrogate re-executes the command groups in the one-shot style, as shown on the right of the figure. When G_1 arrives at the surrogate, Venus will spawn a replaying process P_1', which will replay the command group and then terminate immediately. When G_2 arrives later, Venus will spawn another process P_2' to replay G_2.

Recall from Section 5.1.3 that the result of an application-specific command depends on the in-memory state of the process, and that the command also transforms the in-memory state of the process. For example, in the logging process shown in

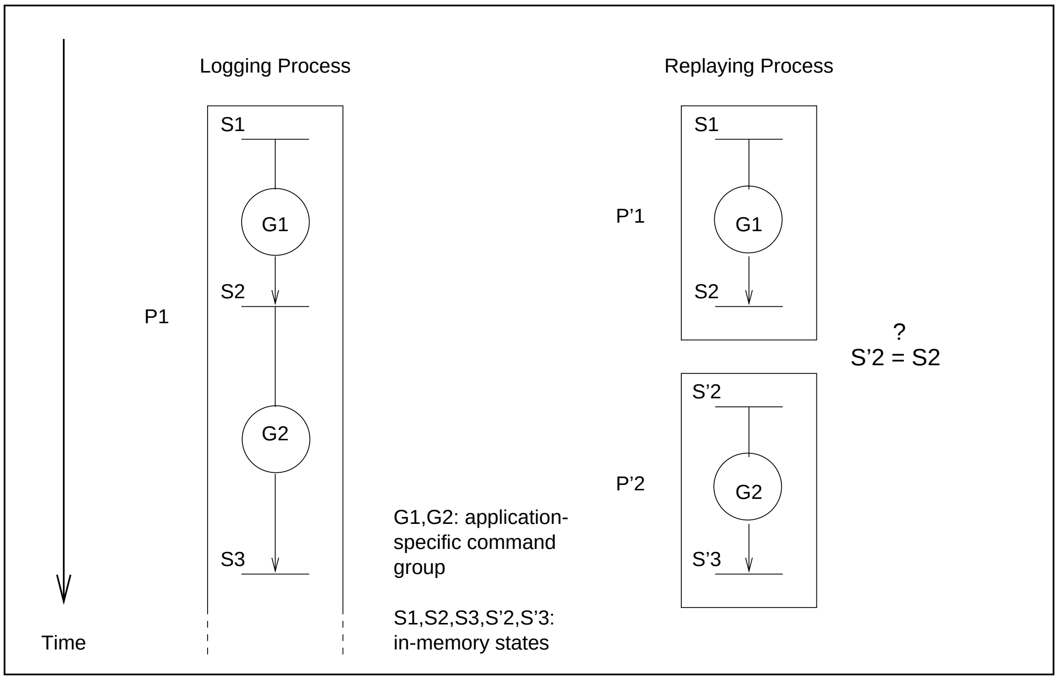

Figure 5.3: Problem of Restoration of Replaying Context

This figure illustrates the problem of restoring relaying context. If, on the surrogate, the two command groups G_1 and G_2 are replayed by two different processes P_1' and P_2', the in-memory state S_2 resulted from the replaying of G_1 will be gone as P_1' terminates, and P_2' will be replaying G_2 with a different in-memory state S_2'.

Figure 5.3, the result of G_1 depends on the in-memory state S_1, but G_1 will also transform the state to S_2.

Now, consider the replaying of G_2 on the surrogate. It can produce the same result as the original only if it is replayed with the same context. That is, S_2' should be the same as S_2. However, in general the two contexts are not the same, since S_2 is reached *only after* the execution of G_1. Therefore, here we have a problem of *restoration of replaying contexts*.

Two solutions are proposed to solve the problem. The first solution is to *restore contexts by an artificial load command*. The application is programmed in such a way that the second replaying process P_2' will insert an artificial load command (Figure 5.4a). The hope is that G_1 may have written something to the disk, by

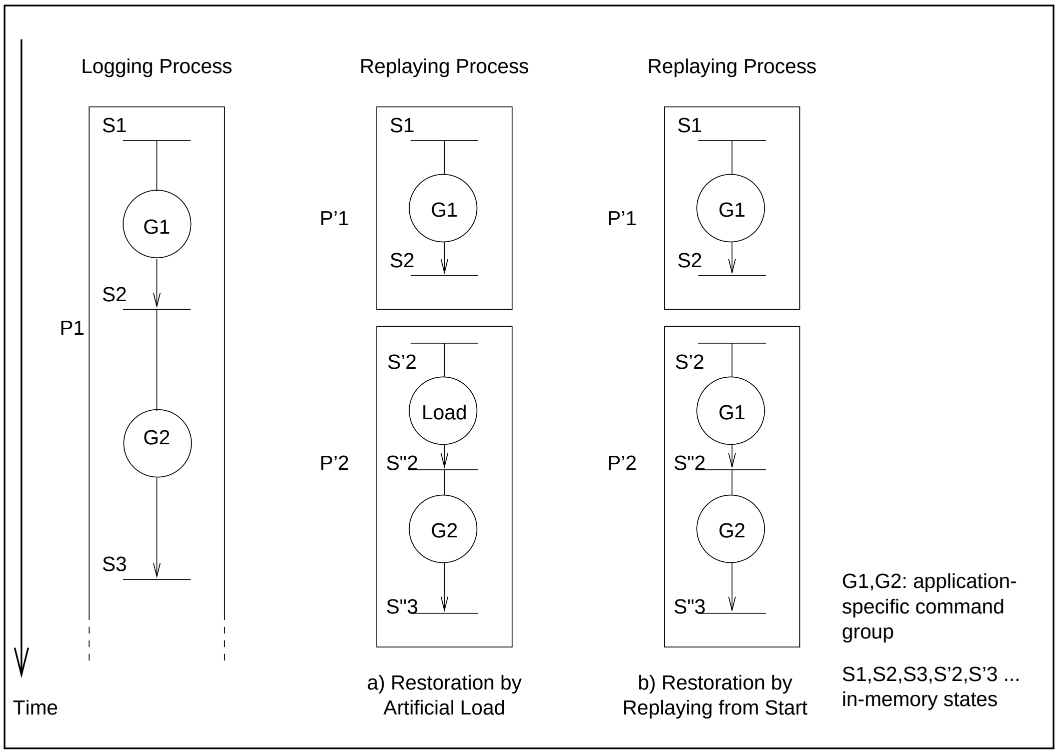

Figure 5.4: Two Proposed Solutions to the Problem of Restoration of Replaying Contexts

This figure illustrates the two proposed solutions to the problem of restoration of replaying contexts: (a) restoration by inserting an artificial load command, (b) restoration by replaying from the start. The original logging process is shown on the left as a reference.

loading them the replaying context can be restored. However, this solution will not work when the applications does not write all of its in-memory states to disk. Such loss of information is in fact very common. For example, when the GIMP saves an image, it does not save some auxiliary layers supporting the editing of the image. Another example is that the GIMP may perform lossy compression (such as the JPEG compression) when it saves an image. Reloading the same image from the disk does not get back the same in-memory state.

The second solution is to *restore contexts by replaying from the start*. That is, to get S_2 that is needed by the replaying of G_2, P_2' replays G_1 (Figure 5.4b). However, since G_1 has been ran once by P_1', it may have changed the state of the machine that

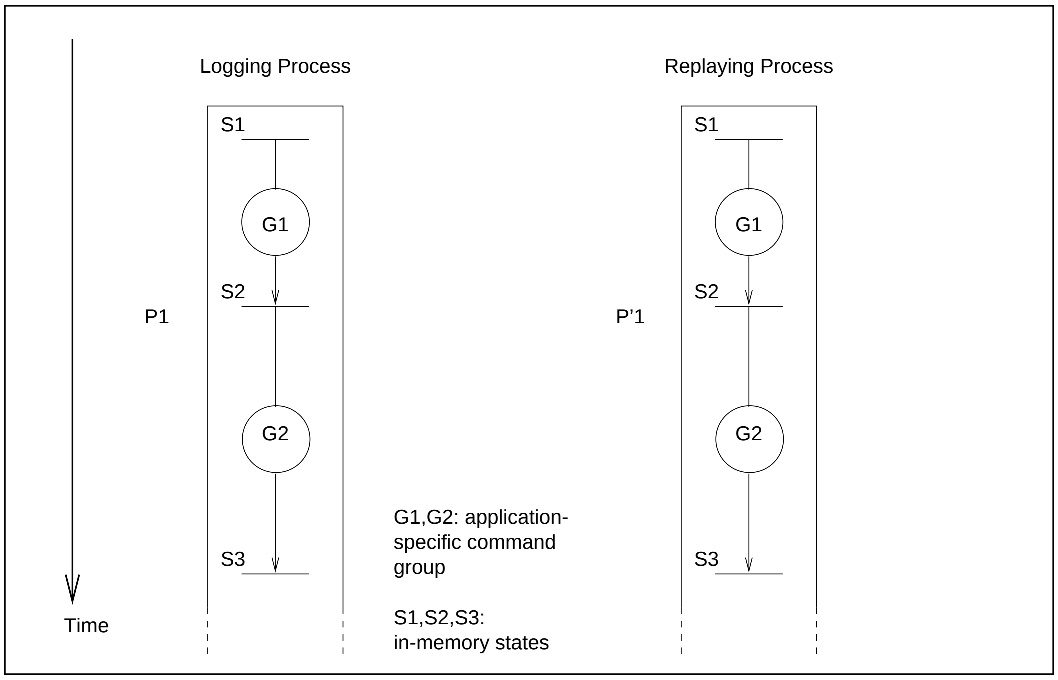

Figure 5.5: Iterative Re-execution Style

This figure illustrates the idea of re-execution in the iterative style. When the replaying process finishes replaying G_1, it will not exit but will sit in an idle loop waiting for more command groups to come. The in-memory state S_2 is thus retained.

even G_1 cannot recompute S_2. For example, if G_1 consists of loading a file F from the file system, changing F by some ways, and then save F to the file system, re-running G_1 will load the modified version of F.

Both of the previous two solutions are not very clean. In general, to support early propagation, the proper re-execution style is the iterative style. The reason is explained in the following.

Iterative Re-execution. Figure 5.5 illustrates the idea of iterative re-execution. When the replaying process finishes replaying G_1, it will not exit. Rather, it will return to an idle loop waiting for more command groups to come. Since the process does not exit, the in-memory state will be retained. To support this, the replaying process is not spawned upon the receiving of an application-specific command group. Rather, it is pre-arranged and sits in an idle loop waiting for the arrival of command groups.

This re-execution style demands a different architecture of operation-shipping mechanism than that is demanded by one-shot re-execution style. A surrogate needs to export two different types of services for its client. The first is for the setting up and shutting down of the replaying process, and the second is for the replaying of application-specific command groups. Note that, in contrast, for the one-shot re-execution style, the two types of services are lumped into one. That is, when a client requests the replaying of a command group, it is also requesting the spawning of a replaying process.

Apart from enabling the approach of early-propagation, the iterative re-execution style has another advantage. That is, the overhead involved in the replaying of a command group will be smaller, because the surrogate needs not spawn a new replaying process every time a new command group is to be replayed.

The next two sections will discuss the two design alternatives based on the two re-execution styles.

5.2 Design Alternative 1: One-shot Re-execution Style

This design alternative needs a simpler design than the other design alternative. It supports pretty well the approach of propagation-after-exit, although it does not support quite well the approach of early-propagation. In favor of simplicity, the prototype implemented with this thesis is based on this alternative.

An popular image program – the GNU Image Manipulation Program (the GIMP) – is selected as an example application. The details of the extension needed for the logging and replaying of user operations are also discussed in this section.

```
VIOC_BEGIN_OP        fs_begin_op       (char *command,
                                        char **args,
                                        char **env,
                                        char *cwd,
                                        mode_t umask);
VIOC_END_OP          fs_end_op         (pgid_t pgid);
VIOC_PUT_APP_OPLOG   fs_put_app_oplog  (char *srcfilename);
VIOC_GET_APP_OPLOG   fs_get_app_oplog  (char *destfilename);
```

Figure 5.6: Interface for Application-aware Operation Shipping - Design Alternative 1

The left column of this figure shows the four `ioctl` commands needed for supporting application-aware operation shipping. The right column shows the four corresponding convenient functions and their arguments. Comparing to Figure 4.3, Two new `ioctl` commands are added to support the putting and getting of application-specific operation logs.

The following discussion will be divided into two parts: the part on the file-system side, and the part on the application side. Not surprisingly, the extension needs to be done on the file-system side is relatively simple, and most of the works need to be done is on the application side. The two parts will be discussed in the following two subsections.

5.2.1 The File-system Side

The extension for the support of application-aware operation shipping is quite simple. The existing mechanism for the support of application-transparent operation shipping can be largely reused, with the following two exceptions.

First, on top of the `VIOC_BEGIN_OP` and the `VIOC_END_OP` commands, two more `ioctl` commands are needed. They are for the "putting" and "getting" of the *application-specific operation log* (Figure 5.6). The idea is the following. The application understands the application-specific commands, but the file system does not. Also, it is the replaying instance of the application who replays the commands, not the file system. Therefore, the application and the file system have different responsibilities. The application is responsible to prepare the application-specific

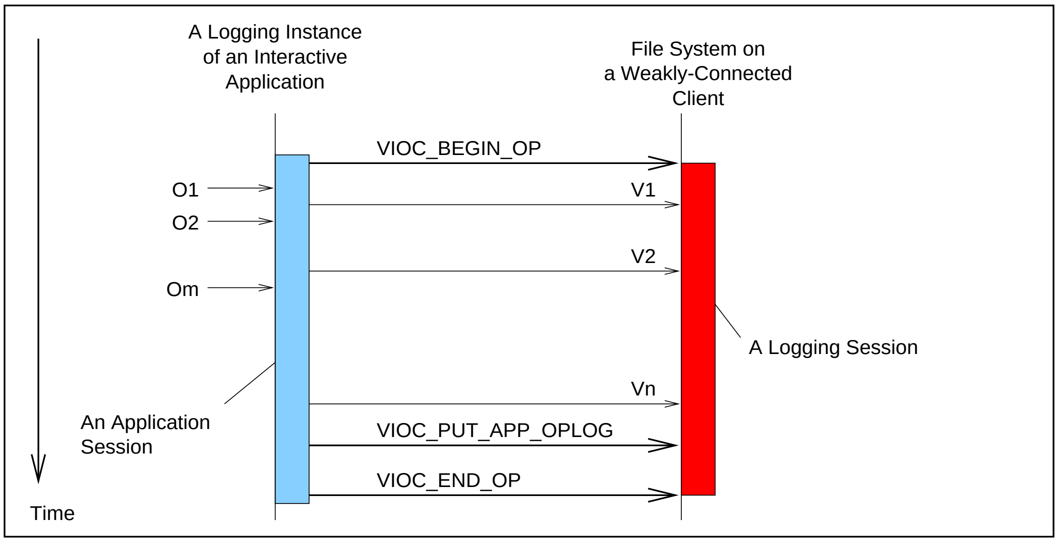

Figure 5.7: Logging of Application-specific Commands - Design Alternative 1

This figure shows how the logging instance of an interactive application makes use of the `VIOC_PUT_APP_OPLOG` to tell the file system about the application-specific operation log.

operation log on the client, and to interpret the log and replay the commands that are stated in the log on the surrogate. The file system is responsible to forward the log from the client to the surrogate. The application and the file system interact with each other using the "putting" and "getting" commands. The logging process of the application puts the log to Venus on the client (Figure 5.7), and the replaying process gets the log from Venus on the surrogate (Figure 5.8).

