# Mitigating the Effects of Optimistic Replication in a Distributed File System

Puneet Kumar

December 1994

CMU-CS-94-215

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

This research was sponsored by the Air Force Materiel Command (AFMC) and the Advanced Research Projects Agency (ARPA) under contract number F19628-93-C-0193. Additional support was provided by the IBM Corporation, Digital Equipment Corporation, Bellcore, and Intel Corporation.

**Keywords:** Application-specific resolution, conflicting updates, Coda, directory resolution, distributed file systems, high availability, mobile computers, one-copy serializability, optimistic replication, transparency.

*In memory of my Father*

*For my Mother*

# Abstract

Optimistic replication strategies can significantly increase availability of data in distributed systems. However such strategies cannot guarantee global consistency in the presence of partitioned updates. The danger of conflicting partitioned updates, combined with the fear that the machinery needed to cope with conflicts might be excessively complex has prevented designers from using optimistic replication in real systems.

This dissertation puts these fears to rest by showing that it is indeed practical and feasible to use optimistic replication in distributed file systems. It describes the design, implementation and evaluation of the mechanisms used to transparently resolve diverging replicas in the Coda file system. Files and directories are resolved using orthogonal mechanisms due to the difference in their structure and semantics. A server-based mechanism that uses operation logging is utilized to resolve directories, while a client-based mechanism that uses application support is utilized to resolve files. When automatic resolution fails a repair tool in conjunction with standard Unix utilities aids the user in merging the diverging replicas. The combination of these mechanisms allows the system to provide high data availability with minimal impact on its usability, scalability, security and performance.

Coda has been in daily use by thirty-five users for more than three years. The system consists of ten servers storing more than four Gigabytes of data and seventy-five clients, consisting of desktop and mobile hosts. Empirical measurements show that the system has maintained usability by automatically resolving partitioned updates on more than 99% of the attempts. Furthermore, the automatic resolution facility has excellent performance, minimal overhead and is rarely noticeable in normal operation. Usage experience with the repair facility has also been positive.

This dissertation makes four significant contributions: a design of simple yet novel automatic resolution techniques; a formalization of the Unix file system model and proof of correctness of the resolution methods; implementation of these methods in a system with a real user community; and measurements, showing the efficacy of the approach.

# Acknowledgments

I came to Carnegie Mellon University seven years ago soon after finishing my undergraduate education at Cornell University. Coming from a school that excels in theoretical work to an environment that takes pride in building real systems was a difficult transition. However having Satya as my mentor, who is one of the most understanding and patient advisors, made this transition easier. It has been a great pleasure to work with him for the past seven years. He provided useful and insightful feedback for all my research ideas. and always made the time to meet me in spite of his hectic schedule. He gave me the confidence to solve difficult problems and taught me the importance of analyzing and validating my research ideas through experimentation. Above all, Satya has been a good friend. I will cherish this friendship for many years to come. Thanks for everything Satya.

I would like to thank other members of my thesis committee, Alfred Spector, Brian Bershad and Maurice Herlihy, for providing feedback on the dissertation. Their feedback has helped in improving the quality of this document. I would like to specially thank Alfred who co-advised me with Satya during my first few years at CMU and gave me useful feedback on early designs of my thesis work.

This thesis wouldn't have been possible without the support of members of the Coda project. I would like to thank past and current members of the project including Maria Ebling, Jay Kistler, Qi Lu, Hank Mashburn, Lily Mummert, Brian Noble, Morgan Price, Joshua Raiff and David Steere, for their help. They are a smart bunch and I am honored to have had the opportunity to work with them. I wish them all luck in their future endeavors. I would like to specially thank two Coda members, Jay Kistler and David Steere. Jay helped me design the resolution architecture and provided useful tips to implement the design. Moreover, he has been a great friend and a constant source of encouragement over the past seven years. David helped me an uncountable number of times in tracking down serious server bugs. He never failed to put a smile on my face even when I was completely stressed out writing the thesis. Thanks for your friendship, Jay and David!

I would like to thank other members of the SCS community who used the Coda system. I appreciate their patience in using an experimental system and thank them for providing me with useful feedback on the pros and cons of the system.

Over the past seven years, I have probably spent more time in my office than any other place at CMU. I would like to take this opportunity to thank my office-mates, both past and present, for providing me with a friendly and enjoyable work environment. My officemates included Alan Christiansen, Andrew Hastings, Alicia Pérez, Henry Rowley, Manuela Veloso and Mei Wang. *Nous nous sommes bien amusés de pratiquer notre Français ensemble*.

I would like to specially thank Manuela for providing me the encouragement needed to finish the dissertation. She helped me on innumerable occasions in dealing with the idiosyncrasies of $\LaTeX$. Moreover, she has been a great companion and a constant source of good advice outside the work environment. Thanks a lot, Manuela.

Some people may find this hard to believe, but it is possible to have the same house mate for six years. I would like to thank my house mate Anurag Acharya, for his companionship and support. Even though we had our differences in the beginning, I always liked his affectionate and humble personality. I would also like to thank Aarti Gupta who also shared our house for three years. I know I will miss them when I leave Pittsburgh.

I have been lucky in enjoying the friendship of a huge group of people during my stay in Pittsburgh. I would like to thank the Brew Crew, especially Gary, Wayne, Jonathan and Sue for providing me with an evening off on Thursdays. I would like to thank my Indian friends especially Rukmini and Manu, Rajeev and Barathi, Anurag Gupta, Ashish and Puja, Satish and Priya, who provided me with the encouragement and support needed to finish the dissertation. I would also like to thank other friends at CMU including Jürgen and Rebecca, Jody and Sam, Chris and Hannah, Deborah and Raj, José, Greg and Tanya, Matt, Scott, Francesmary and Jeff, Juan, João and Lucco. The loss of memory that accompanies the post dissertation blues may have resulted in the omission of names that deserve to be mentioned in this list. To those of you I forgot to mention I apologize and thank you for your support.

Even though I was far away from my parents in India, I was lucky to enjoy the love and affection of a special couple, Phyllis and John Zekauskas, a.k.a. Mrs. Mom and Mr. Dad to some of you. I got to know them through Peter, my best friend and room-mate at Cornell University. I would like to take this opportunity to thank the entire Zekauskas family for allowing me to become a part of their family and to experience first-hand some of the wonderful American family traditions and values.

I would like to warmly thank my sisters, Sangeet and Guneet, and their families, for their support and patience over the past few years. Even though they were going through school at the same time themselves, they never failed to cheer me up whenever I was completely disgusted with school and required that badly needed pep talk.

I met my fiancee, Shuchi, only seven months ago. Even though this was during the period in which I was writing the dissertation, taking that extra time to spend a few more hours or days with her were worthwhile. She has been extremely patient, understanding and comforting and I thank her for all her help.

My father instilled in me the desire to do a PhD. If it weren't for him, I would probably never have thought of coming to graduate school. Unfortunately, he is unable to witness this momentous occasion as he passed away during my second year at CMU. Even though my mother has been alone in India, she has been very supportive over the past six years. Her sacrifices and her boundless love, patience and encouragement have provided me with the strength needed to complete this arduous task of finishing the PhD. I am indebted to both of them forever.

*Puneet Kumar*
*December, 1994*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The importance of optimistic replication as a technique for providing high availability in distributed systems has been known for over two decades [47, 20]. But the use of this technique in actual systems has been minimal. One reason for this has been the fear that conflicting partitioned updates, an inevitable consequence of optimistic replication, might hurt usability unacceptably. A second reason has been concern that the machinery needed to cope with conflicts might be excessively complex and unwieldy. This dissertation addresses these concerns. It shows how one can build an optimistically replicated Unix file system that preserves usability by automating the resolution of partitioned updates and maintains practicality by using simple yet novel resolution techniques.

The rest of this chapter discusses optimistic replication and its related problems. It begins with a discussion on distributed file systems (DFSs) and the use of replication for improving data availability. It then discusses the shortcomings of optimistic replication and proposes a solution for alleviating these problems. The chapter concludes with the thesis statement and a road map for the rest of the dissertation.

## 1.1 Distributed File Systems

DFSs such as AFS [48, 22], NFS [44], Netware and LanManager have become an integral part of organizations with a large number of personal computers. The success of DFSs in such environments can be attributed to three factors.

First, they *simplify the administration* of the large body of machines. Users work at their personal machines or *clients* and concern themselves only with tasks relevant to their work. Administrative tasks like backup and software maintenance are performed at the *servers* by a small group of trained personnel.

Second, they simplify *file sharing* within a user community. File sharing occurs in two forms: inter-user and intra-user. Inter-user sharing arises when one user accesses another user's files; intra-user sharing arises when a user accesses his files from multiple clients. The former simplifies the management of collaborative work and the latter increases a user's mobility within his organization.

Third, DFSs maintain *transparency* by hiding the distributed nature of the system. Applications that are written for a local file system can be used on a DFS without any modifications. Users can also use the DFS just like they would use a local file system.

To provide these three features, most modern DFS designs use a client-server model, provide location transparency and export a programming interface that is the same as or very similar to that of a local file system. In addition, they use client-caching to improve performance. The appropriateness of these design choices is confirmed by their widespread use in most modern DFSs. Two surveys of modern distributed file systems can be found in [29, 46].

A fundamental problem with DFSs is *data availability*. Since files are stored and used on separate machines, a server crash or a network failure that partitions a client from a server can prevent a user from accessing his files. Such situations are very frustrating to a user because they impede computation even though client resources are still available. This problem of data availability in DFSs will increase over time for two reasons:

- The frequency of network failures will increase. As DFSs become more popular and increase in scale, they will cover a larger geographical area, encompass multiple administrative boundaries and consist of multiple sub-networks connected via bridges and routers. As networks get larger, their administration and maintenance becomes more difficult. Furthermore there is an increased probability of congestion or unavailability of a sub-network at any given time.

- The nature of mobile clients will increase the number of occasions on which servers are inaccessible. Wireless technologies such as packet-radio and infrared suffer from inherent limitations like short range and line of sight. Due to these limitations, the network connections between servers and mobile clients will exhibit frequent partitions.

## 1.2 Replication

Replication is the key to providing higher data availability in DFSs. The basic idea is that by replicating a file, its availability can be maintained even if some of the replicas are inaccessible. Two forms of replication are possible. In one form, called *server replication*, a file is replicated across multiple servers. In another form, called *disconnected operation* [25], a copy of the file cached at the client serves as a temporary replica. In either case, availability of a file can be maintained only if at least one of its replicas is accessible.

Replication strategies that have been proposed in the literature can be classified into two broad categories: *pessimistic* and *optimistic*. These strategies differ in the assumptions they make about the environment and the techniques they use to update and maintain consistency between replicas.

Pessimistic replication strategies assume write-sharing is common and provide strict consistency amongst all replicas of a file. To avoid conflicting updates to an object's replicas, they either allow only reads during a failure or allow both reads and writes in only one partition group. Examples of pessimistic strategies are primary site [2, 37], voting [16, 52], quorum consensus [21] and tokens [32]. Pessimistic replication strategies can severely limit the availability of data. For example, the token passing scheme allows an object to be accessed only if a token is presented. Availability is limited if the token is lost due to failures. In the voting scheme, objects can be accessed only in the partition group that obtains a majority vote. In some situations, it is possible that no partition group can obtain a majority vote.

Optimistic replication strategies, unlike pessimistic strategies, allow reads and writes to proceed even during a network partition. They are optimistic in the sense that they assume users will not update the same object in separate partitions. They allow an object to be be read and modified as long as one of its replicas is accessible. As a result optimistic replication strategies provide significantly higher availability than pessimistic strategies.

## 1.3 Shortcomings of Optimistic Replication

Unfortunately optimistic replication suffers from a serious problem – an object's replicas might be updated concurrently in two or more partitions. As a result, such strategies cannot guarantee data consistency across partitions. To cope with this shortcoming, some mechanism is needed to detect and *resolve* the diverging replicas once the partition groups reconnect.

Is resolution feasible in practice? How complex is it to implement the resolution mechanism? What is its impact on a system in terms of its scalability and performance? When automated resolution is not possible, how hard is it for users to perform the resolution manually? Can the system offer any help in such situations? Concerns such as these have held designers back from using optimistic replication strategies in real systems.

## 1.4 The Thesis

The goal of this thesis is to alleviate the potential shortcomings of optimistic replication in Unix systems thereby paving the way for its practical use in real systems. Intuitively, the thesis appears viable for the following reasons:

- A DFS being used in a research and development environment for tasks such as electronic-mail, bulletin boards, document preparation and program development exhibits little write-sharing. Further, the length of a typical partition is much shorter than the average time interval between two write-sharing events. Thus, write-sharing during a partition is rare.
- Even if write-sharing occurs during a partition, the system can often resolve the diverging replicas automatically and transparently when the partitions reconnect. There are two reasons that make this possible.
    1. The semantics of directories (which are more susceptible to write-sharing than files) are well understood by the system.
    2. Although the knowledge needed to resolve a file is not known to the system, it is known to the application that owns the file. Thus the system can use the application's assistance for resolution.

These observations lead to my thesis statement, whose validation is the subject of this dissertation:

*Optimistic replication can be used effectively to improve availability in a distributed file system by reducing the frequency and difficulty of manual resolution. The mechanisms for automated resolution can be designed to minimally impact scalability, security and performance.*

## 1.5 Thesis Validation

This thesis has been validated by designing and implementing techniques for automatic and manual resolution in a DFS named Coda [49]. Coda, a descendant of AFS, uses an optimistic replication scheme to increase availability. The system uses separate mechanisms for automated resolution of directories and files and provides separate tools for manual resolution of each. Thirty-five users have been using Coda on a daily basis for over 3 years. Empirical measurements as well as controlled experiments confirm the safety and efficacy of this approach.

## 1.6 Road-map for the Document

The rest of this document consists of eight chapters. Chapter 2 describes the architecture of the Coda file system and gives a high level view of the mechanisms it uses for resolution. Chapter 3 describes a formal model that precisely defines the criterion for deciding the resolvability of a set of partitioned updates. This model is also used to reason about the correctness of the resolution mechanisms described in the chapters that follow.

Chapters 4 and 5 describe the design and implementation of the mechanisms for automating resolution of directories and files respectively. These chapters include a discussion of the design considerations, details of the resolution architecture and a description of the algorithms.

Chapter 6 describes the design and implementation of Coda's repair facility that is used to manually resolve files and directories.

Chapter 7 evaluates the resolution mechanisms in Coda. It presents empirical and quantitative results. Empirical results are based on actual use of the system over the past year and the quantitative results are based on controlled experimentation using synthetic benchmarks. This chapter also presents some results based on simulations driven by file system traces from our environment.

This document concludes with a discussion of related work in Chapter 8, and a description of the contributions of this dissertation in Chapter 9. The latter chapter also includes a discussion of future work and a summary of key results.

# Chapter 2

# Overview of Resolution in Coda

This chapter gives an overview of the central problem addressed in the thesis and describes its solution at a high level. To set the context of this thesis, the first section describes the Coda file system. It outlines the architecture of the system and emphasizes those considerations that were pivotal in its design. The second section focuses on high availability, a major goal of the system. It discusses the use of optimistic replication and the problems that arise with it in practice. The solutions to these problems are the main thrust of this thesis. Sections 2.3, 2.4 and 2.5 give an overview of these solutions; details of their design and implementation are provided in later chapters.

## 2.1 The Coda File System

The Coda file system is a descendant of the Andrew File System (AFS). AFS was implemented at Carnegie-Mellon University over a period of more than seven years resulting in three different versions: AFS-1, AFS-2 and AFS-3. Coda was derived from AFS-2 and retains all its goals. Specifically, it preserves the *scalability*, *performance*, *security* and *ease of administration* of AFS-2. In addition, it strives to provide *high availability* of data. In the following sections we describe how the system achieves each one of these goals.

### 2.1.1 Scalability

Coda wanted to preserve AFS' ability to support thousands of users. Therefore scalability of the design is an important factor influencing the Coda architecture. Coda divides the system structurally into two distinct groups, *servers* and *clients*. The ratio of the number of servers to clients ranges between 1 : 20 and 1 : 100.

The group of servers, collectively called *Vice*, consists of a small set of dedicated Unix machines. From the viewpoint of security, only the integrity of the servers is important for the correct functioning of the system. As summarized by Satyanarayanan [45], this approach "[decomposes] the system into a small nucleus that changes relatively slowly, and a much larger and less static periphery. From the perspective of security and operability, the scale of the system appears to be that of the nucleus." Physical separation of clients and servers is essential for the system to scale beyond a moderate size.

### 2.1.2 Performance

To improve performance, Coda uses caching of data on clients. Caching reduces the latency of operations for objects found in the cache. It also improves the scalability of the system by offloading some of the work from servers thus allowing more clients to be serviced.

The client cache is managed by a process named *Venus*. Venus transparently intercepts file system requests on Vice objects and services them using information in its cache. Whenever necessary it communicates with the servers. The cached data is stored on disk allowing larger and more effective caches and also shortening the cache warm up time after reboots. Venus caches both files and directories in their entirety. The caching of directories enables Venus to resolve pathnames locally.

File system updates are written through the cache to the servers by Venus. However, file writes are propagated to the server only when the file is closed. This allows Venus to take advantage of faster bulk-transfer protocols and to provide cleaner guarantees about the coherence of its cache. Coherence of cached objects is enforced using a *callback* mechanism. A callback for an object is simply a promise from the server to the client that the latter will be informed if that object is changed. Clients check the validity of an object whenever it is opened by ensuring a callback promise exists for that object. At the servers, a table maps an object to a list of clients that have it cached. When an object is updated, its entry in the table is deleted and all clients that have it cached are informed that their copies are invalid.

### 2.1.3 Ease of Administration

The use of central servers simplifies administration of the system tremendously. The small number of servers can be managed by a relatively small administrative body while the much larger number of clients can be administered individually by their owners.

To further simplify administration, Coda uses a data-structuring mechanism called a *volume*. A volume is a collection of files and directories forming a sub-tree in the Vice name hierarchy. Typically, a volume may contain files and directories belonging to a single user or a project group. All data in a volume resides at a single server and backups are performed by cloning entire volumes. Disk quotas are also enforced at the level of volumes. File system operations never span multiple volumes.

Each volume has a unique 32-bit identifier called the *volume-id* (VID) and a unique name. Either of these can be used to locate the volume the first time it is accessed. Volumes are conceptually similar to mountable Unix file systems. The Vice name space is formed by gluing volumes together by their *root* at *mount points*. Each mount point is a special symbolic link whose contents are the name of the target volume. The process of locating the target volume when crossing over volume boundaries is performed transparently by Venus. This mechanism is pivotal in providing name and location transparency in Coda.

### 2.1.4 Security

At large scale, security becomes a major concern due to the distributed control of clients and relative anonymity of users. Users can no longer depend on the goodwill of their colleagues for protection of their data. Coda addresses these security issues in three ways. First, servers are physically protected and run only trusted software. Second, all client-server communication is done on secure authenticated connections using a variant of the Needham and Schroeder private key protocol [34]. In principle, each connection could be fully encrypted although the system doesn't do this in actual use. Third, protection on individual objects is specified using directory access-control lists containing names of individual users or groups of users. A breach in security at a client gives the intruder rights only to that portion of the name-space to which the user at that client has access. Damage cannot spread to arbitrary regions of the name-space.

### 2.1.5 High Availability

High availability of data is important for clients to provide uninterrupted service to users. The key to providing high availability of data in the face of network failures or remote server crashes is *data replication*. Having multiple copies of data increases the likelihood of at least one replica being available at any given time.

Coda uses two complementary replication mechanisms - *server replication* and *disconnected operation*. The former mechanism replicates data at multiple servers and makes data available to clients even if a subset of the servers are inaccessible. If all servers become inaccessible, the latter mechanism allows a client to service file system requests using data cached on its local disk.

Both replication mechanisms have their disadvantages and neither is adequate for all failure conditions. Maintaining replicas at multiple servers increases the work load and has space overheads. Moreover, server replication can guarantee availability of an object only if at least one of the servers holding that object is accessible to the client. It does not protect against network failures that might isolate the client. Disconnected operation addresses these issues but suffers from other problems. Since the cache size at a client will always be much smaller than the storage capacity of servers, the amount of data accessible to a disconnected client is limited. Limited space becomes a bigger problem when the working set of the user grows beyond the client's cache size. Further, sharing information between disconnected clients is impossible. Finally, backups and media reliability become a concern in small portable clients.

The dichotomy in properties of servers and clients brings out the distinction Coda makes between *first-class* replicas on servers and *second-class* replicas on clients. First-class replicas are permanent, complete, secure and visible to many clients. Second-class replicas on the other hand are weaker in all these respects. Therefore, the strategy used in Coda is to rely on first-class replication as far as possible and to use second-class replication only as a last resort. The two replication strategies are seamlessly integrated by switching between them automatically when the client-server connectivity changes.

Although both mechanisms have the same goal, to provide higher availability, they use very different methodologies. Disconnected operation by itself is a complex facility and has a lot of interesting issues specific to it. These were discussed in great detail in an earlier dissertation [25]. The rest of this thesis discusses server replication in isolation from disconnected operation. Other Coda papers discuss the relationship of disconnected operation to server replication [47, 49].

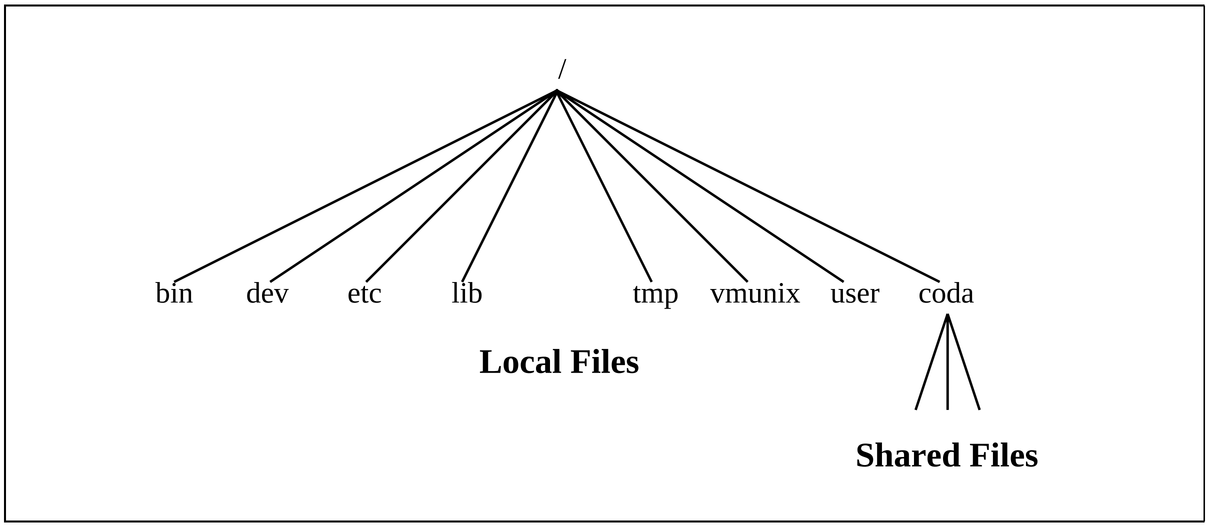
## 2.2 Server Replication

This section provides an overview of server replication in Coda. It describes the granularity of replication, the replica control algorithm and how it realizes high availability in the presence of failures.