Second, in application-transparent operation logging, the two `ioctl` commands `VIOC_BEGIN_OP` and `VIOC_END_OP` are issued by a logging shell. However, here, they are issued by the application itself. This change is a reflection on a difference in philosophy regarding the applications. For application-transparent operation logging, we do not want to modify the application at all; whereas for application-aware operation logging, the application needs to be modified anyway.

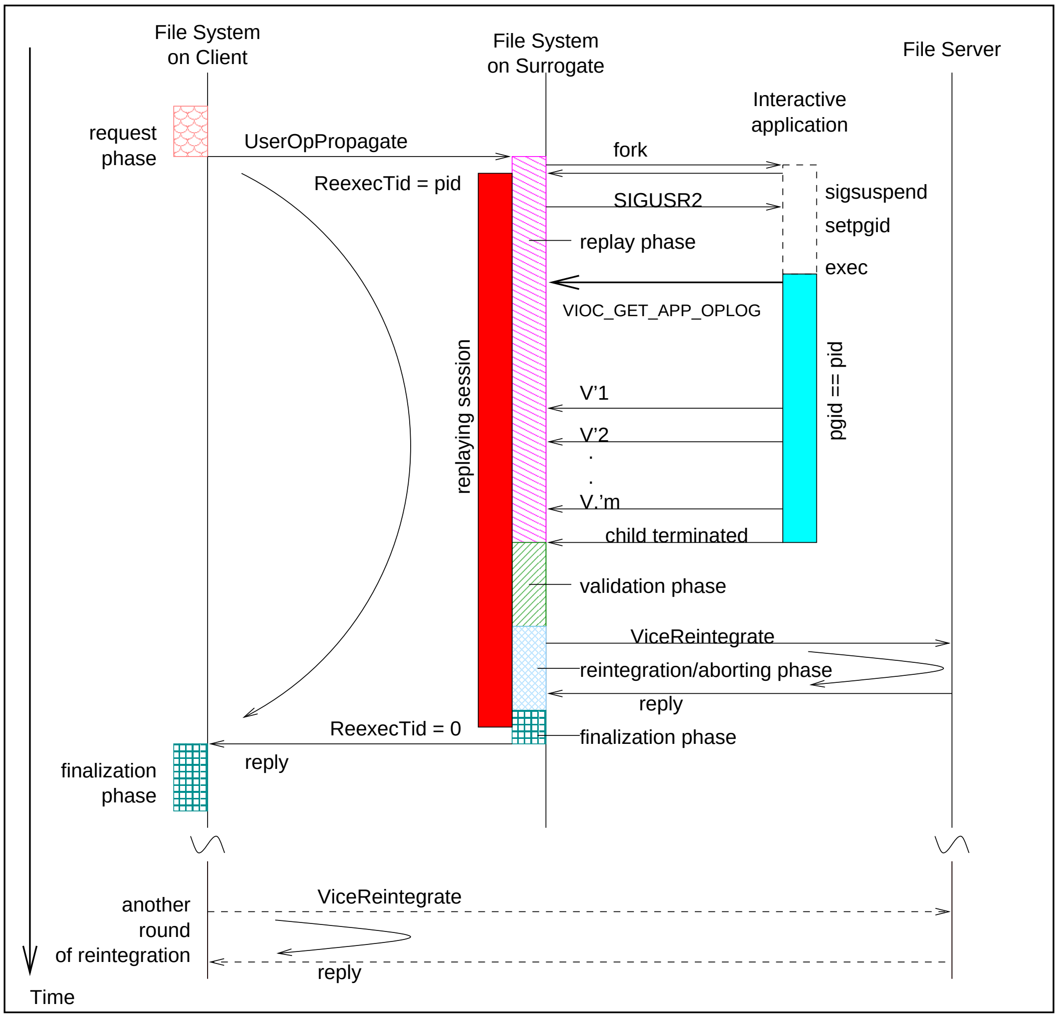

Figure 5.8: Shipping Application-specific Commands - Design Alternative 1

This figure shows how the replaying instance of an interactive application makes use of the VIOC_GET_APP_OPLOG to retrieve from the file system the application-specific operation log.

5.2.2 The Application Side: Using the GIMP as a Case Study

How difficult would it be to extend an existing interactive application so that it can log and replay user operations? This section presents some experience on extending one particular application: the GIMP.

5.2.2.1 GIMP Background

The GNU Image Manipulation Program, or the GIMP, is an open-source program that is suitable for tasks such as photo retouching, image composition and image authoring [3, 22]. It is a very popular program, particularly in the Linux community. Many professional graphical designers are using it everyday. Figure 5.9 shows a screen shot of the application in action.

There are two main reasons for choosing the GIMP as a case study for this project. First, by nature, the GIMP works with image files. These files are typically large (say, larger than 64 Kbytes), and are difficult to be shipped across weak networks. Second, the GIMP is an open-source program, that means everyone has much freedom in obtaining its source code and extending it.

5.2.2.2 Application Structure

The internal structure of an existing application affects how easy it is to add operation-logging and operation-replaying facilities. Ideally, the application should have a clean decoupling of the following two functionalities: *user interface* and *internal functions*. The user interface is handled in some *command-accepting* modules, and the internal functions are handled in some *command-executing* modules. A clean decoupling means that there are confined and well-defined interfaces between the two types of modules. It is often found in Modern applications, since it allows a greater flexibility in the design of the user interface, facilitates the implementation of undoing and redoing commands, and facilitates the support for scripting.

Figure 5.9: GIMP in Action

The upper two windows are two images being edited by the GIMP, the lower left window is the DB Browser, which allow the user to browse the functions exported via the procedure database, the lower right window is the main window with the tool palette.

```
-------------------------------------------------------------------
Name:  file-jpeg-load
Blurb: loads files of the jpeg file format
In:    run_mode     INT32          Interactive, non-interactive
       filename     STRING         The name of the file to load
       raw_filename STRING         The name of the file to load
Out:   image        IMAGE          Output image
-------------------------------------------------------------------
Name:  gimp-color-balance
Blurb: Modify the color balance of the specified drawable
In:    image        IMAGE          the image
       drawable     DRAWABLE       the drawable
       transfer_mode INT32         Transfer mode: (SHADOWS(0,
                                   MIDTONES(1), HIGHLIGHTS(2))
       preserve_lum INT32          Preserve luminosity values at each
                                   pixel
       cyan_red     FLOAT          Cyan-Red color balance:
                                   (-100 <= cyan_read <= 100)
       magenta_green FLOAT         Magenta-Green color balance:
                                   (-100 <= magenta_green <= 100)
       yellow_blue  FLOAT          Yellow-Blue color balance
                                   (-100 <= yellow_blue <= 100)
-------------------------------------------------------------------
Name:  gimp-brightness-contrast
Blurb: Modify brightness/contrast in the specified drawable
In:    image        IMAGE          the image
       drawable     DRAWABLE       the drawable
       brightness   INT32          brightness adjustment
                                   (-127 <= brightness <= 127)
       contrast     INT32          contrast adjustment
                                   (-127 <= contrast <= 127)
-------------------------------------------------------------------
Name:  file-jpeg-save
Blurb: saves files in the jpeg file format
In:    run_mode     INT32          interactive, non-interactive
       image        IMAGE          input image
       drawable     DRAWABLE       Drawable to save
       filename     STRING         The name of the file to save the
                                   image in
       raw_filename STRING         The name of the file to save the
                                   image in
       quality      FLOAT          Quality of saved image
                                   (0 <= quality <= 1)
       smoothing    FLOAT          Smoothing factor for saved image
                                   (0 <= smoothing <=1)
       optimize     INT32          Optimization of entropy encoding
                                   parameters
-------------------------------------------------------------------
```

The three columns in the list of input and output arguments are the name of arguments, type of arguments, and a brief description of the arguments, respectively.

Figure 5.10: Examples of some Exported Functions

The GIMP, for example, indeed has such a clean decoupling. The model that the GIMP uses is called the *procedure database (PDB)*. Each internal function has a well-defined interface. It is registered to the PDB upon the startup of the application. The interface allows it to be called by some other internal functions, such as a user-interface handling routine, or by some external functions, such as plug-ins and scripts. Figure 5.10 lists some functions that are exported via the PDB.

5.2.2.3 Experiences

I found that the effort needed to extend GIMP for operation logging and replaying is very reasonable. It took only three months of work to make an initial prototype to work. [1] The extended GIMP can run in two special modes: `oplog` and `reexec`. In the `oplog` mode, the prototype can log some GIMP-specific commands, write them on a GIMP-specific operation log, and pass it to the file system. In the `reexec` mode, the prototype can get the GIMP-specific operation log from the file system and replay the commands stated on the log. The prototype re-executes in the one-shot style, so it will exit immediately when it finishes replaying all the commands stated on a log.

The modification needed is moderate, but it does not involve major changes in the internal logic of the application. Each GIMP command being logged needs a few lines of straightforward code for operation logging. Currently, 30 different commands can be logged. The number represents about one sixth of the total number of different types of commands that can be performed in the GIMP. Figure 5.11 lists the commands that can be logged and replayed by the prototype.

[1] The prototype is based on the mainstream version 1.0.2. The source code is available for download from [2].

User Operation	Explanation
anchor	anchor (pin) a layer to an image
apply canvas	add a canvas texture to an image
auto-stretch contrast	automatically stretch contrast of an image (in RGB space)
auto-stretch HSV	automatically stretch contrast of an image (in HSV space)
blur	blur an image by random displacement of pixels
brightness/contrast	change the brightness/contrast of an image
bmp load	load a file in the bmp format
bmp save	save a file in the bmp format
color balance	change the color balance of an image
colorify	make an image looks like it is being viewed thru' colored class
cubism	transform an image into cubist art
desaturate	remove colors from an image
emboss	carves a three-dimensional look to an image
equalize	equalize the color value of an image
file new	create a new image
gaussian blur (IIR)	Gaussian blur (IIR)
gaussian blur (RLE)	Gaussian blur (RLE)
gradient map	map the image with an active gradient map
hue-saturation	modify the hue, lightness, and saturation of an image
invert	invert the color value of an image
jpeg load	load a file in the jpeg format
jpeg save	save a file in the jpeg format
levels	modify the intensity levels of an image
move	move a layer to a new position
mosaic	convert an image into a collection of tiles
normalize	normalize the contrast of an image
oilify	oil painting effect
posterize	creating an indexed image
text	add text to an image
threshold	threshold an image

Figure 5.11: GIMP commands that can be logged and replayed by the prototype

```
(define (script-fu-oplog-reexec)
  (let*
      ( ; declaring local variables
        (theImage_1)
        (theLayerDrawable_2)
        (theLayerDrawable_3)
      ) ; end of local variables
      (set! theImage_1 (car (file-jpeg-load 1
        "gibralter2.jpg" "gibralter2.jpg")))
      (set! theLayerDrawable_2 (car
        (gimp-image-get-active-layer theImage_1)))
      (set! theLayerDrawable_3 (car (gimp-text-ext theImage_1
         theLayerDrawable_2 263.000000 571.000000
         "Gibralter" 0 1 15 0
         "*" "helvetica" "*" "*" "*" "*" "*" "*")))
      (gimp-layer-translate theLayerDrawable_3 23 0)
      (gimp-floating-sel-anchor theLayerDrawable_3)
      (file-jpeg-save 1 theImage_1 theLayerDrawable_2 "
         /coda/usr/c.clement/tmp/test2/t31.jpg" "t31.jpg"
         1.000000 0.000000 1)
  )
)
```

Figure 5.12: An Example GIMP-specific Operation Log

This figure shows a portion of an example GIMP-specific operation log. It records the following user operations. The user loaded a jpeg file name "gibralter2.jpg". He then added a text string "Gibralter", moved the text layer to the lower right corner of the image, and anchored the layer. Finally, he saved the annotated image to another file called "t31.jpg". The log was automatically generated by the extended GIMP, but it was slightly edited (with long lines split into shorter lines) for better presentation in the figure.

The GIMP-specific operation logs are actually GIMP's scripts using the Scheme language. The GIMP has the scripting facility that allows experienced users to write scripts for repetitive tasks. Normally, the scripts are created by users, but our prototype constructs the scripts automatically and uses them as application-specific operation log. Figure 5.12 shows an example of such a log.

Section 6.3 will present a quantitative evaluation of the performance of the extended GIMP working with the operation shipping Coda.

5.3 Design Alternative 2: Iterative Re-execution Style

The key idea of the second design alternative is to support early propagation. A new command group is formed when the user issues a saving command. The group will serve as a unit of propagation and will be shipped to the surrogate for replaying. To maintain the in-memory state, replaying processes execute in the iterative style. This requires a more complicated mechanism on the file system.

The mechanism is not yet implemented. The reason is that simplicity is favored in this thesis work, so the first design alternative (the one using one-short re-execution style) was selected as the model for implementing the prototype. The more complicated mechanism will be implemented only when usage experience indicates that the simpler mechanism is not good enough.

Nevertheless, for completeness, a design of the more complicated mechanism is outlined in this section. It helps to identify the key features needed, and can serve as a roadmap for future work. Figure 5.13 shows an overview of the mechanism, and the following two sub-sections will discuss the design in detail.