### 2.2.1 Replication Granularity

The granularity of replication in Coda is a volume. The set of replication sites for a volume is called its *volume storage group* (VSG). The subset of a VSG that is currently accessible at a client is its *accessible volume storage group* (AVSG).

Volume level replication strikes the right balance between flexibility and easy management of replication information. It allows different regions of the name-space to have different degrees of replication depending on the user's availability requirements. Replicating at a higher granularity, for example, replicating entire servers, wouldn't provide this flexibility but would simplify management of replicas. Replicating at a lower level, for example, at the level of files and directories, would provide the flexibility but suffers from two disadvantages. First, the information the system must maintain for replica location and connectivity state would be harder to manage because of its much bigger size. The number of files and directories is several orders of magnitude larger than the number of volumes in a system. Second, the update protocols would be much more complex. Guaranteeing atomicity of operations like `rename`, that mutate multiple objects, would either require distributed commit protocols because the objects involved could be replicated at different sites, or complicate crash recovery. However, if the objects involved in a mutation are all at one site, then the commit protocol need not be distributed. This requirement is provided by storing all objects within one volume replica at one server and preventing all file system operations from spanning a volume boundary.



The subtree under the directory labeled "coda" is identical at all workstations. The other files and directories are local to each workstation.

Figure 2.1: File System View at a Coda Client Workstation

### 2.2.2 Replica Control

Replica control refers to the mapping of read or write requests on a logical object to its physical copies. The logical view of the file system seen by a user on a Coda workstation is depicted in Figure 2.1. Each workstation has its local, non-shared area of the name space that contains temporary files, startup binaries and other files necessary for the running of the system. The shared name-space contains all Vice objects and Venus ensures each object has the same name at all clients. Users utilize this name to make file system requests on the Vice object.

Replication is made transparent to the user by presenting a *one-copy* view of the system. When a user makes a request for an object, Venus locates its replicas dynamically, services read requests using the latest copy of its data and propagates write requests transparently to all its replicas. This section describes how Venus performs these tasks in the absence of failures.

**Locating replicas** To locate an object's replicas, Venus uses its unique identifier called a *fid* (for *f*ile *id*entifier). The fid of an object contains the identity of the *replicated volume* (RVID) it is contained in. A replicated volume is logically one volume that has several physical replicas each with its own VID. The VIDs of the individual replicas are obtained by querying the *Volume Replication Database* (VRDB). The location of each physical replica is then found by querying the *Volume Location Database* (VLDB) using the VID. Both the VRDB and VLDB are stored at all servers. To reduce the frequency of communication with the server, Venus caches the volume replication and location information returned from the VRDB and VLDB respectively.

**Servicing reads and propagating writes** Once the replicas of an object have been located, fetching data from or propagating data to them is also the responsibility of Venus. It uses a read-one-write-all replicas policy. To minimize the latency for a user's write request, it propagates the update to all members of the VSG in parallel. As a side-effect of the update it also changes the version stamp associated with each replica. The version stamp is guaranteed to be unique for each update. Thus, when two replicas have the same version stamp, their contents are equal. To service a read request for an object, Venus first compares the version stamps of the object's replicas to ensure they are identical. It then fetches the object from only one VSG member to optimize the use of limited bandwidth. Note that Venus fetches an object from the servers only if it does not already have a valid copy of that object in its cache.

### 2.2.3 Coping with Failures

Until now the discussion has focussed on server replication in the absence of failures and how it is made transparent to the user for usability reasons. The system has the look and feel of a regular one-copy Unix file system (UFS) that is shared and identical at all clients. When failures do occur and some servers become incommunicable, maintaining a high level of availability requires modifications to the replica control policy. This is discussed in section 2.2.3.2 below. But first, section 2.2.3.1 describes the kinds of failures the system is ready to cope with.

#### 2.2.3.1 Kinds of Failures

Coda provides continued access to data in the face of two kinds of failures: network partitions and server crashes. Network partitions are a consequence of failures in the networking hardware or software, causing clients to lose communication with one or more servers. Server crashes may be caused due to software bugs or hardware faults. Coda doesn't protect against hardware failures like media crashes. However, disk mirroring could be used to improve reliability.

Network partitions are a more serious failure than server crashes because they can divide the group of servers into two or more disjoint sets such that updates in one set are not visible in the other set. Unrestricted access to partitioned replicas can present stale data to the user, leading to inconsistencies and incorrect computations. When a server crash has occurred, its data is not available to any client and thus no inconsistencies can arise. In other words, network partitions force a tradeoff between availability and consistency.

#### 2.2.3.2 Failures and Optimistic Replica Control

Section 2.2.2 described the replica control strategy used by Venus to make replication transparent to the user in the absence of failures. In the presence of failures, Venus can maintain this transparency by using only the accessible replicas to service user requests. A failure is detected when a request for an operation at a server times out. Once a failure is detected, Venus drops all connections to the inaccessible server, pretends that the VSG of all volumes having a replica at that server has temporarily shrunk, and then continues servicing the user request. The modified replica control strategy it uses is called the read-one-write-all-*accessible* replicas. For read requests, equality is checked only amongst the members of the AVSG. For write requests, updates are propagated only to those same members of the AVSG. To the user, there is no visible change in operation except perhaps an initial pause in the handling of the request that detected the failure due to a timeout. The file system still appears to be like any one-copy UFS except that many failures are masked.

The read-one-write-all-accessible replicas policy provides the highest level of availability because it allows unrestricted access to replicas in all partitions. This level of availability can be maintained only if the AVSG is non-empty. If the AVSG does become empty, Venus enters the disconnected mode of operation and continues servicing user requests using its cached data.

The replica control policy is optimistic in the sense that it assumes operations in different partitions will not update the same object. Of course, this assumption may be wrong and the object may end up with diverging replicas due to concurrent updates in multiple partitions. In practice this is a rare occurrence since write-sharing is not common in the Unix environment. However, when it does occur it leads to the task of making replicas converge to a single value. The next few sections will discuss how this task is handled in Coda.

## 2.3 Coping with Problems due to Optimism

The process of converging diverging replicas to the same value is called *resolution*. To reduce the burden of resolving an object's replicas manually, Coda invokes resolution *transparently* when needed and *automates* it when possible. With these two mechanisms, the failure that led to the divergence can be masked from the user. However, it is possible that resolution cannot be automated under certain circumstances. For example, the partitioned updates may not be merge-able because they violate some semantic invariant when merged even though they preserve this invariant independently.

Invoking resolution transparently requires mechanisms to

- detect the end of a failure, and
- detect objects with diverging replicas

The following two sub-sections discuss each of the transparency mechanisms. Then section 2.3.3 describes how the transparency mechanisms fit together to make resolution transparent to the user. The mechanisms used to automate resolution for the two kinds of file system objects, directories and files, are discussed in section 2.4.

### 2.3.1 Detecting the End of a Failure

The end of a failure is usually marked by the recovery of a server from a crash or the healing of a network partition. The former event can be announced by the recovered server but detecting the latter requires frequent polling by the servers and/or clients. For scalability reasons Coda clients, and not servers, are made responsible for this polling. This offloads some of the work and state maintenance from the servers and also prevents recovery storms at the server if it were to broadcast its restart.

For every replicated volume Venus maintains its state of connectivity with the VSG by probing the servers periodically. In our environment, the probe interval is set to ten minutes. The system also provides a user level command at clients to force a probe at any arbitrary time. To accommodate an ever growing number of servers in the environment, Venus uses two mechanisms to minimize space and time resources for this task. First, it probes only those servers from where it has some objects cached. Second, it probes all of these servers in parallel. Even if the system grows very large, the number of objects cached, and hence the state information for the servers, is limited by the size of the local disk and the user's working set. Probing in parallel limits the time needed to check communicability with all servers to a single timeout interval.

### 2.3.2 Detecting Divergence

Recall from section 2.2.2 that each update has a unique version stamp associated with it. Therefore, partitioned replicas of an object that have been updated during a failure will have different version stamps. Venus compares these version stamps whenever it services a read request for an object missing in its cache. To force this comparison even if an object is in the cache, Venus does the following. As soon as a server becomes communicable after a failure, it invalidates its cached copy of all objects serviced by that server. Then, the next time each one of these objects is accessed, Venus will compare the version stamps of the replicas to re-validate its cached copy. If the version stamps are identical, it deduces the object wasn't modified during the failure and it can simply re-mark its cached copy as valid. However, if the version stamps do not match a divergence is declared.