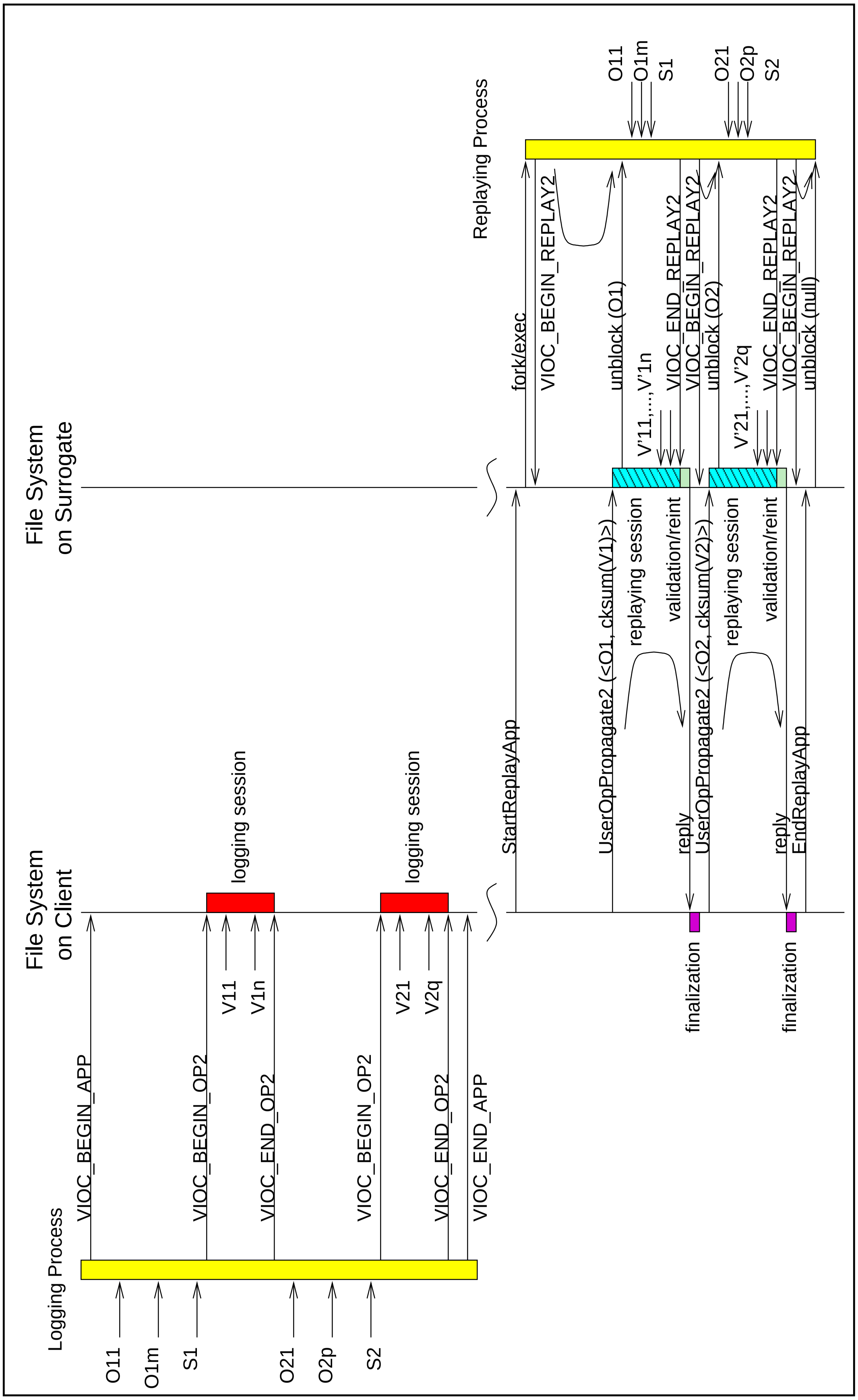

Figure 5.13: Overview of the Mechanism of Application-aware Operation Shipping Using the Second Design Alternative

```
VIOC_BEGIN_APP2  fs_begin_app2 (char *command, char **args,
                                char **env, char *cwd,
                                mode_t umask,
                                struct timeval *timeout,
                                short optimistic);
VIOC_END_APP2    fs_end_app2   ();
VIOC_BEGIN_OP2   fs_begin_op2  (char *srcfilename);
VIOC_END_OP2     fs_end_op2    ();
```

Figure 5.14: Logging Interface - Design Alternative 2

The left column of this figure shows the four new `ioctl` commands added to support operation logging in the mechanism of application-aware operation shipping using the second design alternative. The right column shows the four corresponding convenient functions and their arguments.

5.3.1 Logging

Figure 5.14 shows the new logging interfaces. This interfaces decouples application sessions from logging sessions, such that there can be multiple logging sessions per application session.

A logging process should make a `VIOC_BEGIN_APP2` command upon start up. The command is used to instruct the surrogate to set up a replaying process. Three groups of information are passed with the command: (1) the starting command (`command`); (2) the process environment – the command-line arguments (`args`), the environment-variable list (`envp`), the current working directory (`cwd`), and the file-creation mask (`umask`); and (3) the two policy arguments `timeout` and `optimistic`. The first policy argument controls how long the replaying process can be left idle – Venus on the surrogate will garbage collect any replaying process that has been idled for more than the timeout value, and the second policy argument controls whether the file system should continue to attempt operation shipping if some previous operations of the same application session have failed.

The logging process should make a `VIOC_END_APP2` command immediately before it terminates. The command instructs the surrogate to shutdown the replaying process.

When the logging process receives a saving command, it knows that it is about to make some mutating file-system calls. Therefore, it will carry out the following four steps. First, it constructs an application-specific operation log. On the log, all the application-specific commands of the current command group are stated. The log is physically a local file. Second, it starts a logging session with the file system using the `ioctl` command `VIOC_BEGIN_OP2`. The application-specific operation log is passed to the file system along with the command. Third, it executes the saving command. Since the file system is in a logging session, it can associate all the mutation file-system calls in the session with the command group. Fourth, when the saving command is done, the logging process ends the logging session using the `ioctl` command `VIOC_END_OP2`.

5.3.2 Shipping

Figure 5.15 shows the RPC interface exported by surrogates for supporting application-aware operation shipping. `StartReplayApp2` is used to set up the replaying process, `EndReplayApp2` is used to shut down the replaying process. During the application session of a replaying process, there can be multiple `UserOpPropagate2` RPCs, each requesting the replaying of a command group.

When a `StartReplayApp2` RPC arrives at a surrogate, Venus will spawn a child process for re-executing the specified application. The process will re-execute in the iterative style. That is, it will have a main idle loop. Most of the time, the process sits in the loop waiting for command groups to come. When a command group arrives, the process will replay the commands specified in the application-specific operation log. When it finishes the replaying, the process will return to the idle loop. (In contrast, a

```
StartReplayApp2 (IN RPC2_String    pathname,
                 IN RPC2_Integer   n_argv,
                 IN RPC2_CountedBS argv_buf,
                 IN RPC2_Integer   n_envp,
                 IN RPC2_CountedBS envp_buf,
                 IN RPC2_String    cwd,
                 IN mode_t         umask,
                 IN vuid_t         vuid,
                 IN VolId          VolumeId,
                 OUT pgid_t        pgid_replay,
                 OUT RPC2_Integer  uniqufier_replay);

UserOpPropagate2 (IN pgid_t             pgid_replay,
                  IN RPC2_Integer       uniqufier_replay,
                  IN RPC2_CountedBS     cksum_buf,
                  IN RPC2_Integer       ref_cml_size,
                  IN RPC2_Ingeger       app_oplog_size,
                  OUT ViceVersionVector updateSet);

EndReplayApp2 (IN pgid_t pgid_replay
               IN RPC2_Integer uniqufier_replay);
```

Figure 5.15: RPC Interface for Application-aware Operation Shipping - Design Alternative 2

A surrogate exports three RPCs (remote procedural calls) to the weakly-connected client. `StartReplayApp2` is used to set up a replaying process, which will re-execute in the iterative style, `EndReplayApp2` is used to shut down the replaying process. During an application session of a replaying process, there can be multiple `UserOpPropagate2` RPCs, each requesting the replaying of a command group.

```
VIOC_BEGIN_REPLAY2  fs_begin_replay2 (char *destfilename);
VIOC_END_REPLAY2    fs_end_replay2 ();
```

Figure 5.16: Interface for Replaying Operations - Design Alternative 2

The left column shows the two new added `ioctl` command added to support operation replaying for Type-2 user operations. The right column shows the two corresponding convenient functions and their arguments.

replaying process that re-executes in the one-shot style will terminate when it finishes the replaying of a command group.) The replaying process will terminate only when the RPC `EndReplayApp2` arrives.

The interface exported by Venus to the re-executing application is shown in Figure 5.16. Once started, a replaying process uses the command `VIOC_BEGIN_REPLAY2` to tell Venus that it is *ready* to accept a new command group for replay. It will be blocked until Venus has the information for the next command group (which is sent in from the client via the `UserOpPropagate2` RPC). When the replaying process is unblocked, an application-specific operation log will have been written to the file specified by `destfilename`, and Venus will have entered into a replaying session (Section 4.2.3. The process interprets the log, and replays the command stated on the log. The last command in the command group is a saving command, and it induces mutating file-system calls. Since Venus is in a replaying session, the effect of the mutating file-system calls will be associated with the current replaying session. Finally, the process finishes the replaying of the command group, it ends the replaying session by using the `VIOC_END_REPLAY2` command.

Upon the end of a replaying session, Venus will proceed into the validation, integration, and finalization phases as in the the case of application-transparent operation shipping.

5.4 Chapter Summary

This chapter studies the two research questions regarding application-aware operation shipping. The first question is on the architecture needed. We found that there are two design alternatives: one supports the re-execution in the one-shot style, and the other supports the re-execution in the iterative style. The two styles of re-execution have different implications on the granularities of update propagation: the former supports propagation-after-exit, but the latter supports early-propagation. We favour simplicity in this work and has adopted the first alternative.

The second question is on the programming effort required to extend an existing application to add the capability of operation logging and replaying. To this end, a popular image application – the GIMP – is chosen as a case study. A prototype of an extended GIMP has been implemented, in which about one sixth of the commands can be logged and replayed. The programming effort required has been found to be moderate.

Section 6.3 will present the performance gain of operation shipping using the extended GIMP as an example.

Chapter 6

Evaluation

Three realistic prototypes have been designed, implemented, and evaluated to demonstrate the feasibility and benefits of operation shipping. These prototypes are extended from three existing systems: the Coda File System, the Bourne Again Shell (`bash`), and the GIMP. The `bash` shell is chosen as an example interactive shell that works with Coda in application-transparent operation shipping; the GIMP is chosen as an example interactive application that works with Coda in application-aware operation shipping.

This chapter begins with a brief discussion on the implementation status of the prototypes, and then it discusses the quantitative results for application-transparent and application-aware operation shipping in the two subsequent sections.

6.1 Implementation Status

The prototype file system is an extension of Coda, and is maintained as a branch off the mainline code. By using the CVS system (Concurrent Versions System) [58], the new developments in the mainline branch [11] are merged into the prototype from time to time. The prototype was first developed based on the mainline version 4.2.4 in December 1997. The new developments in the mainline versions 4.4.3 (May 98), 4.6.3 (September 98), 5.2.2 (May 99), and 5.2.4 (June 99) are merged into the prototype subsequently on the dates indicated in the brackets.

The prototype is developed and tested on the Linux platform. However, no Linux-specific features are used in developing the prototype, so porting it to other platforms should be easy.

All the extensions needed for supporting operation shipping have been added only to Venus, the user-level cache manager on the client side. There have been no changes needed in neither the server nor the kernel.

The most primitive version of the prototype was functional in early 1998, but it blindly rejected all re-executions with non-repeating side effects. The problem was solved in summer of the same year. Also, the initial prototype supported only application-transparent operation shipping. Support for application-aware operation shipping using the one-shot re-execution style was added in early 1999. The interaction of operation shipping with cancellation optimization was addressed in summer 1999.

Operation shipping needs the co-operation between the file system and some entities. There are two cases. The first case is application-transparent operation shipping. The entity is an operation-logging interactive shell (the file system serves as the replaying entity). An extended version of the GNU Project's Bourne Again Shell (`bash`) was implemented in early 1998 to add the operation logging capability. The modification needed is very minor. Only a few lines of code are needed.

The second case is application-aware operation shipping. The entity is an interactive application that has operation-logging and operation-replaying capabilities. In this work, the GNU Image Manipulation Program (the GIMP) is chosen as an example application. An extended version of the GIMP was implemented in early 1999.

The source code of all the three prototypes can be downloaded from [2].

6.2 Application-transparent Operation Shipping

Test	Name	Nature	NF	Size (KB)	SE1	SE2
T1	rp2gen callback.rpc2	RPC2 stub generator	5	27.5	•	
T2	rp2gen adsrv.rpc2	RPC2 stub generator	5	76.3	•	
T3	yacc parsepdb.yacc	compiler compiling	1	23.5		
T4	c++ -c counters.cc -o counters.o	compiling	2	26.0		
T5	c++ -c pdlist.cc -o pdlist.o	compiling	2	62.4		
T6	c++ -c fso_daemon.cc -o fso_daemon.o	compiling	2	265.3		
T7	c++ parserecdump.o -o parserecdump	linking	1	23.0		
T8	ar rv libdir.a ...	library building	1	70.2	•	•
T9	ar rv libfail.a ...	library building	1	363.1	•	•
T10	tar xzvf coda-doc-4.6.5-3-ppt.tgz	extracting files	5	269.5		
T11	make coda (in coda-src/blurb)	compiling/linking	3	69.9		
T12	make coda (in coda-src/rp2gen)	compiling/linking	10	237.1		
T13	tar cvf update.tar ...	packaging files	1	60.2		
T14	sgml2latex guide.sgml	translator	1	41.8		
T15	sgml2latex rvm_manual.sgml	translator	1	270.0		
T16	latex usenix99.tex	text formatter	3	93.4	•	

Figure 6.1: Selected Tests and Applications for Application-transparent Operation Shipping

Sixteen tests were run using nine applications with real-life files. In this table, NF means number of files that were updated. Some of the applications exhibited non-repeating side-effects due to time stamps (SE1) and temporary files (SE2). They are marked by bullet points (•) in the table, and have to be handled by the novel techniques discussed in Section 4.3.

This section answers the following three questions related to application-transparent operation shipping:

1. *Can operation shipping be used transparently with common non-interactive applications?*

2. *What is the extent of network-traffic reduction that can be achieved by using operation shipping?*

3. *What is the extent of elapsed-time reduction that can be achieved by using operation shipping?*

The experimental setup will be described in the next sub-section. Answers to the above questions will follow in the subsequent three sub-sections.

6.2.1 Experimental Setup

The client, the surrogate, and the server machine used in the experiments were a Pentium 90MHz, a Pentium MMX 200MHz, and a Pentium 90MHz machine respectively. All three machines were running the Linux operating system (kernel version 2.0.35). The network between the surrogate and the server was a 10-Mbps Ethernet. The bandwidths of the client–surrogate network and the client–server network varied in different tests, and the Coda failure emulation package (`libfail` and `filcon`) [49] was used to emulate different network bandwidths on a 10-Mbps Ethernet.

Sixteen different tests were performed. In these tests, nine different common non-interactive applications were involved (Figure 6.1). The input files for each tests were real-life files found in our environment. The tests were selected such that the data size in each test was close to one of the three reference sizes: 16, 64, and 256 Kbytes. The data size is defined as the total size of the files updated by an operation. The 16 tests were labeled as $T1, T2, \cdots$, and $T16$ respectively.

6.2.2 Transparency to Applications

This thesis does not claim that operation shipping can be used transparently with *all* non-interactive applications. For example, it is anticipated that operation shipping probably cannot be used with the `-j <n>` mode of `GNU Make`, which runs `n` jobs in parallel. However, so far all the nine selected applications can be used transparently with operation shipping. Three of them exhibit non-repeating side effects, but these side effects can be handled by the techniques discussed in Section 4.3.