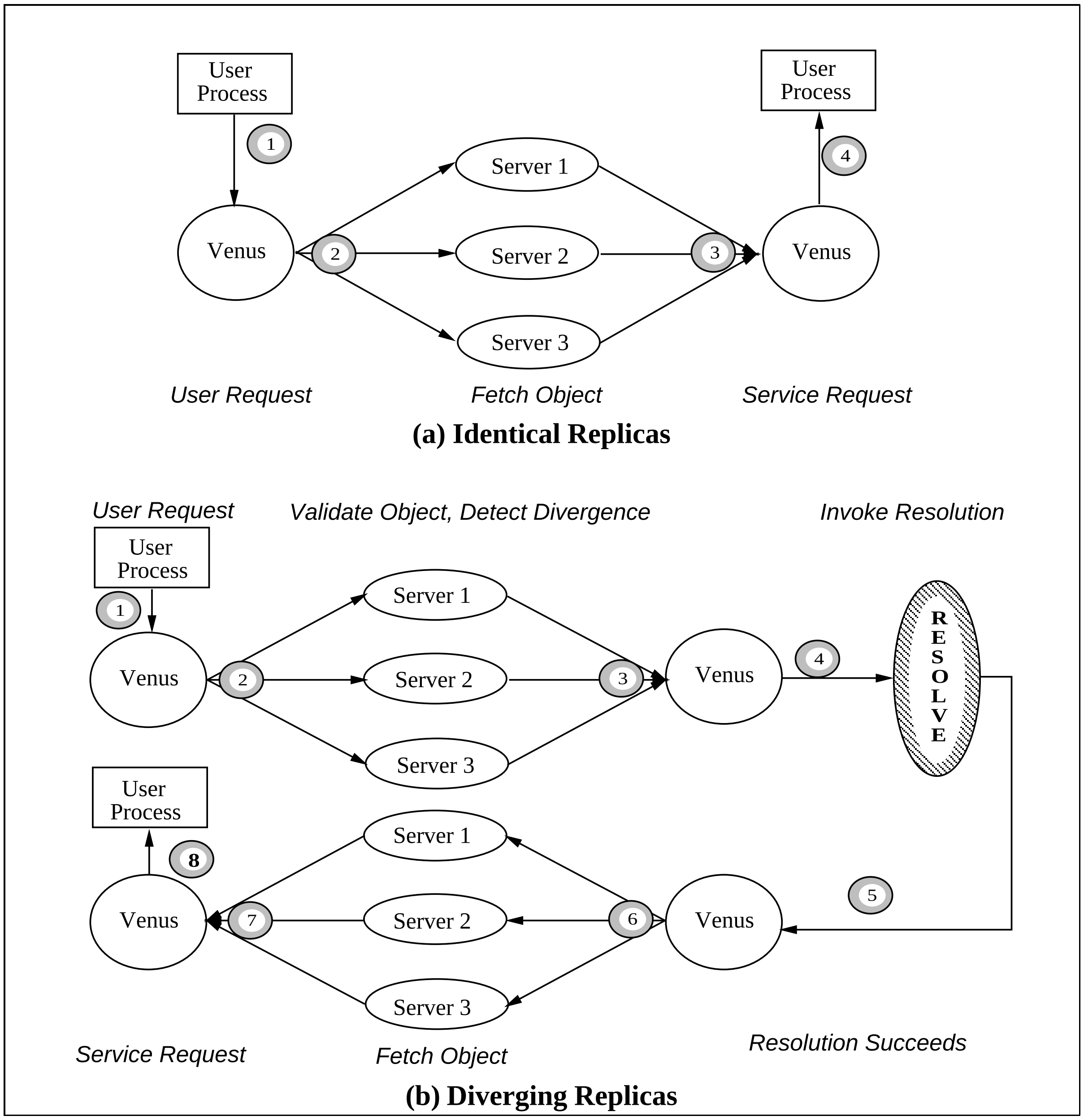
This scheme pushes the work for detecting divergence to the client and allows it to be performed *lazily*, i.e. only for those objects being accessed by the user, not for all objects with diverging replicas. It relieves the servers of the book-keeping and overheads of distributed protocols that might otherwise be necessary for efficiently computing the set of objects with diverging replicas. Moving the book-keeping to the client improves the scalability of the system significantly. The laziness of the scheme reduces the peak demand on the system soon after a recovery from a failure. An *aggressive* approach, in contrast, would try to find all objects with diverging replicas as soon as a server recovers or partitions reconnect. A problem with the lazy scheme is it allows stale replicas to persist and if failures were to re-occur, this data may be used in other updates leading to an increased likelihood of conflicts.

It is possible to have a hybrid scheme in which a client is forced to fetch all the objects serviced by the newly recovered server. If the fetches are separated by long enough gaps, then the demand on the servers wouldn't be bursty and it would also prevent an object with diverging replicas from not being detected. The current implementation could be extended easily to use this scheme. However our usage experience with the system does not indicate the need for such a scheme.

### 2.3.3 Transparency of Resolution: View to the User

When Venus detects diverging replicas for an object while servicing a user request, it suspends the request and transparently invokes resolution on the object. Once resolution completes successfully, Venus can continue to service the user request after fetching the latest merged version of the object. In this case, resolution is completely transparent to the user. The only noticeable effect is perhaps a slight delay in the servicing of the file system request. Figure 2.2 shows the message exchange, between Venus and the servers, that makes resolution transparent to the user.

If resolution is unable to complete its task successfully, the object is marked with a special *inconsistency* flag. Venus will not service requests on this object until it is manually repaired. This prevents any further damage from being propagated due to the unresolvable update. The system provides a special *repair-tool* which can be utilized to examine the object's replicas and



The labelled arrows show the order of messages exchanged by a client and the servers in order to service a user's request. The object is assumed to be replicated at three servers. Figure (a) shows the message exchange when no failures have occurred, all replicas are identical and the object is not in the client's cache. Figure (b) shows the message exchange when resolution is needed. The latter case has more message exchanges and thus also a higher latency to service the user's request.

Figure 2.2: Message Exchange to Service a User Request

merge them manually. Manual repair exposes the failure to the user and therefore the system tries to minimize the frequency of this event.

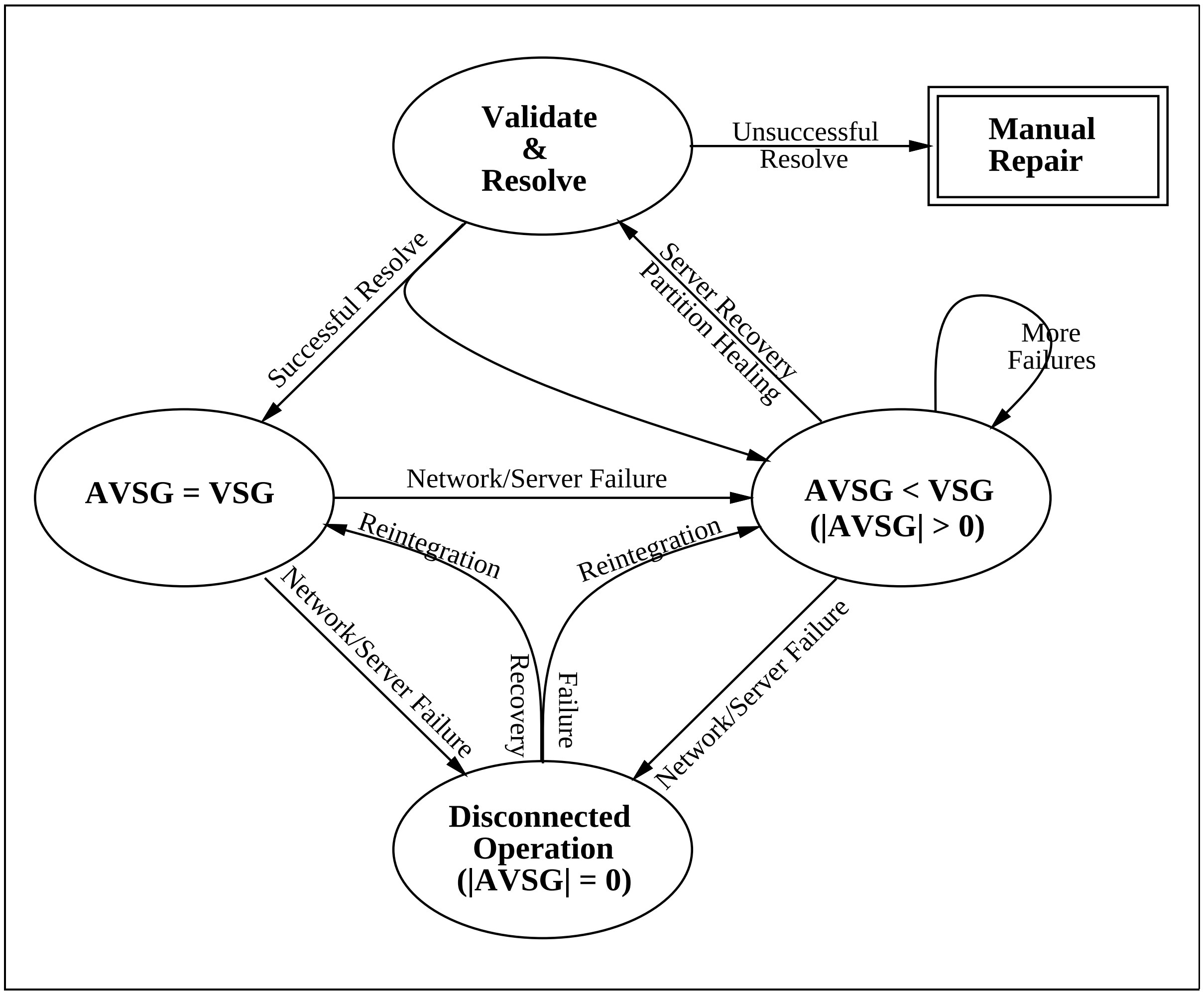


Figure 2.3: Coda Object States and Transitions

Figure 2.3 summarizes the possible states of an object as viewed from a Coda client and the events that trigger the transitions between them. The “validate and resolve” state is only a transitory one which the object passes through whenever communication is re-established with a server. All states except the one labelled “manual repair” are transparent to the user . The object could be in any one of the other four states and seamlessly transition between them without any visible effect to the user. However, an object enters the manual-repair state when it has diverging replicas and the system is unable to automatically resolve the replicas. The object can leave this state only after manual intervention.

## 2.4 Automating Resolution: The Resolution Architecture

The previous section described the framework for invoking resolution transparently. This section gives an overview of the methodology for *automating* resolution. The process of resolving an object consists of several subtasks:

1. **Deducing** the sets of partitioned updates.
2. **Communicating** each set of partitioned updates to all replicas.
3. **Checking** for unresolvable partitioned updates that do not satisfy certain conditions or violate some system invariant.
4. **Performing** the partitioned updates in a fault-tolerant manner.
5. **Marking** the replicas to
   - allow normal file system requests to be serviced once again,
   - or, if unresolvable updates were made, confine them and prepare the object for manual repair.

Coda uses completely different strategies to perform these tasks for the two kinds of objects, files and directories. This section first motivates the need for separation of strategies and then describes the two strategies.

### 2.4.1 Separation in Methodology: Directories vs. Files

The main reason for having different resolution methods for directories and files is the difference in their *structure* and *update methods*. Directories have a well defined structure known to the system. A directory consists of entries that bind unique names to files or other directories. Entries are added or removed from a directory using a well defined interface consisting of a few, fixed set of operations. On the other hand, a file is an untyped byte stream whose semantics are not known to the system. The only file update visible to the system is a rewrite of the entire contents of the file. Since any useful resolution policy requires understanding the semantics of partitioned updates, the system cannot automate the resolution of files. Moreover, since each file's data has unique semantics, a resolution policy must be specific to the application that interprets the file. Therefore, Coda's resolution strategy consists of an automated system-wide policy for directories and a mechanism for transparently invoking application-specific resolvers that implement the application-specific policy.

Files and directories impose different requirements on their respective resolution strategies along two dimensions, *efficiency* of resource usage and *smartness* i.e. effectiveness in recognizing the set of partitioned updates as resolvable or unresolvable. Directory resolution needs to be better than file resolution along both of these dimensions for two reasons. First, since write-sharing is more common for directories than for files, for example, project members share a project directory, the likelihood of a directory needing resolution is significantly higher. Second, since directories are navigational objects, their unavailability could make large portions of the file system hierarchy rooted under that directory unavailable. Therefore, the directory resolution algorithm must be careful not to wrongly classify a set of resolvable updates as unresolvable. Such misclassification for a file affects only the availability of its data.

The argument for separating the resolution strategy for files and directories is strengthened by the different restrictions imposed by security constraints in the system. Since application specific resolvers are arbitrary untrusted programs, not only must all their file system accesses be verified but they must also be disallowed from executing on the trusted Coda servers. In contrast, directory resolution may require examination and modification of regions of the file system for which the user has insufficient access privileges. A failed resolution due to insufficient privileges for a directory high in the hierarchy would significantly reduce availability. Therefore, directory resolution must be able to proceed without requiring special access privileges. This suggests that directory resolution must be performed on the servers. As a result files are resolved at the client but directories are resolved by the servers. We now describe the two strategies in the following subsections.