6.2.3 Network Traffic Reduction

Test	Nature	Traffic by value-shipping (Kbytes) L_v	Traffic by operation-shipping (Kbytes) L_{op}	Traffic reduction by operation-shipping L_v/L_{op}
T1	rp2gen	28.7	2.0	14.4
T2	rp2gen	77.5	1.9	40.8
T3	yacc	23.7	1.0	23.7
T4	c++ -c	27.1	1.9	14.3
T5	c++ -c	63.4	1.8	35.2
T6	c++ -c	266.3	2.0	133.2
T7	c++	23.9	2.0	12.0
T8	ar	70.2	1.9	36.9
T9	ar	364.0	2.2	165.5
T10	tar x	271.8	4.7	57.8
T11	make	71.6	2.3	31.1
T12	make	242.0	5.9	41.0
T13	tar c	60.2	1.0	60.2
T14	sgml2latex	42.0	1.0	42.0
T15	sgml2latex	270.3	1.1	245.7
T16	latex	94.1	1.4	67.2

Figure 6.2: Network Traffic Reductions by Application-transparent Operation Shipping

In column 5, the network traffic reduction factors, L_v/L_{op} are listed, where L_v and L_{op} is the network traffic by value shipping and by application-transparent operation shipping respectively.

6.2.3.1 Methodology

For each test, both value shipping and operation shipping were used to propagate the updated files, and the traffic volume of each case was measured. Both the file data and the overhead were included in the traffic. In particular, for operation shipping, all fields in the operation logs: command, command-line arguments, current working directory, environment list, file-creation mask, meta-data, fingerprints, and so on, were counted towards the traffic.

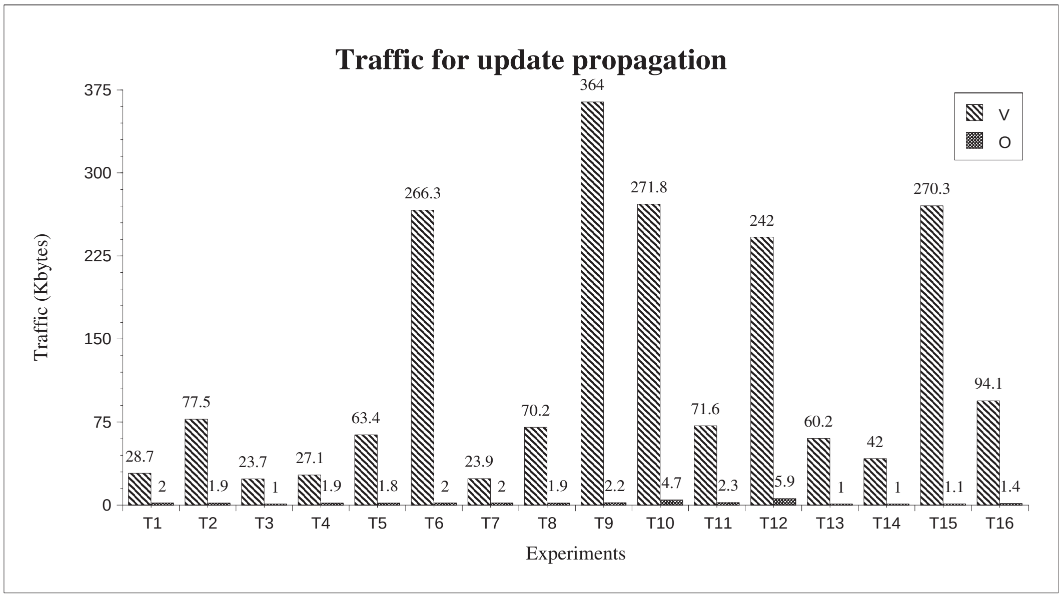

Figure 6.3: Network Traffic for Value Shipping and Application-transparent Operation Shipping

This figure depicts the data presented in Figure 6.2. There are two bars for each test. The left bar (in light gray color) indicates the network traffic for value shipping (V); the right bar (in dark gray color) indicates the network traffic for application-transparent operation shipping (O).

6.2.3.2 Results

In Figure 6.2, the traffic reduction is shown. Traffic reduction is defined as L_v/L_{op}, where L_v and L_{op} are the traffic volumes required for the update propagation by value shipping and by operation shipping respectively. The same data is graphically illustrated in Figure 6.3.

The traffic reductions achieved by operation shipping were very substantial. In 13 out of the 16 tests, the reduction exceeded a factor of 20. The highest reduction factor was 245.7 (T15); the smallest reduction was 12 (T7). In other words, operation shipping reduced the network traffic volumes by one to nearly three orders of magnitude.

The dramatic traffic reduction came from the fact that the traffic volumes required for operation shipping were all very small – from 1.0 Kbytes to 5.9 Kbytes.

Therefore, the larger the traffic volume required for value shipping, the larger the traffic reduction factor that can be achieved by operation shipping. These results confirm the observation stated in Chapter 1 of this thesis: “Big file, small operation.” (Section 1.5.1)

6.2.4 Reduction of Elapsed Time

The previous section has shown that operation shipping can achieve very substantial traffic reduction. However, since operation shipping involves replaying of user operation and computation of fingerprint and forward-error-correction code, there are some computational overheads involved. It is interesting to see if the elapsed time of reintegration is also reduced correspondingly.

6.2.4.1 Methodology

For each test, both value and operation shippings were used to propagate the updated files, and the elapsed time of reintegration of each case was measured. The elapsed time is the time to complete the respective procedures for reintegration: `IncReintegrate` or `PartialReintegrate` for value shipping, and `IncReintegrateViaSurrogate` for operation shipping (Section 4.2.2). For the latter, the elapsed time comprises the time for shipping the operation log, re-executing the operation, and other overheads, such as checking the fingerprints and error correction. Since the elapsed time depends heavily on the network bandwidth, it was measured under three different network bandwidths: 9.6, 28.8, and 64.0 kilobits per second. Each test was repeated three times.

6.2.4.2 Results

Figure 6.4 shows the elapsed time for value shipping (T_v) and operation shipping (T_{op}) under the three different network conditions. Figure 6.5 shows the speedup, which is

Test	Nature	Data size (Kbytes)	Elapsed time (msecs) 9.6-Kbps T_v	9.6-Kbps T_{op}	28.8-Kbps T_v	28.8-Kbps T_{op}	64-Kbps T_v	64-Kbps T_{op}
T1	rp2gen	27.5	27,921 (312)	8,282 (73)	9,666 (8)	6,404 (50)	4,539 (20)	5,637 (37)
T2	rp2gen	76.3	71,937 (27)	9,322 (61)	24,294 (39)	7,358 (9)	11,416 (133)	6,706 (90)
T3	yacc	23.5	22,025 (31)	3,215 (60)	7,563 (13)	2,364 (34)	3,506 (0)	2,049 (9)
T4	c++ -c	26.0	25,112 (64)	5,098 (31)	8,683 (38)	3,491 (88)	4,164 (176)	2,928 (107)
T5	c++ -c	62.4	59,144 (254)	7,546 (48)	19,899 (51)	5,927 (12)	9,591 (93)	5,377 (43)
T6	c++ -c	265.3	257,143 (23,989)	15,645 (82)	88,274 (8,418)	13,877 (92)	39,167 (233)	13,181 (9)
T7	c++	23.0	22,218 (27)	4,425 (27)	7,637 (12)	2,874 (30)	3,599 (12)	2,297 (20)
T8	ar	69.3	65,473 (58)	5,613 (21)	22,059 (77)	4,104 (25)	10,571 (343)	3,646 (138)
T9	ar	363.1	345,241 (24,172)	13,143 (142)	118,929 (7,402)	11,634 (74)	55,725 (3,472)	10,944 (91)
T10	tar x	269.5	247,674 (327)	12,825 (156)	85,041 (276)	9,448 (96)	39,954 (182)	8,448 (60)
T11	make	69.9	67,113 (580)	8,839 (79)	22,723 (354)	6,793 (25)	10,633 (142)	6,115 (30)
T12	make	237.1	224,135 (2,452)	22,272 (132)	77,279 (73)	18,098 (39)	36,256 (293)	17,085 (396)
T13	tar c	60.0	55,355 (36)	3,674 (8)	18,826 (33)	2,978 (92)	8,802 (7)	2,602 (74)
T14	sgml2latex	41.8	38,602 (15)	5,433 (18)	13,160 (12)	4,648 (25)	6,209 (87)	4,439 (235)
T15	sgml2latex	270.0	245,709 (266)	13,780 (103)	83757 (162)	12852 (42)	39414 (32)	12600 (107)
T16	latex	93.4	86,619 (30)	8,522 (646)	29,429 (54)	7,194 (669)	13,869 (49)	6,243 (39)

Figure 6.4: Elapsed Time for Value Shipping and Application-transparent Operation Shipping.

Elapsed time, in milliseconds, for update propagation using value shipping (T_v) and application-transparent operation shipping (T_{op}) under three different network conditions. Figures are means of three runs. Figures in parentheses are the standard deviations.

defined to be the ratio T_v/T_{op}. Figure 6.6 depicts the same information in graphical form.

The speedups obtained by operation shipping were substantial. They were the most substantial in the 9.6-Kbps network. Eight out of the 16 tests were accelerated by a factor exceeding 10. The maximum speedup was 26.3 (T9); the minimum speedup was 3.4 (T1). In the other two networks, the speedups ranged from a factor of 1.4 (T4 and T14, 64-Kbps) to 10.2 (T9, 28.8-Kbps). (There was one exception: test T1 was slowed down when using operation shipping at 64 Kbps.)

Beside speeding up propagation, operation shipping has another advantage. That is the elapsed time of update propagation is much less sensitive to the network condition than that of value shipping. This can be seen from the elapsed-time–bandwidth curves. Figures 6.7 and 6.8 show the curves for T1 and T9 respectively. In both figures, the curves for value shipping are steep (sensitive to bandwidth) whereas the curves for operation shipping are flat (not sensitive to bandwidth). These two tests are particularly interesting since they are the tests with the minimum and the maximum speedup respectively. The curves for other tests show similar trends.

Test	Nature	Data size (Kbytes)	Speedup (T_v/T_{op}) 9.6 Kbps	28.8 Kbps	64 Kbps
T1	rp2gen	27.5	3.4	1.5	0.8
T2	rp2gen	76.3	7.7	3.3	1.7
T3	yacc	23.5	6.9	3.2	1.7
T4	c++ -c	26.0	4.9	2.5	1.4
T5	c++ -c	62.4	7.8	3.4	1.8
T6	c++ -c	265.3	16.4	6.4	3.0
T7	c++	23.0	5.0	2.7	1.6
T8	ar	69.3	11.7	5.4	2.9
T9	ar	363.1	26.3	10.2	5.1
T10	tar x	269.5	19.3	9.0	4.7
T11	make	69.9	7.6	3.3	1.7
T12	make	237.1	10.1	4.3	2.1
T13	tar c	60.0	15.1	6.3	3.4
T14	sgml2latex	41.8	7.1	2.8	1.4
T15	sgml2latex	270.0	17.8	6.5	3.1
T16	latex	93.4	10.2	4.1	2.2

Figure 6.5: Speedups for Update Propagation by Using Application-transparent Operation Shipping - Table

Speedups for update propagation under three different network speeds: 9.6 Kbps, 28.8 Kbps, and 64 Kbps. Speedup is defined as the ration T_v/T_{op}.

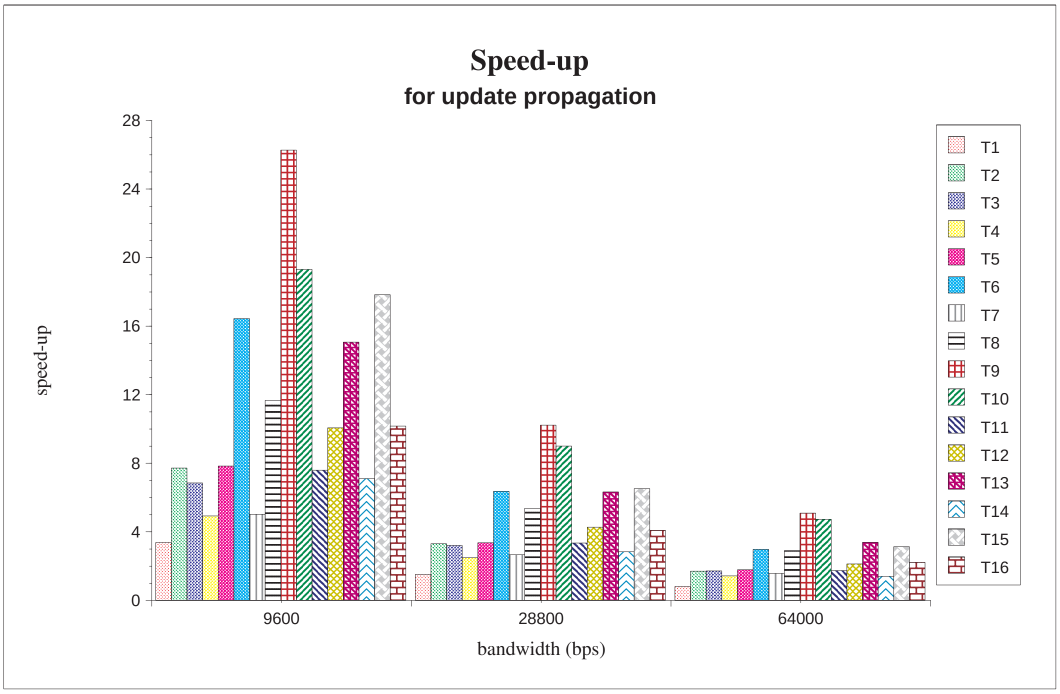

Figure 6.6: Speedups for Update Propagation by Using Application-transparent Operation Shipping - Graph

This figure illustrates the data presented in Figure 6.5.

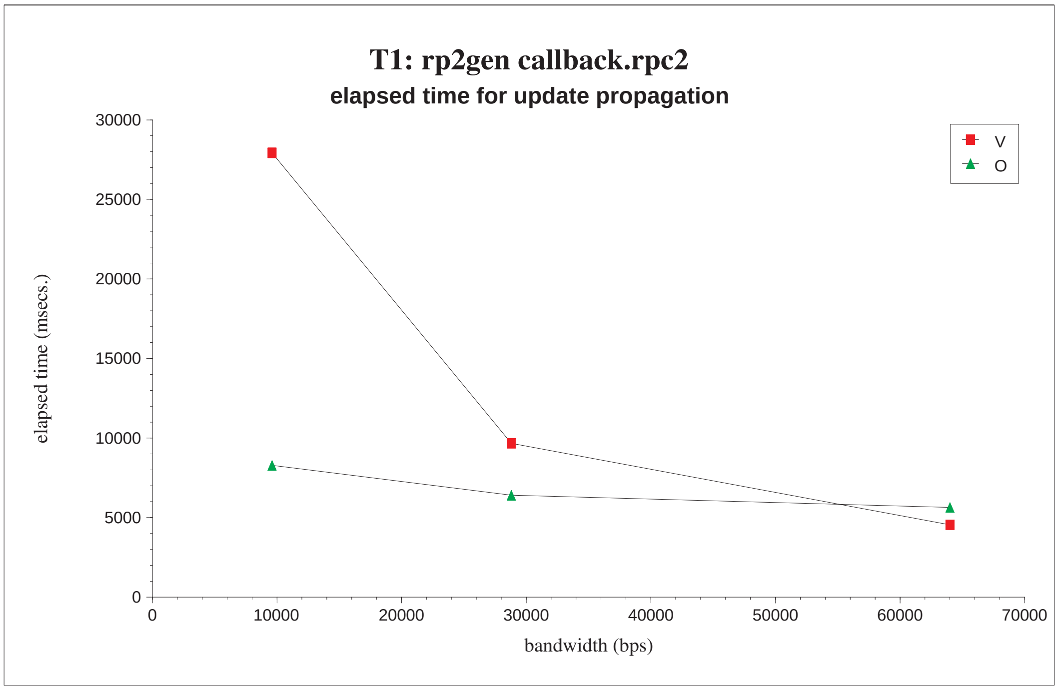

Figure 6.7: Elapsed Time vs. Bandwidth for Test T1

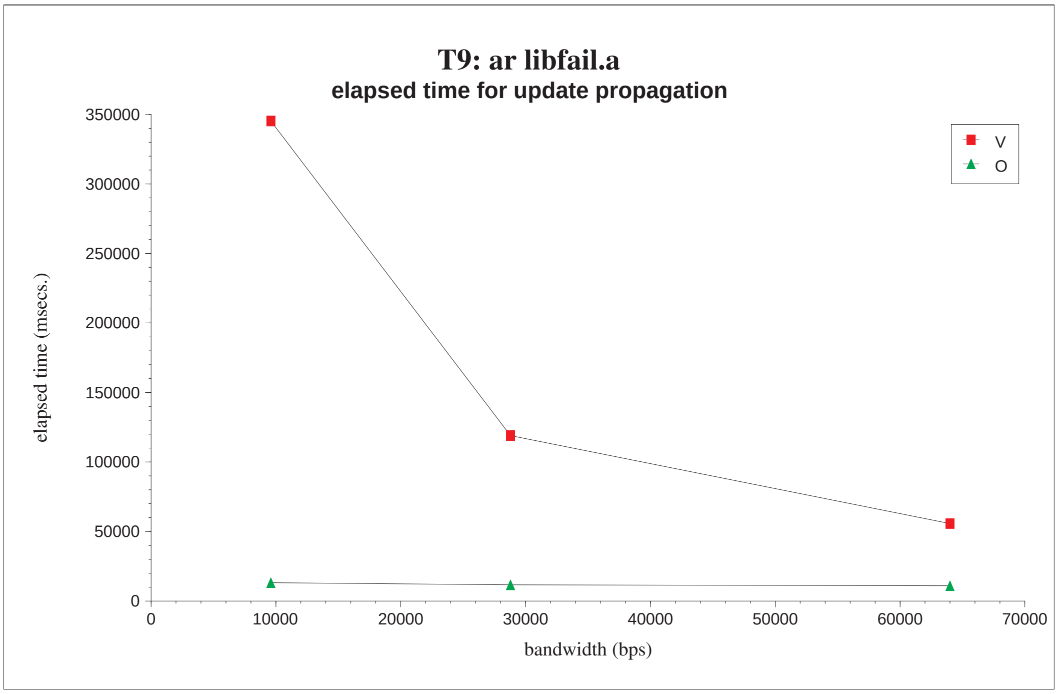

Figure 6.8: Elapsed Time vs. Bandwidth for Test T9

6.3 Application-aware Operation Shipping

This section answers the following three questions related to application-aware operation shipping using the GIMP as an example application:

1. Can operation shipping be used with common user operations that a user may perform with the GIMP?

2. What is the extent of network-traffic reduction that can be achieved by using operation shipping?

3. What is the extent of elapsed-time reduction that can be achieved by using operation shipping?

These three questions are very similar to those posed in the previous section. However, the previous section studies these questions with application-transparent operation shipping; whereas this section studies the questions with application-transparent operation shipping.

The experimental setup will be described in the next sub-section. Answers to the above questions will follow in the subsequent three sub-sections.

6.3.1 Experimental Setup

Both the hardware and the software have been slightly upgraded when compared with those used in Section 6.2. This is because the two sets of experiments were performed at different periods of time. In this set of experiments, the client, the surrogate, and the server machine used were a Pentium MMX 200MHz, a Pentium II 300MHz, and a Pentium 90MHz machine respectively. All three machines were running the Linux operating system (the client and the surrogate were using kernel version 2.2.5, and the server were using kernel version 2.0.34).

Like in the tests described in Section 6.2, the network between the surrogate and the server was a 10-Mbps Ethernet. The bandwidths of the client–surrogate network and the client–server network varied in different tests, and the Coda failure emulation package (`libfail` and `filcon`) [49] was used to emulate different network bandwidths on a 10-Mbps Ethernet.

Eleven tests, labelled as $T30, T31, \ldots,$ and $T40$ were selected for evaluating the performance. They represent some tasks that a user may perform with the GIMP, and each of them comprises a number of GIMP-specific interactive commands. Figures 6.9 and 6.10 give the details of the tests.

6.3.2 Applicability of Operation Shipping

Operation shipping is applicable for a user operation when either of the following two conditions is true: (1) the user operation, when replayed on the surrogate, is repeating (i.e., the replayed operation is producing exactly the same values as its original execution); or (2) the replayed user operation is not repeating, but the replaying discrepancy can be fixed by some handling techniques (Section 3.4).

Among the eleven tests that were selected for evaluating the performance of application-aware operation shipping, it was found that operation shipping cannot be used with $T40$. This is because one of the function "*blur*" used by the test needs a random seed for the blurring algorithm, and it is implemented in such a way that it gets the random seed from the current time value. Upon replaying, the blur function gets a different current time value and thus a different random seed. So it is not repeating on the surrogate. Unlike the non-repeating time stamp that was studied in Section 4.3.1, the random seed has a global effect on the whole image, so the replaying discrepancy cannot be fixed by techniques such as forward error correction.

Note that the requirements for application transparency are different in the two types of operation shipping. For application-transparent operation shipping, the applications are not expected to be modified to cater for the need of operation shipping; whereas for application-aware operation shipping, the applications are

Test	Name and Description	Size (KB)	NR
T30	*Embossment:* load korea7.jpg (157.7 KB), invoke emboss plug-in (azimuth=73.4, elevation=57.7, depth=20) save as t30.jpg	243.8	
T31	*Annotation:* load gibralter.jpg (90.3 KB), add a text string "Gibralter", move the text string layer to the lower right corner of the image, anchor the text layer, save as t31.jpg	184.6	
T32	*Color inversion:* load france1.neg.jpg (128.6 KB), invert the colors of the image, save as t32.jpg	128.7	
T33	*Color adjustment:* load monterey.jpg (240.9 KB), change color balance (Cyan-Red=52, Magenta-Green=20, Yellow-Blue=0), change brightness and contrast (brightness=9, contrast=60), save as t33.jpg	260.9	
T34	*Conversion to the BMP format:* load coppermt.jpg (129.5 KB), save as t34.bmp	951.6	
T35	*Gradient Map:* load hk003.jpg (80.1 KB), choose "Default" as the active gradient map, map the contents of the image with the active gradient map, save as t35.jpg	118.2	

Figure 6.9: Selected Tests for Application-aware Operation Shipping (to be continued)

This and the next table list the eleven tests selected for evaluating the performance for application-aware operation shipping. The column labelled "Size" gives the output file size of the test. A bullet point (•) in the column labelled "NR" indicates that the replayed user operation is not repeating on the surrogate.

Test	Name and Description	Size (KB)	NR
T36	*Canvas Effect:* load france15.jpg (155.8 KB), add a canvas texture map to the image (direction="Top-right", depth=4), save as t36.jpg	305.2	
T37	*Mosaic Effect:* load morocco3.jpg (88.4 KB), convert the image into a collection of tile (tile_size=15.0, tile_height=4.0, tile_spacing=1.0, tile_neatness=0.65, light_dir=135.0, color_variation=0.2, anti-aliasing, color averaging, hexagon tiles, smooth surface, black/white grout color), save as t37.jpg	284.2	
T38	*Oil Painting Effect:* load kuleuven1.jpg (216.7 KB), modify the image to resemble an oil painting (mask_size=7, RGB algorithm), save as t38.jpg	350.2	
T39	*Poster Effect:* load sjapan2.jpg (109.7 KB), posterize the image (levels=3), save as t39.jpg	71.0	
T40	*Blurring:* load swiss8.jpg (109.8 KB) blur the image by applying a 3x3 blurring convolution kernel, to the image (randomization percentage=50, repeat count=1), save as t40.jpg	58.3	•

Figure 6.10: Selected Tests for Application-aware Operation Shipping (continued)

The previous and this table list the eleven tests selected for evaluating the performance for application-aware operation shipping. The column labelled "Size" gives the output file size of the test. A bullet point (•) in the column labelled "NR" indicates that the replayed user operation is not repeating on the surrogate.

already modified to add operation logging and replaying capabilities. Therefore, the solution to the problem of non-repeating blur function is simple: the function can be modified to have a repeating behavior. This can be done by having an option for the invoker of the function to supply the random seed (via the PDB interface). Once the function is modified, the logging GIMP can logged the random seed used in the original execution, and the replaying GIMP can supply the logged value of random seed when it is replaying the blur function on the surrogate.

The needed modification for the blur function is conceptually straightforward but has not been carried out. Test $T40$ was thus excluded from the test set. In the following performance-evaluation experiments, only Test $T30$ - $T39$ were used.

6.3.3 Network Traffic Reduction

6.3.3.1 Methodology

For each test, both value and operation shippings were used to propagate the update files, and the traffic volume of each case was measured. Both the file data and the overhead were included in the traffic. In particular, for operation shipping, the application-specific operation log and all fields in the operation log (command, command-line arguments, current working directory, environment list, file-creation mask, meta-data, fingerprints, and so on) were counted towards the traffic.

6.3.3.2 Results

In Figure 6.11, the traffic reduction is shown. Traffic reduction is defined as L_v/L_{op}, where L_v and L_{op} are the traffic volumes required for the update propagation by value shipping and by application-aware operation shipping respectively. The same data are graphically illustrated in Figure 6.12.

Similar to the experiments for application-transparent operation shipping (Section 6.2.3.2), this set of experiments found that very substantial traffic reductions were achieved by application-aware operation shipping. The reductions all exceeded a factor of 30. The highest reduction factor was 396.6 ($T34$); the smallest reduction factor was 33.9 ($T39$). In other words, operation shipping reduced the network traffic volumes by one to nearly three orders of magnitude.

Like its application-transparent counterpart, application-aware operation shipping achieved the dramatic traffic reductions because the traffic volumes required for operation shipping were so small. In these experiments, they ranged from 2.1 Kbytes to 2.4 Kbytes. Therefore, the larger the traffic volume required for value shipping, the larger the traffic reduction factor that can be achieved by operation shipping.

Test	Name	Traffic by value-shipping (Kbytes) L_v	Traffic by operation-shipping (Kbytes) L_{op}	Traffic reduction by operation-shipping L_v/L_{op}
T30	Embossment	244.0	2.1	116.2
T31	Annotation	184.8	2.3	80.3
T32	Color inversion	128.9	2.1	61.4
T33	Color adjustment	261.1	2.3	113.5
T34	BMP conversion	951.8	2.4	396.6
T35	Gradient Map	118.4	2.1	56.4
T36	Canvas	305.4	2.2	138.8
T37	Mosaic	284.4	2.3	123.7
T38	Oil Painting	350.4	2.2	159.3
T39	Poster	71.2	2.1	33.9

Figure 6.11: Network Traffic Reductions by Application-aware Operation Shipping

In column 5, the network traffic reduction factors, L_v/L_{op} are listed, where L_v and L_{op} is the network traffic by value shipping and by application-aware operation shipping respectively.

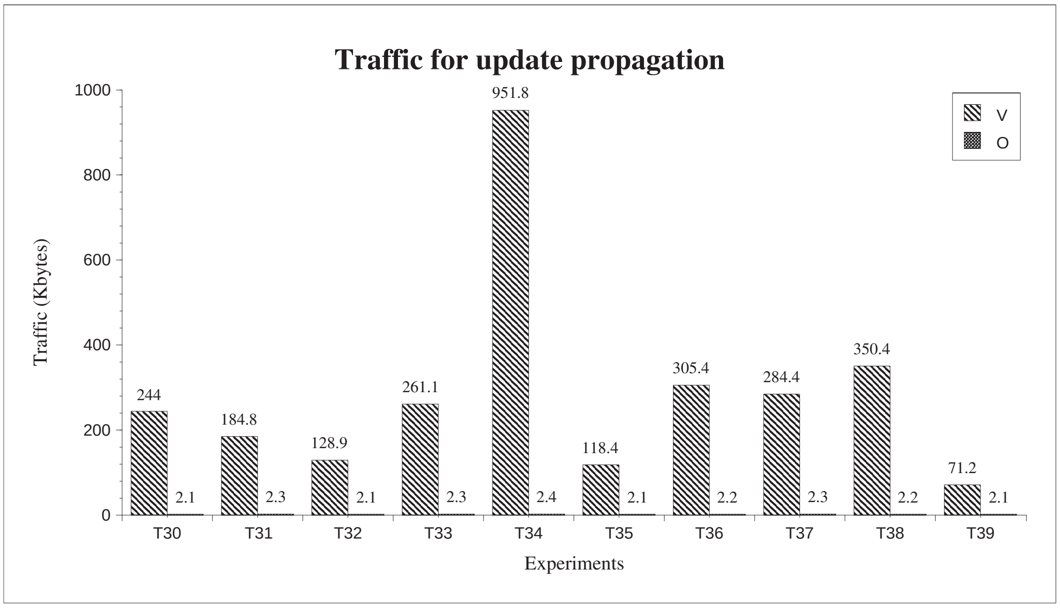

Figure 6.12: Network Traffic for Value Shipping and Application-aware Operation Shipping

This figure depicts the data presented in Figure 6.11. There are two bars for each test. The left bar (in light gray color) indicates the network traffic for value shipping (V); the right bar (in dark gray color) indicates the network traffic for application-aware operation shipping (O).

6.3.4 Reduction of Elapsed Time

The previous section studies the traffic reduction of application-aware operation shipping; this section studies the elapsed time reduction. Note that the user operations in this series of experiments are the invocation of some image-manipulation functions, which in general are quite computationally intensive. Therefore, it is interesting to see if shipping these operations still give performance improvements over value shipping in term of the elapsed time of update propagation.

6.3.4.1 Methodology

For each test, both value shipping and application-aware operation shipping were used to propagate the updated files, and the elapsed time of reintegration of each case was measured. The elapsed time is the time to complete the respective procedures for reintegration: `IncReintegrate` or `PartialReintegrate` for value shipping, and `IncReintegrateViaSurrogate` for operation shipping (Section 4.2.2). For the latter, the elapsed time comprises the time for shipping the operation log, the creation of the re-executing GIMP process, the replaying of the user operations, and other overheads, such as checking the fingerprints and error correction. Since the elapsed time depends heavily on the network bandwidth, it was measured under three different network bandwidths: 9.6, 28.8, and 64.0 kilobits per second. Each test was repeated three times.

6.3.4.2 Results

Figure 6.13 shows the elapsed time for value shipping (T_v) and operation shipping (T_{op}) under the three different network conditions. Figure 6.14 shows the speedup, which is defined to be the ratio T_v/Top. Figure 6.15 depicts the same information in graphical form.