### 2.4.2 Overview of Directory Resolution

Directory resolution is triggered by Venus at one of the servers in the AVSG. This server, called the coordinator, leads all the other servers in the AVSG through a multi-phase protocol that performs the five subtasks of resolution mentioned in section 2.4. Once resolution completes, an error code indicating its success or failure is sent back by the coordinator to the Venus that triggered it. If resolution completed successfully, all the replicas of the directory are identical. Venus must re-fetch the directory's contents from any member of the AVSG to service the suspended file system request.

For the purpose of resolution each server maintains a data structure, called the *resolution log*, along with each directory replica. The resolution log uses *operation logging* to record all updates made to the directory. The log is used in the first four subtasks of resolution, i.e. deducing, propagating, checking and replaying the set of partitioned updates at the corresponding replica. Each log provides the exact sequence of updates in chronological order. Therefore, partitioned updates can be found by simply comparing logs. The log records have enough information to check and replay each partitioned operation at sites that missed that operation. Therefore, propagating the resolution logs of each replica to all VSG sites is sufficient to complete resolution.

The directory resolution protocol consists of four phases. During the first two phases, the coordinator collects the resolution log from each replica. During the third phase, the logs are distributed to all AVSG members who then deduce the updates they missed, check for correctness and replay them. All the partitioned operations that are replayed are executed atomically, for fault-tolerance reasons, at each AVSG site. During the fourth phase, the directory replicas are either marked as being equal or with an inconsistency flag. Details of this protocol are provided in chapter 4.

### 2.4.3 Overview of File resolution

Section 2.4.1 described why a system-wide policy for resolving files isn't possible in a Unix environment. In practice, concurrent updates in multiple partitions are rare. Therefore, when file replicas differ, it is often due to updates only to a subset of the replicas in a single partition. There is at least one replica containing the latest version of the data. File resolution, in this instance, could simply *force* or propagate this data to the replicas containing stale data.

The Coda design for file resolution realized this early on and therefore used a simple light-weight mechanism, based on *version-vectors,* to solve the frequently occurring case. To address the less frequent and harder case of divergent file replicas, it uses a more complex mechanism based on *application-specific resolvers* (ASR). The former mechanism is purely a syntactic method of detecting concurrent writes in multiple partitions whereas the latter mechanism is a semantic method that uses knowledge of the data contained in the file to perform the resolution. The following sections discuss each of these sub-cases separately.

#### 2.4.3.1 Version-vector Techniques

The use of version-vectors to detect concurrent writes on the same file in multiple partitions was first proposed by Locus [39]. Each replica of the file has a vector associated with it containing as many elements as the number of sites in the file's VSG. Each vector counts the number of updates made at each VSG site. Concurrent updates in multiple partitions can be detected by comparing the version vectors.

When two vectors have identical elements, the corresponding replicas are also equal. If every element of a vector A is greater than or equal to the corresponding element of another vector B, then A *dominates* B. In this case, the replica with vector A has the later version of the file's data. If some elements of A are greater than and other elements are less than the corresponding elements of B, then vector A is *conflicting* with vector B. In this case, the replicas are diverging.

As in directory resolution, Venus invokes resolution for a file at one of the servers in the AVSG. This server collects and compares the version-vectors from all the replicas in the AVSG. If a replica exists, whose vector dominates the vector of all other replicas, then its vector and contents are distributed to all AVSG members. However, if the version-vectors are found to conflict, application-specific resolution must be performed.

#### 2.4.3.2 Application Specific Resolvers

Application-specific resolvers are programs, provided by the application writers, that understand the structure and semantics of data contained in the application files. These programs use this knowledge to resolve diverging replicas of the application's files. Thus, very broadly, automating the resolution of diverging file replicas amounts to finding and running the *correct* ASR that understands the semantics of data contained in that file. It is the ASR's responsibility to perform the five tasks of resolution mentioned in section 2.4. If no ASR is found or if the file is unresolvable it is marked with the inconsistency flag and must be repaired manually.

There are several ASR related issues that the system needs to address.

- **How is an ASR bound to its file or group of files?** The system must have some method by which a user can specify *a priori* the ASR corresponding to each or a group of files. This mechanism is needed to maintain the transparency of resolution.
- **Where does an ASR run?** Since the ASR can be any arbitrary program, it cannot be trusted. Therefore, the system must be careful in deciding where to run the ASR. At no cost should the security of the system be compromised.
- **How is replication exposed to an ASR?** As mentioned before, Coda provides a one-copy view of the file system even though it is replicated. However, this is one instance when the replication needs to be exposed. The ASR needs some mechanism to name and access various replicas of a file.
- **How is the execution of an ASR made fault-tolerant?** The ASR may consist of multiple programs that write out multiple files. If only partial results get written out due to crashes, then another program is needed to correct the errors of the ASR when the system recovers. To simplify this, the system should guarantee the atomicity of the results of the ASR.
- **How does the system cope with multiple machine types?** Coda files are shared across many different machine architectures. Since executable programs are now associated with the file store, it is important for the system to provide some mechanism for specifying and executing the correct binary for each machine type.

Answers to these questions are deferred until chapter 5 of the thesis which describes the ASR mechanism in detail.

## 2.5 Manual Repair

Manual repair of an object is needed whenever it has diverging replicas that cannot be resolved successfully. The object has been marked as inconsistent, and normal applications cannot access the data it contains. To aid in the manual repair, Coda provides a *repair-tool* that can be run at any client. The tool has two purposes: first, to make the replicas of the object visible to the user; second, to provide commands for the user to repair the object.

To expose the replicas of the object, the tool has a `BeginRepair` command. When executed, this command causes the inconsistent object, file or directory, to be converted into a *pseudo-directory*. This directory contains as children all the accessible replicas of the object. The children are all read-only so they cannot be mutated via normal Unix applications. However, the user is free to use any Unix tools like `vi` or `emacs` to peruse the replicas.

To repair an object, the tool has a `DoRepair` command which takes as input the name of a *special file*. The `DoRepair` command behaves differently for files and directories. For files, the special file is used as the new version of the file being repaired and is propagated to all members of the VSG. For directories, the special file contains a series of operations to be performed by each VSG member. For fault-tolerance reasons, the operations are performed atomically at the respective servers.

More details of the design and implementation of the repair tool are provided in chapter 6.

# Chapter 3

# Computation Model for Partitioned Operation

The effectiveness of a system in masking failures depends critically on the success of resolution. If the partitioned updates are resolvable, they can be propagated transparently to all the replicas. The key issue is deciding which set of partitioned updates are resolvable. File systems that have previously addressed this issue [19, 38], used *ad hoc* methods and settled for a solution that seemed reasonable. The Coda approach is to formally define a model for the file system and define resolvability in its context. The goal of this chapter is to define precisely the criterion used by Coda to decide which set of partitioned updates are resolvable.

This chapter is divided into three parts. The first part uses examples to intuitively show the kind of partitioned updates that should be resolvable. The second part formally defines Coda's computation model. This model is based on transactions rather than the traditional Unix shared-memory approach and it defines correctness in terms of *serializability*. If file system updates are modelled as transactions on entire Unix objects, then most concurrent partitioned updates aren't serializable. Therefore, the third part of this chapter modifies the basic transaction model so that a majority of concurrent partitioned operations will be resolvable, i.e. serializable.

## 3.1 Resolvable Partitioned Updates

The task of resolving diverging replicas can be achieved in one of two ways: either by undoing the partitioned updates and re-doing them at all replicas or by performing some compensating actions at some or all replicas. Undo-redo of an update requires knowledge of the computation that initiated it. Since Unix doesn't have any means to capture the static representation of a computation, this method isn't feasible in Coda. Therefore, the second method is the only option for resolution without modifying existing applications. Partitioned operations are resolvable only if the compensating actions can be found.

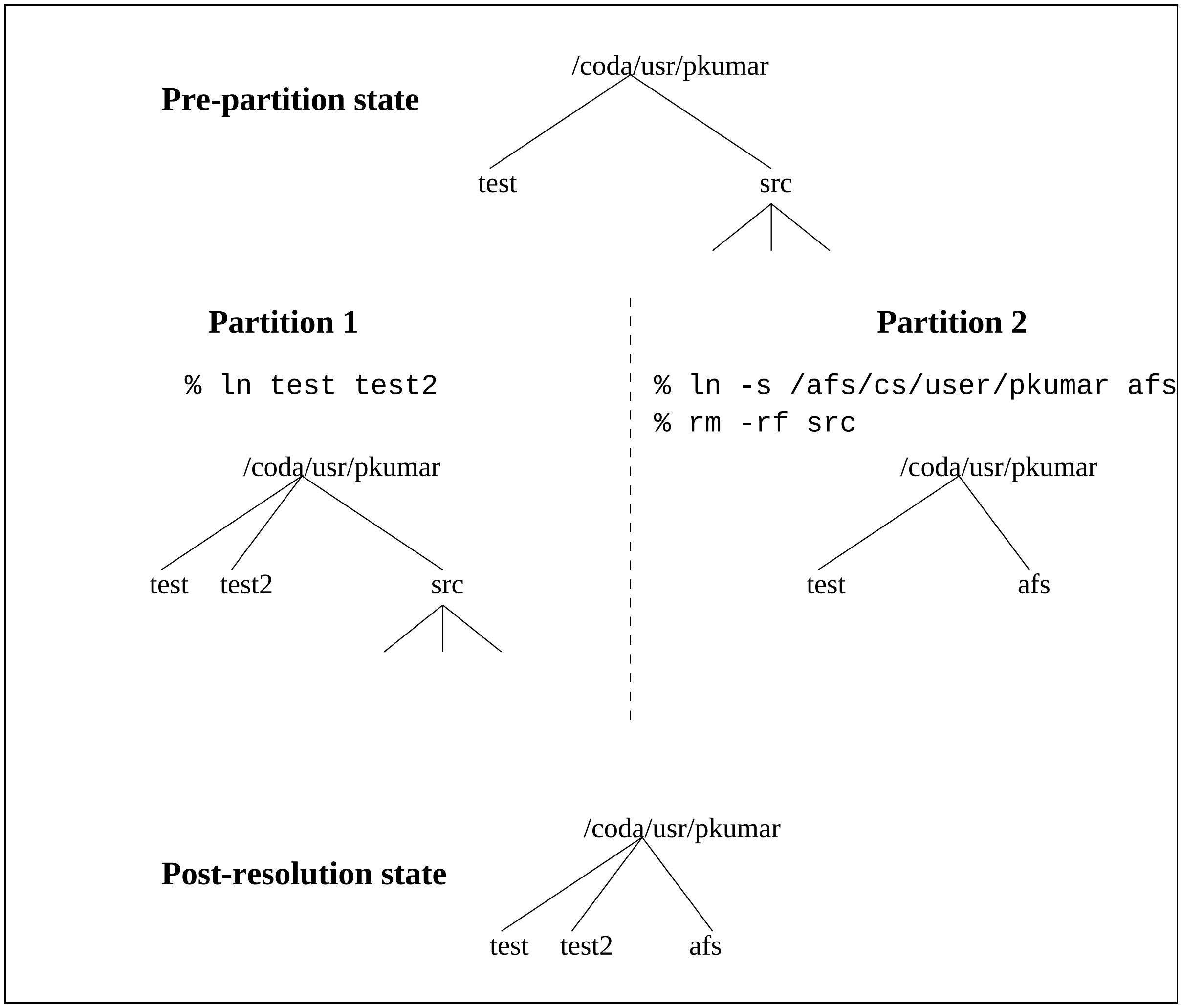
The goal of each compensating operation is to replay the corresponding partitioned update. Intuitively, compensating operations can be effective only if the partitioned updates are mutually independent, i.e. if they modify different entries of a directory or different regions of a file or separate attributes of an object. Let us consider examples of resolvable and un-resolvable partitioned operations.