Likes its application-transparent counterpart (Section 6.2.4.2), application-aware operation shipping achieved very substantial speedups in the experiments. The speedups were the most substantial in the 9.6-Kbps network. Eight out of the ten tests were accelerated by a factor exceeding 10. The maximum speedup was 48.8 (T34); the minimum speedup was 8.8 (T39). In the other two networks, the speedups ranged from a factor of 1.7 (T30, 64-Kbps) to 21.9 (T34, 26.8-Kbps).

Also, application-aware operation shipping has the advantage that the elapsed time of update propagation is much less sensitive to the network condition than that of value shipping. Figures 6.16 and 6.17 show the elapsed-time–bandwidth curves for T30 and T34 respectively. From both figures, we can see that the curves for value shipping are steep (sensitive to bandwidth) whereas the curves for operation shipping are flat (not sensitive to bandwidth). T30 and T34 are the two tests with the minimum and the maximum speedup respectively. The curves for other tests show similar trends.

Test	Name	Data size (Kbytes)	Elapsed time (msecs) 9.6-Kbps T_v	9.6-Kbps T_{op}	28.8-Kbps T_v	28.8-Kbps T_{op}	64-Kbps T_v	64-Kbps T_{op}
T30	Embossment	243.8	226,487	22,852	76,640	21,662	34,285	20,300
			(613)	(52)	(325)	(353)	(74)	(2,028)
T31	Annotation	184.6	171,485	8,333	58,267	6,767	25,927	6,064
			(471)	(562)	(890)	(305)	(23)	(570)
T32	Color inversion	128.7	119,853	7,650	40,566	6,223	18,090	5,701
			(84)	(560)	(261)	(285)	(3)	(299)
T33	Color adjustment	260.9	242,607	11,757	82,303	10,395	36,679	10,377
			(177)	(590)	(383)	(568)	(65)	(273)
T34	BMP conversion	951.6	889,491	18,228	317,380	14,467	134,009	14,587
			(11,550)	(1,550)	(32,531)	(16)	(608)	(1,172)
T35	Gradient Map	118.2	113,546	7,830	37,278	6,253	16,670	5,728
			(5,198)	(591)	(468)	(591)	(67)	(583)
T36	Canvas	305.2	286,422	10,052	95,924	8,418	42,881	8,024
			(5,124)	(315)	(150)	(269)	(65)	(489)
T37	Mosaic	284.2	263,713	16,876	90,062	16,013	39,888	15,646
			(356)	(306)	(380)	(870)	(9)	(286)
T38	Oil painting	350.2	324,745	19,519	110,228	18,051	49,167	17,417
			(156)	(41)	(23)	(300)	(44)	(154)
T39	Poster	71.0	66,179	7,506	22,531	5,788	10,059	5,260
			(144)	(281)	(69)	(14)	(86)	(7)

Figure 6.13: Elapsed Time for Value Shipping and Application-aware Operation Shipping.

Elapsed time, in milliseconds, for update propagation using value shipping (T_v) and application-aware operation shipping (T_{op}) under three different network conditions. Figures are means of three runs. Figures in parentheses are the standard deviations.

Test	Name	Data size (Kbytes)	Speedup (Tv/T_{op}) 9.6 Kbps	28.8 Kbps	64 Kbps
T30	Embossment	243.8	9.9	3.5	1.7
T31	Annotation	184.6	20.6	8.6	4.3
T32	Color inversion	128.7	15.7	6.5	3.2
T33	Color adjustment	260.9	20.6	7.9	3.5
T34	BMP conversion	951.6	48.8	21.9	9.2
T35	Gradient Map	118.2	14.5	6.0	2.9
T36	Canvas	305.2	28.5	11.4	5.3
T37	Mosaic	284.2	15.6	5.6	2.5
T38	Oil painting	350.2	16.6	6.1	2.8
T39	Poster	71.0	8.8	3.9	1.9

Figure 6.14: Speedups for Update Propagation by Using Application-aware Operation Shipping - Table

Speedups for update propagation under three different network speeds: 9.6 Kbps, 28.8 Kbps, and 64 Kbps. Speedup is defined as the ratio T_v/T_{op}.

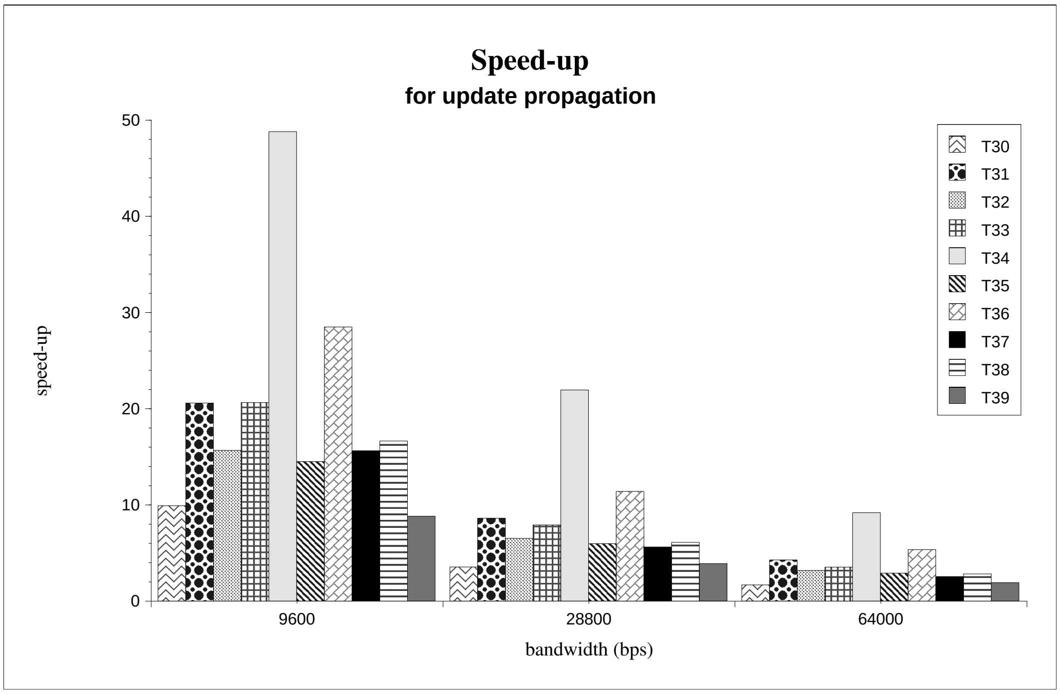

Figure 6.15: Speedups for Update Propagation by Using Application-aware Operation Shipping - Graph

This figure illustrates the data presented in Figure 6.14.

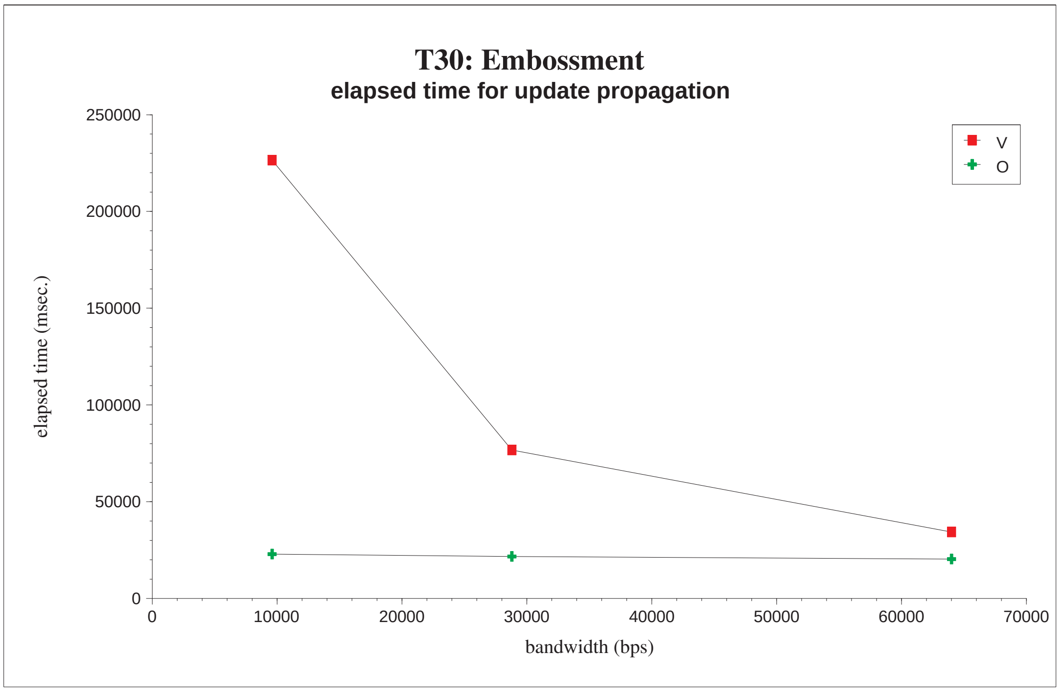

Figure 6.16: Elapsed Time vs. Bandwidth for Test T30

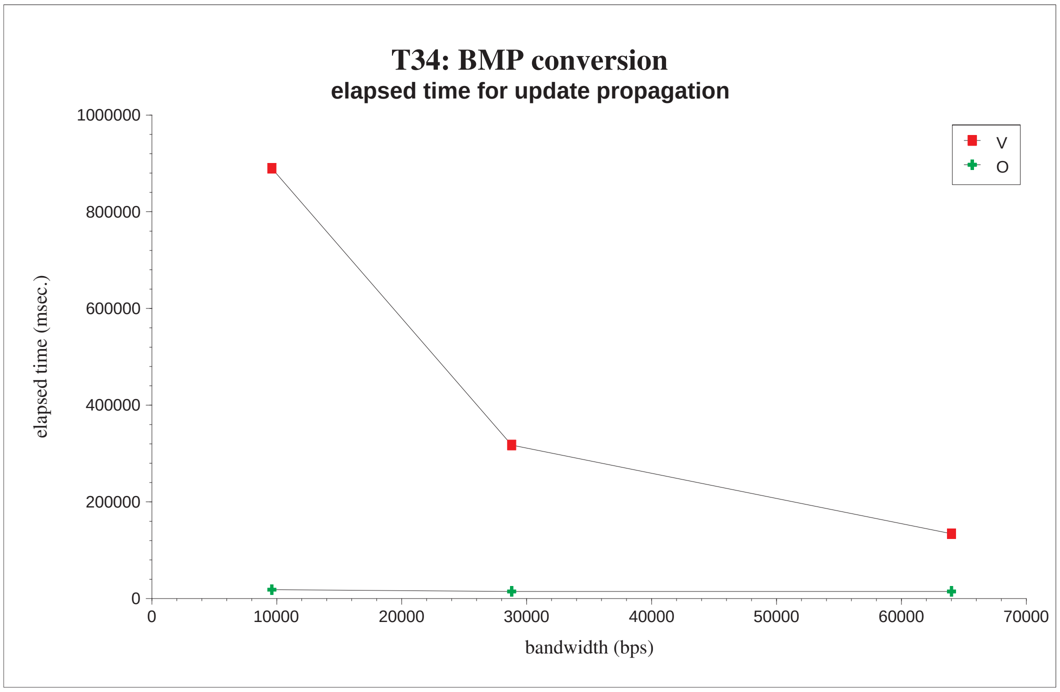

Figure 6.17: Elapsed Time vs. Bandwidth for Test T34

6.4 Chapter Summary

This chapter presents the implementation status of the prototypes. It also presents the quantitative evaluations of application-transparent and application-aware operation shippings. The evaluations were based on controlled experiments. They demonstrated the huge performance gain that can be achieved by both types of operation shippings. The performance gain has three aspects:

- The network traffic required for shipping an update is dramatically reduced.
- The elapsed time required for shipping an update is substantially reduced.
- The elapsed time required for shipping an update is much less sensitive to the network condition.

The first aspect means the mobile file system will be less intrusive to weak networks. The scarce network bandwidth of the weak networks can be saved for other uses, such as the World-Wide-Web or electronic mails. The second aspect means a smaller reintegration latency. The third aspect means the file system can hide from the users more effectively the unpleasant facts of the weak network conditions.

Chapter 7

Related Work

To the best of my knowledge, this is the first work that attempts to propagate file updates by operations. However, some techniques used in this work resemble the techniques that were used in some previous research works. This chapter discusses these techniques in Section 7.1.

This thesis addresses the issue of how to propagate large updated files across a weak network. The proposed solution is not the only possible one to address the issue. Section 7.2 discusses some of the alternative solutions.

7.1 Related Work

7.1.1 Use in Databases

The idea of operation-based update propagation has been used in the database community [36]. This dissertation is distinctive in the sense that the same idea is being applied in a different context: distributed file systems. There are three main differences. First of all, logging and shipping of operations in our case has to be done at a level higher than the low-level file system operations (such as `write`), which are not compact enough and are not appropriate for operation shipping. Therefore, a co-operation between the file system and a logging entity in the user space is needed. Second, several new concepts are required in the context of distributed file system: replaying of user operations on the surrogate, adjustment of status information, validation of replayed operations, and the handling of non-repeating side effects, etc. Third, in our system, operation shipping is only one of the mechanism for update propagation, and the existing mechanism of value shipping are retained as a fallback mechanism. Therefore, it can attempt operation shipping more boldly – if there is a failure, the file system can just fall back to value shipping.

7.1.2 Directory Operations

Prior to this work, logging and shipping of directory operations have been implemented in Coda [53, 52]. In fact, the Coda client-modify log can be viewed as an operation log for directories and a value log for files. When a directory is updated on a Coda client (e.g., a new entry is inserted), instead of shipping the whole new directory to the server, the client ships only the update operation (e.g., the insertion operation). Directory operations are more like database operations, since they can be mapped directly to insertion, deletion, and modification of directory entries. This work is distinctive, since it extends the idea of operation logging and shipping to also file updates. Also, the operations of concern are no longer file-system operations but user operations.

7.1.3 Re-executions

Several previous research projects have made extensive uses of re-executions, such as for fault tolerance [6] and load balancing [8]. In the former case, a process P can be backed up by another process P_b. If P crashes, then P_b will repeat the execution of P from a recent checkpoint, and will thereafter assume the role of P. In the latter case, a process can migrate to another host to reduce the load imposed on the original host. In this work, re-executions are used to re-produce some file modifications that are identical to those produced by the original executions.

7.1.4 Isolation-Only Transactions

A previous Coda project has implemented a mechanism for re-execution of operations [24] [25]. It addresses the update conflicts that may happen in optimistically controlled replicas. It proposes that a user can declare a portion of execution as an Isolation-Only Transaction (IOT). If an update conflict happens, Coda will re-execute the transaction. This work focuses on performance improvement rather than transactional guarantee. Also, in this work, re-executions take place in a different host (the surrogate), whereas re-execution of IOTs take place in the same host. This implies that we must handle the case when a re-execution does not produce the same result as the original execution.

7.2 Alternative Solutions

7.2.1 Delta Shipping

The idea is to ship only the incremental difference, which is also called the *delta*, between different versions of a file. It has been proposed by many people and is currently being used as a general mechanism [60] or in specific systems including file systems [14], web proxies [29], file archives [26], and source-file repositories [58, 40].

It is possible to compute deltas not only for text files but also for binary files. For example, the `rsync` algorithm [60] is a binary-delta algorithm. When shipping a file, the sending host suppresses the shipping of some blocks of data if they are found to be present on the receiving host already. It determines whether they are already present on the receiving host by using the checksum information supplied by the receiving host. The algorithm exploits a rolling checksum algorithm so that the blocks being matched can be started at any offset, not just multiples of block size.

Delta shipping has several limitations. First, a newly-created file has no previous version. Second, the effectiveness of delta shipping largely depends on how similar the two versions of a file are, and how those incremental differences are distributed in the file. In pathological case, a slightly changed file may need a huge delta. This can happen, for example, when there is an global substitution of string in a text file, or when there is a global brightness or contrast adjustment in an image file. In general, we believe operation shipping can achieve a larger reduction of network traffic.

On the other hand, delta shipping does not involve re-execution of applications and pre-arrangement of surrogate clients, as operation shipping does. Therefore, it is simpler in terms of system administration. I believe delta shipping and operation shipping can complement each other in a distributed file system. In particular, delta shipping can be used when the file system has to use value shipping, probably because of a lack of user-operation information or a replayed user operation is rejected.

7.2.2 Data Compression

Data compression reduces the size of a file by taking out the redundancy in the file. This technique can be used in a file system [14, 4] or a web proxy [29]. However, the reduction factors achieved by data compression may be smaller than that of operation shipping. The following is a comparison of the traffic reductions that can be achieved by operation shipping and data compression.

Methodology. The `gzip` utility, which uses the Lempel-Ziv coding (LZ77), was chosen as a representative data-compression implementation. It was used to run against the updated files of the 16 tests in Section 6.2. The total size of the compressed files gave the estimated traffic volume of value shipping with data compression $L_{v,gz}$. By dividing the uncompressed total file size L_v by $L_{v,gz}$, an estimated traffic reduction by data compression can be obtained.

Result and Discussion. Figure 7.1 compares the traffic reduction by operation shipping (L_v/L_{op}) with the estimated traffic reduction by data compression ($L_v/L_{v,gz}$).

Test	Nature	Traffic by value shipping (Kbytes) L_v	Traffic reduction by operation shipping L_v/L_{op}	Estimated traffic reduction by data compression $L_v/L_{v,gz}$
T1	rp2gen	28.7	14.4	4.8
T2	rp2gen	77.5	40.8	8.1
T3	yacc	23.7	23.7	4.4
T4	c++ -c	27.1	14.3	3.4
T5	c++ -c	63.4	35.2	3.5
T6	c++ -c	266.3	133.2	3.7
T7	c++	23.9	12.0	2.8
T8	ar	70.2	36.9	3.2
T9	ar	364.0	165.5	4.6
T10	tar x	271.8	57.8	3.8
T11	make	71.6	31.1	2.8
T12	make	242.0	41.0	3.0
T13	tar c	60.2	60.2	6.0
T14	sgml2latex	42.0	42.0	3.0
T15	sgml2latex	270.3	245.7	3.8
T16	latex	94.1	67.2	2.7

Figure 7.1: Comparing the Traffic Reduction by Operation Shipping and Data Compression

This table compares the traffic reductions that can be achieved by operation shipping with that are estimated to be achieved by data compression. The 16 tests are the same as those listed in Figure 6.1. The fourth column, traffic reduction by operation shipping, is obtained from Figure 6.2.

The former is obtained from the result in Section 6.2.3, and the latter is obtained using the methodology stated above.

The estimated traffic reductions by data compression ranged from 2.7 to 8.1, substantially smaller than that achieved by operation shipping, which ranged from 12.0 to 245.7. This is not surprising, since, as we know, operation shipping exploits the semantic information of the user operations, whereas data compression operates only generically on the files. Therefore, it is very natural that the former can achieve higher reduction than the latter.

Nevertheless, like delta shipping, data compression has the advantages that it does not involve re-executions of applications and pre-arrangements of surrogate clients. Therefore, it can complement operation shipping, and be used to enhance the value shipping mechanism in a file system.

7.2.3 Logging Keystrokes

A file system may log keystrokes and mouse clicks, ship them, and replay them on the surrogate. As such, it may be transparent to an application even if the application is interactive. However, I am pessimistic about this approach, because it is very difficult to make sure the logged keystrokes and mouse clicks will produce the identical outcome on the surrogate machine. Too many things can happen at run-time that could cause the keystrokes to produce different results.

7.2.4 Operation Shipping without Involving the File System

Can we use operation shipping without involving the file system? We can imagine that someone may design a meta-application that logs every command a user types, and, without involving the file system, remotely executes the same commands on a surrogate machine. For the following reasons, I believe such a system would not work. First, if the file system had no knowledge that the second execution was a re-execution, it would treat the files produced by the two executions as two distinct copies, and would force the client to fetch the surrogate copy. Second, it might even think that there was an update/update conflict. Finally, it cannot ensure the correctness of the re-execution. Therefore, I believe that the file system plays a key role in useful and correct operation shipping.

Chapter 8

Conclusions

Mobile computing imposes a number of challenges to system designers. Traditional distributed file systems were designed with the assumption that the computers in the systems are connected by pretty good networks, mobile computing breaks this assumption. Therefore, designers must find ways to adapt a distributed file system to the new mobile computing environment.

This dissertation has studied in depth one such adaptation. It has re-thought about the traditional wisdom that files should always be shipped by their contents (value shipping). In response to the challenge of mobile computing, I have proposed an alternative that the file system can ship instead the operations that have been performed on the files (operation shipping). The traditional wisdom makes a lot of sense when the distributed file system works with only strong networks, but it needs a re-visit when the distributed file system has to work with also weak networks.

The concept of operation-based update propagation is conceptually simple. But the real question is whether it can indeed be used in the specific context: distributed file systems. This work has provided an affirmative answer to the question by really designing, implementing, and evaluating such a file system. Along the way, we have experienced a few surprises: that of non-repeating side-effects, that of the complication with cancellation optimization, and that of the design alternatives regarding re-execution styles and propagation granularities for application-aware operation shipping. I believe these are all valuable experiences to the designer community.

A number of research questions were posed in Chapter 1. They have already been answered by various chapters in this dissertation. Here, a quick summary of the answers is given:

1. *How are user operations logged? What kind of user operations can be logged? Who are responsible for logging them?*

   User operations are logged by a co-operation between the file system and a logging entity in the user space. There are two types of user operations that can be logged. The first is an invocation command of a non-interactive application, and the second is an application-specific command of an interactive application. For the first type of user operations, the high-level logging entity is an interactive shell; for the second type of user operations, the high-level logging entity is the concerned application itself.

2. *How are user operations re-executed? Will server scalability be hampered?*

   User operations are re-executed on a surrogate client machine that is strongly connected to the server. The scalability of the server will not be hampered as the re-execution load is delegated to the surrogate client.

3. *How about the correctness of the update propagation? What should the file system do if the re-execution does not re-generate the same file?*

   The correctness is ensured by the four-step mechanism described in Section 3.4. The key steps are that the surrogate uses fingerprints to ensure the re-generated files are the same as the originals, and that if they are not the same, the file system will fallback to use value shipping to propagate the updated files. Further, if the re-generated files are different from the originals in some minor ways, it is possible to use the techniques of forward error correction and temporary-file renaming to fix the minor re-execution discrepancies.

8.1 Contributions

The top-level contribution of this thesis is the substantiation of the claim that is stated in Chapter 1. That is, "*operation-based update propagation is a feasible and beneficial mechanism in a mobile file system.*" The claim is substantiated by the design, implementation, and evaluation of three realistic prototypes. At the next level of detail, the following specific contributions have been made:

- *Surrogate.* The concept of the surrogate, which helps to preserve server scalability, has been proposed. The prototype file system has been designed around the concept. (Section 3.2)

- *Validation and Fallback Mechanism.* A four-step validation mechanism and a fallback mechanism have been designed and implemented in the prototype file system. Through the two mechanisms, the correctness of update propagation can be ensured. (Sections 3.4 and 4.2)

- *Application-transparency for Non-interactive Applications.* Most non-interactive applications have been identified as being transparent to operation shipping; they need not be aware of operation shipping while being involved in it. Experiments have been done to confirm their transparencies. The transparencies imply that operation shipping is backward compatible with these application. (Sections 3.5 and 6.2.2)

- *Design Alternatives for Application-aware Operation Shipping.* The properties of interactive applications have been analyzed. It has been found that they execute in the iterative style (in contrast, non-interactive applications execute in the one-shot style), and that update propagation can be done with two different granularities: propagate-after-exit and early-propagation. Also, it has been found that there are two design alternatives for application-aware operation shipping. In favor of simplicity, the first alternative has been adopted in the design and implementation of the prototype file system; also, for completeness, the second alternative has also been studied extensively. (Sections 5.1, 5.2, and 5.3)

- *Techniques for Fixing Non-repeating Side Effects.* The phenomenon of non-repeating side effects has been identified. It prevents some common applications from being used with operation shipping. Two techniques have thus been proposed to fix these side effects: a novel use of forward-error-correction code for fixing side effects due to time stamps, and the technique of temporary-file renaming for fixing side effects due to temporary files. Both techniques have been implemented and incorporated into the prototype file system. (Section 4.3)

- *Re-enabling the Use of Cancellation Optimization with Operation Shipping.* It has been found that cancellation optimization interferes with operation shipping. The former wants to delete CML (client-modify log) records so as to reclaim storage space and reduce reintegration work load; the latter wants to keep the records for validation. This complication has been addressed and solved by using the concept of ghost CML records; cancellation optimization has thus been re-enabled for use with operation shipping. The solution has been incorporated into the prototype file system. (Section 4.4)

- *Operation Shipping Coda.* The Coda File System has been extended to add the support for operation shipping. This exercise has demonstrated that the idea of operation shipping can indeed be incorporated into an existing file system. (Chapters 4 and 5)

- *Operation-logging Interactive Shell.* The Bourne Again Shell (`bash`) has been extended, as an example interactive shell, to add the capability of operation logging. It can thus work with the extended Coda for application-transparent operation shipping, in which most non-interactive applications can be involved transparently. The extension has demonstrated that the needed modification for operation logging is trivial. (Sections 4.1.3.2)

- *Extended GIMP.* The GNU Image Manipulation Program (the GIMP), as an example interactive application, has been extended to add the capability of operation logging and replaying. After the extension, it can work with the extended Coda to demonstrate the feasibility of application-aware operation shipping. The needed modification has been found to be moderate. (Sections 5.2.2)

- *Qualitative Evaluation.* The feasibility of operation shipping has been demonstrated with experiments. For application-transparent operation shipping, all of the selected non-interactive applications could be used transparently for operation shipping, and three of them have demonstrated that the techniques for fixing non-repeating side effects are effective; for application-aware operation shipping, all except one of GIMP commands can be operation shipped, and the exception can be remedied by minor modification of the GIMP software. (Sections 6.2.2 and 6.3.2)

- *Quantitative Evaluation.* Both application-transparent and application-aware operation shipping have been quantitatively evaluated using controlled experiments. In both cases, the evaluations have demonstrated the following three benefits of using operation shipping: (1) the network traffic for update propagation can be dramatically reduced, (2) the elapsed time for update propagation can be substantially reduced, especially when the network is slow, and (3) the elapsed time for update propagation becomes much less sensitive to the network condition. (Sections 6.2 and 6.3)

8.2 Future Work

This work can be improved or extended in a number of ways. In the following, nine suggestions are made. They are, in fact, suggestions of two different scopes. The first five are relatively minor enhancements or extensions to the current work: *flexible logging policies*, *application profiles*, *dynamic decision*, *more interactive applications as case studies*, and *handling time stamps that change in lengths*. The last four are more major undertakings: *incorporation of other traffic reduction techniques*, *re-execution in the iterative style*, *shared surrogate*, and *downstream operation shipping*.

8.2.1 Flexible Logging Policies

The current logging shell uses the most straightforward policy for operation logging (and hence shipping). That is, all user operations performed with the shell are logged. However, the users may not want all applications be involved in operation shipping. For example, they may know a priori that the replayed operations are not going to be repeating – because, say, the application is randomized in nature. Another example is that they know a priori that the application involved has unwanted side effects – such as sending an email message on every execution.

Currently, the users can work around this simple-minded policy; they can switch to use an ordinary shell when they do not want their operations to be logged. However, this breaks user transparency and is tedious in the long run. Therefore, a minor improvement for the logging shell is to add a policy module to support more flexible logging policies. With the module, the users can selectively enable or disable an application from operation shipping.

8.2.2 Application Profiles

The preceding idea can be further generalized; the shell can allow the users to provide some hints about the properties of the applications via an *application profile*. As we know, applications have different properties that affect whether some steps are indeed needed for operation shipping. For example, LaTeX puts time stamps in their output files, but `sgml2latex` does not; therefore, the former needs error correction code for fixing the re-execution discrepancies, but the latter does not. Another example is that the output files of some applications can be handled more effectively by other traffic-reduction techniques (Section 8.2.6), such as delta shipping (for instance, the applications are randomized in nature, or they are too compute-intensive to be re-executed).

With these hints, the file system can uses more optimized procedures for output files of different applications. Note that, to provide these hints, the users do not need accesses to the source code of the applications, nor do they need to modify the applications, because they can provide these hints from observing the external behaviors of the applications. In other words, the operation shipping mechanism is still transparent and backward compatible to these applications. Also note that the file system should tolerate wrong hints; in the worst case, only the performance of update propagation should suffer, but the correctness should not be compromised.

8.2.3 Dynamic Decision

As explained in Section 4.2, the current version of our prototype attempts operation shipping for a record whenever it is eligible for operation shipping. This is in fact a *static* approach. It assumes that the connectivity between a mobile client and its server is always weak. In real life, however, a mobile client may have occasional strong connectivity. During that time, as explained in the Appendix A, value shipping is more efficient than operation shipping. One possible future work is to incorporate the cost model detailed in Appendix A so that a mobile client *dynamically* decides whether it should use operation shipping or value shipping for a record.

8.2.4 More Interactive Applications as Case Studies

Due to time limitation, only one interactive application – the GIMP – is studied in this work. My experience with the GIMP is that operation logging and replaying facilities are reasonably easy to add, and there is no need to make a lot of changes in the internal logic of the application. It will be very interesting to see if the same experience can be said to other interactive applications.

Furthermore, there are some applications that the general public does not have access to the source code. The two popular office-automation software packages on Linux and Unix – Applixware and StarOffice – are good examples. Is it possible to add operation logging and replaying capabilities even when the source code is not available? If it is possible, then we need not wait for the vendor to recognize the importance of mobile computing in general and operation shipping in particular. Satyanarayanan, Flinn and Walker's recent work on Visual Proxy has demonstrated that they can add capabilities to some applications without having source code [50]. Whether their idea can be used with operation logging and replaying is an interesting question to explore.