Consider a directory whose replicas get updated simultaneously by users in two separate partitions. In one partition an entry `foo` is *inserted* and in the other partition an entry `bar` is inserted. In the UFS interface, these updates are independent of one another and succeed as long as there are no duplicate names within the same directory. The compensating action during resolution is that each replica perform the insert operation it missed. The resolved directory replicas would contain both `foo` and `bar`. Figure 3.1 gives examples of some more partitioned "independent" directory operations that are resolvable.

Now, consider a calendar application `cal` that manages appointments in a file. For a user X, `cal` manages a replicated file `cal.personal` which gets updated, sometimes in different partitions, whenever appointments for X are added or deleted. For example, in one partition a new appointment for the following Monday is added by X, and in another partition, an appointment for the following Wednesday is cancelled by X's secretary. When the partition ends, `cal.personal` has diverging replicas. However, the partitioned updates are "independent" since they add or change appointments for different times. Therefore, these updates can be merged and should be resolvable. Note that in this case, the resolution functionality would be provided by an ASR.

Some partitioned updates are not resolvable because they update the same directory entry or same region of the file. Consider the directory example above again: instead of `foo` and `bar` an entry `core` gets inserted in the two partitions. This is a common situation in the Unix environment that arises when a bad binary is executed. When the partition ends, the creation of `core` cannot get propagated to the other replica because Unix directories do not allow duplicate names in the same directory. User intervention is needed to decide which one of the `core` entries can be deleted or renamed.

The above examples show the feasibility of automatic resolution in the Unix environment. They show, only intuitively, the kind of partitioned updates that are resolvable and un-resolvable. The next few sections develop a formal model that defines precisely what kind of partitioned updates are resolvable in the Coda system. The model identifies exactly how much consistency can be traded-off for higher availability. Since, this thesis reuses parts of the model developed by Kistler [25], it only gives a synopsis of the model.



```
% ln test test2
```

```
% ln -s /afs/cs/user/pkumar afs
% rm -rf src
```

Figure 3.1: Examples of Resolvable Partitioned Directory Updates

## 3.2 Computation Model

Since Coda emulates UFS semantics in a distributed environment, it is natural to use the UFS model as the starting point for our definition of correctness. UFS uses the *shared memory* consistency model, i.e. the file system is treated as a block of common store to which read and write requests are made. The system is unaware of the computations to which these requests belong. Further, the effect of a write is immediately visible to all reads that follow. This model when extended to a replicated system becomes the one-copy Unix equivalent (1UE) model. In 1UE, a partitioned execution of a set of computations is considered correct if the final file system state it generates is the same as that resulting from some execution of the same computations on a single Unix host. In this model, two partitioned operations are conflicting if they both access the same object and at least one of them is a write operation. Therefore, 1UE can be trivially achieved in a replicated system with partitionings by restricting the conflicting operations to a single partition or disallowing any mutations (and thus all conflicting operations also) during partitions. However, this is completely antithetical to Coda's goals since it does not provide high availability.

The two restrictions above are sufficient but not necessary for 1UE. They are over zealous in excluding certain partitioned conflicting operations that would otherwise have an equivalent one-copy history. For example, if a file is perused in one partition and edited in another partition then even though a partitioned conflict exists, it is 1UE to reading and then editing the same file in the same partition. Unfortunately, since the system does not know the computation boundaries of the perusal and editing session, it must pessimistically disallow the partitioned mutations to guarantee that non-1UE computations are not allowed.

### 3.2.1 Why Transactions?

The deficiency of the shared memory model is its inability to distinguish conflicting operations that have one-copy equivalents from those that do not. To overcome this deficiency, we can tag the read/write requests with the identity of the computation they belong to. This is made possible by considering each computation as a *transaction*. Each read or write operation can be tagged with the identifier of the transaction it belongs to and the boundaries of the computation are made visible to the file system interface. Once the computation boundaries are exposed, the system can recognize some of the computations with conflicting operations that have one-copy equivalents.

The correctness criterion in the transaction model has traditionally been *serializability*, i.e. the concurrent execution of a set of transactions is equivalent to their serial execution. Serializability is the preferred correctness criterion because it recognizes more concurrent executions (with conflicting accesses) as correct and can be enforced using knowledge of the computation boundaries.

### 3.2.2 System-Inferred Transactions

So how does the system know to which transaction a read/write request belongs? Typically, this is specified in the programming interface which allows programmers to bracket off computations with a begin- and end-transaction, and then all read/write requests identify the enclosing transaction they belong to. However, the Unix programming interface does not provide any transaction support. Adding this support would not be too difficult, but modifying all applications to utilize this new interface is extremely expensive and violates Coda's goal of binary compatibility with Unix.

Our approach therefore, is to use a simple heuristic to map certain sequences of system calls to individual transactions called *inferred transactions*. Without any changes, existing applications are made capable of an expanded set of operations during partitions which isn't possible with the Unix shared memory model. The mapping of the file system calls to transactions is described in detail in [25]. To summarize, Table 3.1 lists the Unix file system interface and Table 3.2 lists the transaction types inferred from that interface.

| | |
|---|---|
| `access` | Check access permissions for the specified object. |
| `chmod` | Set mode bits for the specified object. |
| `chown` | Set ownership for the specified object. |
| `close` | Terminate access to an open object via the specified descriptor. |
| `fsync` | Synchronize the specified object's in-core state with that on disk. |
| `ioctl` | Perform a control function on the specified object. |
| `link` | Make a hard link to the specified file. |
| `lseek` | Move the read/write pointer for the specified descriptor. |
| `mkdir` | Make a directory with the specified path. |
| `mknod` | Make a block or character device node with the specified path. |
| `mount` | Mount the specified file system on the specified directory. |
| `open, creat` | Open the specified file for reading or writing, or create a new file. |
| `read, readv` | Read input from the specified descriptor. |
| `readlink` | Read contents of the specified symbolic link. |
| `rename` | Change the name of the specified object. |
| `rmdir` | Remove the specified directory. |
| `stat` | Read status of the specified object. |
| `statfs` | Read status of the specified file system. |
| `symlink` | Make a symbolic link with the specified path and contents. |
| `sync` | Schedule all dirty in-core data for writing to disk. |
| `truncate` | Truncate the specified file to the specified length. |
| `umount` | Unmount the file system mounted at the specified path. |
| `unlink` | Remove the specified directory entry. |
| `utimes` | Set accessed and updated times for the specified object. |
| `write, writev` | Write output to the specified descriptor. |

Table 3.1: 4.3 BSD File System Interface

An obvious disadvantage of this model is that the actual transaction boundaries may differ vastly from those of the inferred transactions. As shown in Figure 3.2, when an actual transaction spans multiple inferred transactions, the system may wrongly declare a non-serializable execution (with respect to the true transactions) to be serializable. However, this is not a major problem

| | |
|---|---|
| `readstatus`[*object, user*] | `access` \| `ioctl` \| `stat` |
| `readdata`[*object, user*] | (`open read`$^*$ `close`) \| `readlink` |
| `chown`[*object, user*] | `chown` |
| `chmod`[*object, user*] | `chmod` |
| `utimes`[*object, user*] | `utimes` |
| `setrights`[*object, user*] | `ioctl` |
| `store`[*file, user*] | ((`creat` \| `open`) (`read` \| `write`)$^*$ `close`) \| `truncate` |
| `link`[*directory, name, file, user*] | `link` |
| `unlink`[*directory, name, file, user*] | `rename` \| `unlink` |
| `rename`[*directory1, name1, directory2, name2, object, user*] | `rename` |
| `mkfile`[*directory, name, file, user*] | `creat` \| `open` |
| `mkdir`[*directory1, name, directory2, user*] | `mkdir` |
| `mksymlink`[*directory, name, symlink, user*] | `symlink` |
| `rmfile`[*directory, name, file, user*] | `rename` \| `unlink` |
| `rmdir`[*directory1, name, directory2, user*] | `rename` \| `rmdir` |
| `rmsymlink`[*directory, name, symlink, user*] | `rename` \| `unlink` |

The first line of each group shows the transaction type and the second line shows the sequence of system calls that map to it. The notation used in the second line of each description is that of regular expressions; i.e. juxtaposition represents succession, "$*$" represents repetition, and "|" represents selection.

Table 3.2: Coda Transaction Types and System Call Mapping

in practice. First, the chance of concurrent activity leading to non-serializable behavior is rare in practice. Second, Unix itself allows non-serializable behavior since it does not provide any support for concurrency control. Replication and partitioned activity expand this window of vulnerability for non-serializable behavior to the length of the partition rather than just the duration of the transaction. However, users typically monitor partitionings with special purpose tools and use external synchronization means like telephones during partitionings to ensure serializable executions.

```
Initial values, (X,Y) = (0,0)

      T1:  Begin_trans.
                             T2:  Begin_trans.
             (T1.1)Read X                              |
                                    (T2.1)Read Y       |
             Y = X + 1                                 |  Time
                                    X = Y + 2          |
             (T1.2)Write Y                             |
                                    (T2.2)Write X      |
           End_trans.                                  v
                                  End_trans.

Final values: (X,Y) = (2,1)
```

This figure shows the interleaved execution of two transactions T1 and T2. Assume X and Y correspond to real files and that file `open`, `read`, `write`, and `close` are inserted as necessary. In the inferred transaction model the individual file reads and writes would be separate transactions. The final values (X,Y) = (2,1), are a result of executing transactions T1.1, T2.1, T1.2, and T2.2 serially. This is not equivalent to either of the serial schedules (T1,T2) or (T2,T1).

Figure 3.2: A Non-serial Execution that is Serializable with Inferred Transactions

### 3.2.3 Enforcement of Correctness within a Partition

This section describes how Coda enforces correctness of transactions within a partition. To simplify the discussion, assume a partition consists of one server connected to multiple clients. Transactions are initiated and executed at the client but are committed at the servers. Correctness of a transaction is enforced at the time of its commitment. Its serializability is verified by using a mechanism called *certification*. Each object has a *version identifier*, which uniquely identifies the last transaction to modify the object. Certification of a transaction succeeds only if the version identifier of each object it accesses hasn't changed since the start of the transaction. In other words, $C_i$, i.e. the certification of transaction $T_i$, succeeds only if the following is true for all transactions $T_j$ that were committed since $T_i$ started.

1. read-set($T_i$) $\cap$ write-set($T_j$) $= \phi$
2. write-set($T_i$) $\cap$ write-set($T_j$) $= \phi$

| Partition 1 | Partition 2 |
|---|---|
| `T1:  if (Y % 2 == 0) X++` | `T2:  if (X % 2 == 0) Y++` |

Assume that X and Y correspond to individual files, and that file `open`, `read`, `write`, and `close` calls are inserted as necessary. The 1SR-ness of partitioned exection of T1 and T2 depends upon the pre-partitioning state of the system. If either X or Y (or both) are odd, then the partitioned execution is 1SR. If both data items are even at the time of the partitioning, however, the partitioned execution is not 1SR. Since a syntactic approach does not understand the logic of the computation, it will always declare this to be a non-1SR history.

Figure 3.3: Partitioned Transaction Example where 1SR-ness is Data-Dependent

Note that write-set($T_i$) $\cap$ read-set($T_j$) may be non-empty. $T_i$ is still certifiable because its write can be serialized after the commitment of transactions $T_j$.

Each transaction is executed in three steps. First, Venus makes sure it has a valid copy of each object read or written by the transaction and records their version identifiers. An object is valid if its callback promise isn't broken. Second, Venus performs the transaction's reads and writes on the cached copies. Finally, the transaction's read and write sets are certified by the servers. If certification succeeds, the values and version identifiers of objects in the transaction's write set are updated, the callback promises on all other clients are broken and the transaction is committed. If certification fails, the transaction must be restarted at the client.

A transaction with updates is certified and commited at the servers. Servers use locking protocols for concurrency control between transactions being certified simultaneously. Read only transactions are certified at the client by verifying the validity of callback promises on all objects read. The second certification condition is trivially true for read only transactions.

The above protocol for executing transactions in a single partition certifies only those histories that are serializable. To check the serializability of a transaction $T_i$, consider only those transactions $(T_j)$, that are executed concurrently with $T_i$. All transactions that were committed before $T_i$ started and start after $T_i$ is committed, are serializable before or after $T_i$ respectively. The two conditions for certification to succeed guarantee that $T_i$ will succeed only when it has no contention with, and hence is serializable with, all concurrent transactions $T_j$. If there is some contention, i.e. $T_i$ and $T_j$ have conflicting operations on the same data item, one of them will be aborted. This protocol is equivalent to the optimistic concurrency control protocol whose correctness is proved formally in [27].

## 3.3 Transaction Specification for High Availability

Including replicated data within the framework of the transactional model translates the correctness criterion from serializability to one-copy serializability (1SR). A concurrent, possibly partitioned, execution of transactions is 1SR if it is equivalent to their serial execution on non-replicated data. One copy serializability recognizes more partitioned executions (with conflicting accesses) as correct than the traditional 1UE model. Determining exactly those transactions that are 1SR requires knowledge of their intra-partition ordering, the specific data accesses they make, the pre-partition state of the system and the logic of the computation. Collecting and analysing this knowledge may be intractable and sometimes infeasible. Therefore, only the syntactic representation of transactions, i.e. the *history*, consisting of the names and access methods (read or write) of all data items, is used to efficiently decide if a multi-copy transaction history is 1SR. Syntactic approaches are restrictive in that they recognize only a subset of the 1SR histories. Figure 3.3 shows an example 1SR history that cannot be recognized by a purely syntactic approach.

It is important for the syntactic specification to reflect a transaction's true behaviour. This specification is used to decide which partitioned histories are 1SR, and therefore resolvable, which in turn affects the availability offered by the system. Omission of data accesses from the specification may wrongly admit some non-serializable histories in the set of 1SR histories. Similarly, wrongly including data in the specification that is not accessed by the transaction may lead to *false conflicts*, i.e. some serializable histories are declared as non-1SR.

In the inferred transaction model, false conflicts can occur if data accesses are *under-specified*, i.e. the specification names not only the actual item accessed but also some other neighboring items. For example, if the specification of the `chmod` and `chown` transactions indicates that the entire object (directory or file) is written, then a partitioned `chown` and `chmod` transaction on the same object would appear to have a write/write conflict even though they updated different fields of the same object. To avoid situations like this, Coda inferred transactions are specified at the sub-object level. At this level, the attributes and elements of a data object are all independent items. Figure 3.4 shows the description of all three data types: files, directories and symbolic links. All object types have a common base structure called a *vnode* which contains the meta-data of the object. The type specific portion of each object is a fixed size array, with each element being potentially independent.

There are two important parts of the object specification, the modelling of access-rights and directory contents, that significantly effect the availability realized by the system. Each of these is discussed below.

**Access-Rights** In Unix, access to an object is controlled using a tightly encoded vector. Modeling transactions like `chmod` as updates to the entire vector can lead to false conflicts. Consider a `chmod` transaction in one partition that restricts access to a particular group. This transaction would falsely conflict with transactions in other partitions that involved other users, for example, the owner. The Coda model avoids false conflicts, and thus increases availability, by using an explicit access matrix representation (called `rights` in the vnode). Each rights-mutating transaction updates independent elements of the matrix.

```
const int MAXUIDS = 2^32;
const int MAXFILELEN = 2^32;
const int MAXDIRENTRIES = 256^256;
const int MAXSYMLINKLEN = 2^10;

struct vnode {
   fid_t fid;
   uid_t owner;
   time_t modifytime;
   short mode;
   short linkcount;
   unsigned int length;
   rights_t rights[MAXUIDS];        /* implemented as access-control list */
};

struct file : vnode {
   unsigned char data[MAXFILELEN];     /* implemented as list of blocks covering [0..length - 1] */
};

struct directory : vnode {
   fid_t data[MAXDIRENTRIES];        /* implemented as hash table containing only bound names */
};

struct symlink : vnode {
   unsigned char data[MAXSYMLINKLEN];  /* implemented as list of blocks covering [0..length - 1] */
};
```

Figure 3.4: Coda Data Item Description

**Directory Contents** Directories are a list of <name, id> bindings that are independent of one another. The only constraint is that the name must be unique with respect to all other bindings within the same directory. Consider an operation that adds a new binding to the directory. This operation is independent of all other operations and succeeds if and only if no binding with the same name component already exists in the directory.

Modeling individual directory mutations as updates to the entire directory can lead to false conflicts. The Coda specification avoids this by modeling a directory as an array whose size is equal to the cardinality of the set of all possible entry names. Each array element represents the state of one name in the space of all possible names. If a name is bound in a directory, then the array location corresponding to that name contains the id of the object; otherwise it contains a special value $\perp$. False conflicts are prevented in this model because transactions on the same directory conflict only if they access the same name, i.e. the same array location, which is a real conflict. It is important to note that this representation of directories is only used in the model to discuss correctness but a more compact and practical representation is implemented.

The Coda model enhances this basic object specification to expand the syntactically recognizable set of 1SR preserving transactions. It

- redefines certain system call semantics to shrink the write sets of some inferred transaction,
- exploits type-specific information of data items accessed, and
- enforces the 1SR correctness criterion for update transactions and not for read-only transactions.

We discuss each of these points in the following sections.

### 3.3.1 Redefining System Calls

In the BSD Unix file system, each object maintains time attributes like `access-time`, `change-time`, `modify-time` which get updated as a side-effect of certain system calls. Translating these semantics into the Coda transactional model leads to two problems. First, if a `readdata` transaction is required to modify the `access-time` attribute of the object read, then the transaction is no longer a read-only transaction, making it harder to serialize it with other readdata transactions on the same object in another partition. Second, all update transactions on the same object would be in write/write conflict since each one would be writing the same attributes: `change-time` and `modify-time`. In short, the system would have to restrict access to an object to a single partition and in turn reduce the availability significantly.

The Coda model changes the system call semantics, and in turn the transaction specification, so that `access-time` and `change-time` are never modified for any object and `modify-time` for a directory can only be changed via the `utimes` system call. This alleviates the two problems mentioned above and provides greater availability. Read-only transactions continue to be strictly read-only and can be executed unrestricted in any partition. Update transactions that change different names in the same directory are considered independent once again and can be serialized even if they are executed in different partitions.

In our experience, the semantic impact of these changes is small in practice. As evidence, consider distributed file systems that exist today. Most of them do not maintain an accurate `access-time` to improve the effectiveness of caching file meta-data. Yet, few users or applications are affected by this. Maintaining an accurate `modify-time` for files seems to be more important to users and some programs, for example, `make`. However, this isn't true for directory modify-time.

### 3.3.2 Exploiting Type-Specific Semantics

Directory attributes like `link-count` and `length` get updated as a side-effect of transactions like `link`, `rmdir` etc. If these attributes are treated like any other data item written by these transactions, then all independent transactions on the same directory will be in write/write conflict and availability will be significantly reduced.

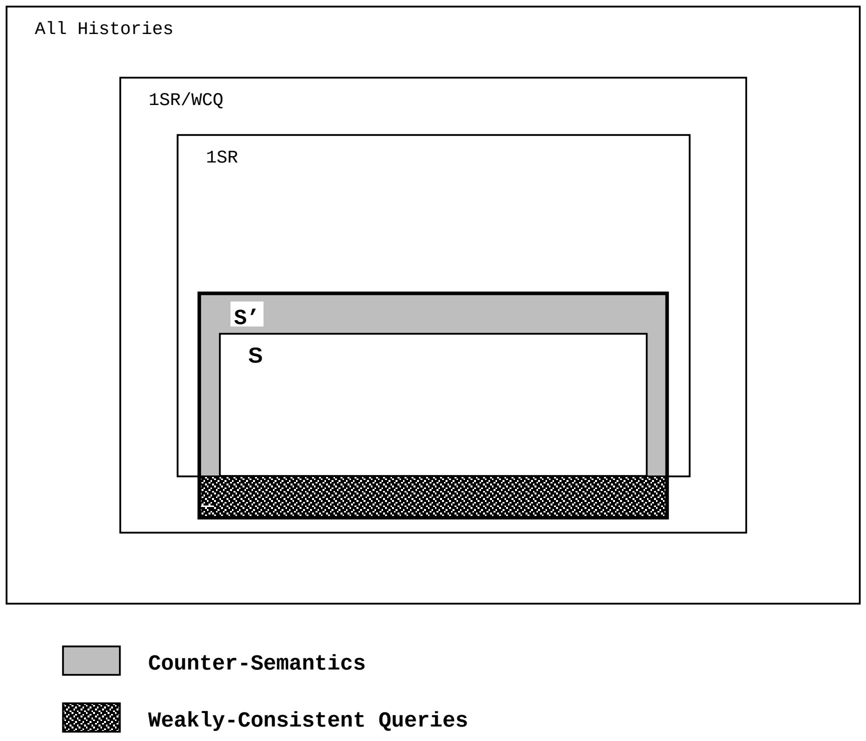
Unlike time-attributes these attributes are fairly important quantities, so they cannot simply be ignored. The key point to realize is that these attributes are used as *counters* that reflect the state of the directory and that from the viewpoint of the transaction the actual value of the counter it writes is not important. Therefore, instead of using the write set to logically reflect the increment/decrement operation, these operations are explicitly specified in the transaction syntax. Transactions that modify the same counter are no longer considered to be in conflict, thus yielding higher availabiity. Each counter's value is correctly compensated at the end of a partition by propagating the "net change" value from each partition to replicas from other partitions.

### 3.3.3 Read-only Transactions

Read-only transactions, or queries, can be exploited to provide higher availability. If a query reads a single logical item, it can be executed in any partition regardless of whether the object it accesses is updated in other partitions. This is because, the query can be serialized before any conflicting partitioned update transaction. All histories that are 1SR without this query will continue to be 1SR with the query as a part of the history.