8.2.5 Handling Time Stamps That Change Length

In Section 4.3.1.3, the technique of forward error correction is proposed to fix the non-repeating side effect due to time stamps. The technique has solved the problem effectively. However, it is also known to have a limitation. That is, forward error correction cannot correct errors that change the length of the data block.

Some applications write time stamps in a form that involves length changes. We need other techniques to fix side effects of this kind. Some binary-delta algorithms, such as the rsync algorithm, are good candidates for us to consider.

8.2.6 Incorporation of Other Traffic Reduction Techniques

In Section 7.2 the techniques of delta shipping and data compression have been discussed. They both have some disadvantages as well as advantages when compared to operation shipping. Therefore, they can serve as complements for operation shipping. It is possible to build a file system in such a way that all three techniques are incorporated. The file system can choose dynamically the most appropriate techniques. Also, combined with the idea of application profiles (Section 8.2.2), the file system can use the hints provided the users while selecting the right techniques.

8.2.7 Re-execution in the Iterative Style

As discussed in Section 5.1, the prototype in its current state supports only the one-shot re-execution style. In this style, user operations performed to an application are propagated only after the application process has exited. An interesting future work is to explore re-executions in the iterative style, which support the finer-grain early-propagation approach. There are three advantages: (1) the semantics of individual saving commands are honored, and the users does not need to change their working habits; (2) replayed command groups will have a higher chance to be repeating, since the probability that the whole group is repeating increase as the number of commands in the group decreases; and (3) the logging sessions will be of shorter durations, and the chances for breaking conditions to happen are smaller (a breaking condition makes a user operation not eligible for operation shipping). Section 5.3 has an outline of the design.

8.2.8 Shared Surrogate

This work assumes that each weakly-connected client has a dedicated surrogate. A justification for this assumption is given in Section 3.2.3. The most important reason is that many users already own their second personal computers that can be configured as the surrogates. Therefore, the use of dedicated surrogates does not imply extra investments in hardware. However, there are other users who do not own a second personal computers that can be configured as the surrogates. For example, many college students can use their notebooks to access the campus networks in their colleges, but they do not own some machines on the campus networks that can serve as the surrogate machines. For this kind of users, they may need to have some shared surrogate machines. There are at least two interesting research questions. First, is it possible for one machine to instantiate multiple different execution environments? What is the best model to achieve this? Second, how should the shared surrogate handle concurrent requests of re-execution from multiple weakly-connected clients? What is the right level of concurrency control?

8.2.9 Downstream Operation Shipping

This work addresses update propagation in one direction only: from client to server. However, files are also being transferred in the reverse direction – that is, from server to client. Is it possible to use the idea of operation shipping also in this direction? In the following, operation-based update propagation in the former direction is called *upstream operation shipping*, and that in the latter is called *downstream operation shipping*.

Downstream operation shipping may be useful in the following scenario. Suppose a client – let us call it the $Writer$ client – updates a file F, and later another client – let's call it the $Reader$ client – references the file, then $Reader$ needs to fetch the file from the server. The traditional way of fetching the file of course is to transmit the contents (value) of the file from the server to the $Reader$. However, if $Reader$ is weakly connected, the server may find that it is more easy to transfer to $Reader$ an operation log of how F was updated by $Writer$.

In general, upstream operation shipping is important when a weakly-connected client is primarily a producer of data, whereas the downstream operation shipping is important when the client is primarily a consumer of data.

This thesis considers upstream operation shipping first because of the following three reasons. First, upstream operation shipping is the simpler of the two cases. Second, it itself is an important enough problem in real life. Third, many weak links are asymmetric: usually there are higher bandwidth in the downstream direction than in the upstream direction. For example, a 56K modem can transmit data at 56 Kbps downstream but only at 28.8/33.6 Kbps upstream [1]. Another example is the ADSL modem, which can transmit data at 640 Kbps-2 Mbps downstream but only 64 Kbps upstream [9].

Nevertheless, the downstream operation shipping should be an interesting research problem. There are at least the following issues that need to be addressed. First, $Writer$ must prepare an operation log in anticipation of a downstream operation shipping. However, how does $Writer$ know that there will be a need of downstream operation shipping? Should $Writer$ always prepares operation logs for every file that it updates?

Second, similarly, should the server always keeps operation log for every object that it hosts? For how long should these operation logs be kept?

Third, in the case of upstream operation shipping, the client can nominate a surrogate to replay the user operations. However, in the case of downstream operation shipping, the client itself has to replay the user operations. Also, the replaying mechanism is more complicated. This is because the replaying has to be done in the middle of the servicing of an object's file-system read request, which triggers the downstream operation shipping.

Fourth, in the case of upstream operation shipping, the update propagation is done in the temporal order of updates. However, in the case of downstream operation shipping, the references to the files can be in an arbitrary order. When a user operation is to be replayed on $Reader$, some of the read-from files by the user operation may not yet be present on $Reader$, so the server needs to ship also all these read-from files. This has two complications: (1) these read-from files may be so bulky that they are even more difficult to ship than the referenced file F, (2) some of the read-from files may be shippable by operation, and we shall have a complicated situation in which a downstream operation shipping triggers another downstream operation shipping.

8.3 Final Remarks

Should we force the users to adapt to a mobile computing environment, or should we let the file system to adapt to the environment? Without an efficient update propagation scheme, users would have to adapt their behaviors to the unpleasant weak network environment. For example, a user would be forced to bring her mobile computer to her office so that it can reintegrate with the server more effectively with the availability of a strong network. Another user would choose to work on the local file system on his mobile computer; he would manually replay his operations on the DFS when he wants to share his results with others. A third user would be forced to keep her dial-up network connected for hours after she has finished her work, so that her updates can be trickle through the slow network.

The goal of my work is to save mobile users from these user adaptations. The ideal of mobile computing is to let users to carry out their work everywhere they go, without having to worry about the constraints imposed by the environments. With this work, I hope we are one more step closer to this ideal.

Appendix A

Cost Model

In the following, the cost model of value shipping and that of operation shipping will be presented. For each case, there are two different costs involved: network traffic and elapsed time.

Value Shipping

Assuming the overhead is negligible, the network traffic is

$$\text{Traffic}_v = L, \tag{A.1}$$

where L is the total length of the update files.

The elapsed time is

$$T_v = L/B_c, \tag{A.2}$$

where B_c is the bandwidth of the network connecting the client to the server.

Operation Shipping

The network traffic is

$$\text{Traffic}_{op} = L_{op}, \tag{A.3}$$

where L_{op} is the length of the operation log.

The elapsed time is T_{op}. The latter is composed of four components:

1. the time needed to ship the operation log (L_{op}/B_c),
2. the time needed for re-executing the operation (E),
3. the time needed for additional computational overhead (H_{op}) such as computing checksum information and encoding and decoding of forward-error-correction codes, and
4. the time needed to ship the updated files to the servers. There are two cases for this last component. If the re-execution passes the validation (accepted), the updated files will be shipped from the surrogate (the time cost will be L/B_s, where B_s is the bandwidth of the network connecting the surrogate to the server); if the re-execution fails the validation, the updated file will be shipped from the client (the time cost will be L/B_c).

The following equation summarizes the time costs involved:

$$T_{op} = \begin{cases} L_{op}/B_c + E + H_{op} + L/B_s & \text{if accepted} \\ L_{op}/B_c + E + H_{op} + L/B_c & \text{if rejected} \end{cases} \tag{A.4}$$

Comparing the Costs

We can now compare the costs involved with the two types of update propagations. Operation shipping is more favorable than value shipping only in certain conditions.

First, operation shipping saves network traffic if the operation log is more compact than the updated files ($L_{op} < L$).

Second, it speeds up the update propagation ($T_{op} < T_v$) if the following five conditions are true:

1. the re-execution is accepted,
2. the operation log is compact ($L_{op} \ll L$),
3. the re-execution is fast ($E \ll L/B_c$),
4. the time needed for additional computational overheads is small ($H_{op} \ll L/B_c$), and
5. the surrogate has a much better network connectivity than the client ($B_s \gg B_c$).

Bibliography

[1] 56K modem FAQ. Available from http://www.24hoursupport.com/56k.html.

[2] http://www.cse.cuhk.edu.hk/ clement/source_code/.

[3] http://www.gimp.org.

[4] D. Bachmann, P. Honeyman, and L. Huston. The Rx Hex. In *Proceedings of the First IEEE Workshop on Services in Distributed and Networked Environments*, Prague, Czech Republic, Jun 1994.

[5] M. G. Baker, J. H. Hartmann, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurement of a Distributed File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, Oct 1991.

[6] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1), February 1989.

[7] G. Bozman, H. Ghannad, and E. Weinberger. A Trace-Driven Study of CMS File References. *IBM Journal of Research and Development*, 35(5-6), Sep-Nov 1991.

[8] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software–Practice and Experience*, 21(8):757–785, August 1991.

[9] ADSL Forum. Office web site. Available from http://www.adsl.com.

[10] Free Software Foundation. BASH. Available from http://www.gnu.org/software/bash/bash.html.

[11] The Coda Group. Coda File System. Available from http://coda.cs.cmu.edu.

[12] A. Houghton. *The Engineer's Error Coding Handbook*. Chapman & Hall, 1997.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), December 1988.

[14] Airsoft Inc. Powerburst – The First Software Accelerator That More Than Doubles Remote Node Performance. Available from http://www.airsoft.com/comp.html, Cupertino, CA, USA.

[15] A. D. Joseph, J. A. Tauber A. F. deLespinasse, and M. F. Kaashoek D. K. Gifford. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, USA, Dec 1995.

[16] P. Karn. Error Control Coding, a Seminar handout. Available from http://people.qualcomm.com/karn/dsp.html.

[17] M. L. Kazar, B. W. Leverett, O.T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S.-T. Tu, and E. R. Zayas. Decorum File System Architectural Overview. In *Proceedings of the Summer 1990 USENIX Technical Conference*, June 1990.

[18] J. J. Kistler. *Disconnected Operation in a Distributed File System*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1993.

[19] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[20] P. Kumar. *Mitigating the Effects of Optimistic Replication in a Distributed File System*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1994.

[21] P. Kumar and M. Satyanarayanan. Flexible and Safe Resolution of File Conflicts. In *Proceedings of the USENIX Winter 1995 Technical Conference*, New Orleans, LA, USA, Jan 1995.

[22] O. S. Kylander and K. Kylander. *GIMP: The Official Handbook*. The Coriolis Group, Also available from http://manual.gimp.org, 1999.

[23] L Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley Publishing Company, 2nd edition, 1994.

[24] Q. Lu. *Improving Data Consistency for Mobile File Access Using Isolation-Only Transaction*. PhD thesis, Carnegie Mellon University, School of Computer Science, May 1996.

[25] Q. Lu and M. Satyanarayanan. Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions. In *Proceedings of the Fifth IEEE HotOS Topics Workshop*, Orcas Island, WA, USA, May 1995.

[26] J. MacDonald. Versioned File Archiving, Compression, and Distribution. submitted for the Data Compression Conference, an earlier version is available from http://www.XCF.Berkeley.edu/~jmacd/xdelta.html, 1998.

[27] T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart. A Coherent Distributed File Cache with Directory Write-Behind. *ACM Transactions on Computer Systems*, 12(2), May 1994.

[28] H. Mashburn, M. Satyanarayanan, D. Steere, and Y. W. Lee. *RVM: Recoverable Virtual Memory, Release 1.3*. School of Computer Science, Carnegie Mellon University, Available from
http://coda.cs.cmu.edu/doc/html/rvm_manual.html, 1997.

[29] J.C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceeding of the ACM SIGCOMM'97*, 1997.

[30] L. B. Mummert. *Exploiting Weak Connectivity in a Distributed File System*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1996.

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

[32] L. B. Mummert and M. Satyanarayanan. Large Granularity Cache Coherence for Intermittent Connectivity. In *Proceedings of the 1994 Summer USENIX Conference*, Boston, MA, USA, Jun 1994.

[33] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1), Feb 1988.

[34] B. Noble and M. Satyanarayanan. An Empirical Study of a Highly Available File System. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, TN, USA, May 1994.

[35] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Knupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In

Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Orcas Island, Washington, USA, Dec 1985.

[36] K. Patersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[37] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, Jun 1994.

[38] J. Peek, T. O'Reilly, and M. Loukides. *UNIX Power Tools*. O'Reilly & Associates, 1993.

[39] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. A Network Transparent, High Reliability Distributed System. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, Pacific Grove, CA, USA, Dec 1981.

[40] The FreeBSD Documentation Project. CVSup: in FreeBSD Handbook. Available from
http://www.freebsd.org/handbook/cvsup.html.

[41] R. Rivest. The MD5 Message-Digest Algorithm, Internet RFC 1321. Available from http://theory.lcs.mit.edu/˜rivest/publications.html, April 1992.

[42] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *USENIX Summer Conference Proceedings*. USENIX Association, June 1985.

[43] M. Satyanarayanan. A Study of File Sizes and Functional Lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, December 1981.

[44] M. Satyanarayanan. Integrating Security in a Large Distributed System. *ACM Transactions on Computer Systems*, 7(3), Aug 1989.

[45] M. Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *Computer*, 23(5), May 1990.

[46] M. Satyanarayanan. The Influence of Scale on Distributed File System Design. *IEEE Transactions on Software Engineering*, 18(1), Jan 1992.

[47] M. Satyanarayanan, editor. *RPC2 User Guide and Reference Manual*. School of Computer Science, Carnegie Mellon University, Jul 1995.

[48] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Fifteenth ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, USA, May 1996.

[49] M. Satyanarayanan, M. R. Ebling, J. Raiff, and P. J. Braam. *Coda File System User and System Administrators Manual*. School of Computer Science, Carnegie Mellon University, August 1997. version 1.1.

[50] M. Satyanarayanan, J. Flinn, and K. R. Walker. Visual Proxy: Exploiting OS Customizations without Application Source Code. *Operating Systems Review*, 33(3), July 1999.

[51] M. Satyanarayanan, J. Howard, D. Nichols, R. Sidebotham, A. Spector, and M. West. The ITC Distributed File System: Principles and Design. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, Dec 1985.

[52] M. Satyanarayanan, J. J. Kistler, P. Kumar, and H. Mashburn. On the Ubiquity of Logging in Distributed File Systems. In *Third IEEE Workship on Workstation Operation Systems*, Key Biscayne, FL, USA, Apr 1992.

[53] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transaction on Computers*, 39(4), April 1990.

[54] M. Satyanarayanan, J. J. Kistler, L. B. Mummert, M. R. Ebling, P. Kumar, and Q. Lu. Experience with Disconnected Operation in a Mobile Environment. In *Proceedings of the USENIX Symposium on Mobile & Location Independent Computing*, Cambridge, Massachusetts, USA, Aug 1993.

[55] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1), Feb 1994. Corrigendum: May 1994, Vol. 12, No. 2, pp. 165-172.

[56] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.

[57] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. Addison-Wesley, 5th edition, 1998.

[58] Cyclic Software. Concurrent Versions System (CVS). Available from http://www.cyclic.com/.

[59] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, USA, Dec 1995.

[60] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, The Australian National University, Available from http://samba.anu.edu.au/rsync/, June 1996.