If a query reads more than one logical item, it cannot always be serialized before all partitioned update transactions that write the object it reads. Figure 3.6 shows one such example query. For higher availability, the Coda model does not enforce 1SR for read only transactions. It guarantees 1SR only for update transactions and treats multi-item read only transactions as *weakly-consistent* queries [15]. Such queries see a consistent system state resulting from a 1SR execution of update transactions but may not themselves be serializable with respect to one another. This is useful for improving availability, especially in the context of file systems where a large percentage of transactions are queries. 1SR guarantees can be provided selectively for certain queries if needed.

Changing the model along the three dimensions, time attribute modifications, directory attribute mutations, and read-only transactions, significantly increases the number of partitioned transactions that are serializable. As a result, the availability offered by the system also increases. Figure 3.5 shows how the syntactically recognizable set of serializable transactions is expanded. A summary of all Coda transaction types is shown in Table 3.3. For each transaction type, the table describes its read-set, write-set and counter operations. This specification is used during resolution to decide if the partitioned transactions are serializable, and hence resolvable.



S' extends the syntactically recognizable set of serializable histories(S) to include two history subclasses: (1) histories which become serializable once counter operations are added to the syntax, and (2) histories which become serializable once queries are allowed to be weakly-consistent. (In the diagram, "1SR/WCQ" stands for *all* histories that are 1SR with Weakly-Consistent Queries.)

Figure 3.5: Expansion of the Basic Syntactically Recognizable History

In summary, the important points of the model are

- queries of any kind are legal in any/all partitions and never reduce availability,
- update transactions are conflicting mostly when they modify the same directory entry, the same data file or the same attribute of an object, and
- intuitively "independent" directory operations within the same directory do not conflict

The next section describes how serializability is checked during resolution.

| Transaction Type | Read Set | Write Set | Increment Set | Decrement Set |
|---|---|---|---|---|
| readstatus[*o*, *u*] | `o.fid, o.owner, o.modifytime, o.mode, o.linkcount, o.length, o.rights[u]` | | | |
| readdata[*o*, *u*] | `o.fid, o.rights[u], o.length, o.data[*]` | | | |
| chown [*o*, *u*] | `o.fid, o.rights[u], o.owner` | `o.owner` | | |
| chmod[*o*, *u*] | `o.fid, o.rights[u], o.mode` | `o.mode` | | |
| utimes[*o*, *u*] | `o.fid, o.rights[u], o.modifytime` | `o.modifytime` | | |
| setrights[*o*, *u*] | `o.fid, o.rights[u]` | `o.rights[u]` | | |
| store[*f*, *u*] | `f.fid, f.rights[u], f.modifytime, f.length, f.data[*]` | `f.modifytime, f.length, f.data[*]` | | |
| link[*d*, *n*, *f*, *u*] | `d.fid, d.rights[u], d.data[n], f.fid` | `d.data[n]` | `d.length, f.linkcount` | |
| unlink[*d*, *n*, *f*, *u*] | `d.fid, d.rights[u], d.data[n], f.fid` | `d.data[n]` | | `d.length, f.linkcount` |
| rename[*d1*, *n1*, *d2*, *n2*, *o*, *u*] | `d1.fid, d1.rights[u], d1.data[n1], d2.fid, d2.rights[u], d2.data[n2], o.fid, o.data[‘‘..’’]` | `d1.data[n1], d2.data[n2], o.data[‘‘..’’]` | `d2.linkcount, d2.length` | `d1.linkcount, d1.length` |
| mkobject[*d*, *n*, *o*, *u*] | `d.fid, d.rights[u], d.data[n], o.*` | `d.data[n], o.*` | `d.linkcount, d.length` | |
| rmobject[*d*, *n*, *o*, *u*] | `d.fid, d.rights[u], d.data[n], o.*` | `d.data[n], o.*` | | `d.linkcount, d.length` |

Note that in the `rename` transaction `o.data[‘‘..’’]` is relevant only when the renamed object is a directory.

Table 3.3: Coda Transaction Specification

```
            Partition 1              |            Partition 2
T1:   read A                         | T3:   read B
      write A                        |       write B
T2:   read A                         | T4:   read A
      read B                         |       read B
```

This execution is not 1SR because `T2` requires an ordering where `T1` precedes `T3`, and `T4` requires one where `T3` precedes `T1`. Such precedence conditions are derivable using two simple rules; `T`$x$ must precede `T`$y$ in an equivalent one-copy history if either:

- `T`$x$ and `T`$y$ are in the same partition, `T`$x$ preceded `T`$y$ in the partition sub-history, and `T`$x$ wrote a data item that was later read by `T`$y$ or it read a data item that was later written by `T`$y$; or
- `T`$x$ and `T`$y$ are in different partitions and `T`$x$ read a data item that `T`$y$ wrote.

Figure 3.6: Multi-Item Queries Violating One-Copy Serializability

### 3.3.4 Checking Resolvability of Updates from Different Partitions

Assume $k$ sites, $S_1, \ldots, S_k$ are partitioned into $n$ groups $P_i, \ldots, P_n$. In each partition, $P_i$, a set of transactions $H_i$, consisting of $T_{P_{i_1}}, \ldots, T_{P_{i_t}}$, is executed. The history of partitioned transactions $H_1, \ldots, H_n$, is resolvable if it is one-copy serializable in the model developed in the previous section. The process of verifying one-copy serializability proceeds as follows. Each site $S_i$, that existed in partition $P_i$, sequentially processes the transaction histories $H_1, \ldots, H_{i-1}, H_{i+1}, \ldots, H_n$. Transactions from each history $H_i$ are processed in the order in which they were committed. Processing a transaction has two steps: it is first certified and then its write-set is tentatively updated. If transactions from all histories can be certified successfully by all sites $S_1, \ldots, S_k$, then $H_1, \ldots, H_n$ is 1SR.

How are transactions certified? Recall from Section 3.2.3 that the serializability of operations within a partition is verified using a version certification scheme. Unfortunately, this scheme is impractical for verifying 1SR because a version identifier is associated with an entire Unix object, not with its independently modifiable data items. The version certification scheme would suffer from too many false conflicts. For example, adding entries named `foo` and `bar` to the same directory in different partitions would change the version identifier of the same directory. In this case, version certification would declare a conflict.

A solution to this problem is to associate a version identifier with each independent data item. However, this solution is impractical. The number of independent data items maybe large (for example, $256^{256}$ in the case of directory entry names) and maintaining a version identifier for each item is infeasible.

Another solution, the one used in Coda, is to augment the version certification scheme with a different methodology, called *value certification*. In this scheme, the old value of each data item accessed by a transaction is stored in the history log. At certification time, the logged value of the data item is compared with its current value. If the two values are the same, certification is successful. Otherwise, the transaction is not certifiable and the history $H_1, \ldots, H_n$ is non-1SR. The correctness arguments of value certification will be discussed in detail in Chapter 4.

# Chapter 4

# Automatic Directory Resolution

The goal of directory resolution is to merge updates made to partitioned replicas of a directory. Resolution is initiated when Venus, while servicing a user's request, detects that a directory has diverging replicas. It requests one of the servers in the VSG to perform resolution and pauses the user's request. This server becomes the coordinator of a multi-phase protocol that performs resolution. If resolution completes successfully, all the replicas of the directory are identical. In this case, Venus fetches the contents of the directory and continues servicing the paused user request. Otherwise, an appropriate error is returned for the user request.

This chapter discusses details of the resolution protocol in four parts. The first section discusses its design rationale. An overview of the protocol is presented next. The third section describes the algorithm used to decide the resolvability of a group of partitioned updates. This section also formally proves certain properties of a group of resolvable updates. Finally, the last section describes details about the implementation.

## 4.1 Design Considerations

Recall from Chapter 2 that Coda uses a log-based approach for directory resolution. This section describes the rationale for using this approach and discusses other design optimizations that make the approach efficient and scalable without compromising security.

### 4.1.1 Log-based Approach

The basic idea behind log-based directory resolution is to record the history of partitioned updates, and use it to propagate updates to all replicas when the partition ends. The history is stored in a data structure called the *resolution log*. This section discusses the design tradeoffs of using a log-based approach for directory resolution.

The main advantage of using a log-based approach is it provides a complete chronological history of all partitioned updates. A complete history allows the set of partitioned updates to be deduced efficiently using only the order of operations in the log without semantic knowledge about the individual operations. Alternatively, a "log-less" scheme could infer the partitioned updates from the state of each replica during resolution and the semantics of all possible operations. However, such schemes introduce additional complexity since multiple operation sequences can lead to the same final state and resolution must be performed with incomplete information. Moreover, such "log-less" schemes eventually need some kind of logging to disambiguate recent deletes from new creates. For example, the presence of an entry only in one replica of a directory could mean one of two things: either the entry was created only at that replica during a partition, or the entry was deleted at all other replicas. This problem can be avoided by recording the delete when it occurs, or in other words by logging. Note that recording only the deletes, does not provide other benefits of logging listed earlier.

An added advantage of logging is the order of operations in the log provides a natural order for verifying their correctness and propagating them to each replica. Furthermore, since all information for resolution is in the log, it can be managed efficiently in a single data structure, and be shared easily between servers during resolution. Storing the information for resolution in a single linear structure reduces external fragmentation and simplifies the garbage collection of stale data.

The main disadvantage of logging is the extra space required for the log. For example, adding a new object to a directory uses space not only to create the object, but also to record the operation in the resolution log. However, this overhead is acceptable in the context of Unix directories for two reasons. First, the space required for logging a directory transaction is approximately equal to the size of a directory entry, typically less than fifty bytes. Second, directory transactions are rare compared to other updates in the system. Chapter 7 will provide empirical evidence showing that the space overhead due to logging is acceptable.

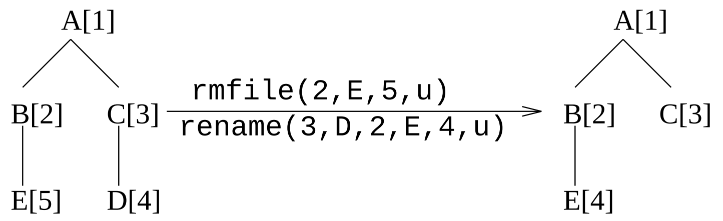
### 4.1.2 Optimizations in Resolution Methodology

Recall from chapter 2 that due to security reasons, directory resolution must be performed at servers rather than clients. However, for scalability reasons, the work at servers must be minimized. Furthermore, since Venus pauses the user request while resolution is being performed, it is important to minimize the latency of each resolution call. This section describes the mechanisms used by Coda to address these concerns.

To improve scalability, Coda performs resolution *lazily*, i.e. a directory with diverging replicas is resolved if and only when a user requests service for it. This reduces the peak demands made on servers immediately after recovery from a crash or network partition. The disadvantage of this scheme is that unresolved partitioned updates may persist until a subsequent crash or partition, thus increasing the chances of stale data being used or a conflicting update being made. In contrast, an aggressive approach would strive to eliminate all unresolved partitioned updates as soon as the partition ends. However, aggressive approaches are susceptible to recovery storms. This reduces scalability and may potentially cause more failures. A compromise would be to perform resolution lazily when triggered by a client, but to conduct aggressive resolution in the background during periods of low server load. Our usage experience so far with Coda has not indicated the need for such a hybrid policy.

A lazy policy also minimizes the latency of resolution. Only the directory for which the user is requesting service needs to be resolved. The total cost of resolving all partitioned updates is amortized over many low latency calls, one for each directory that participated in any partitioned operation.

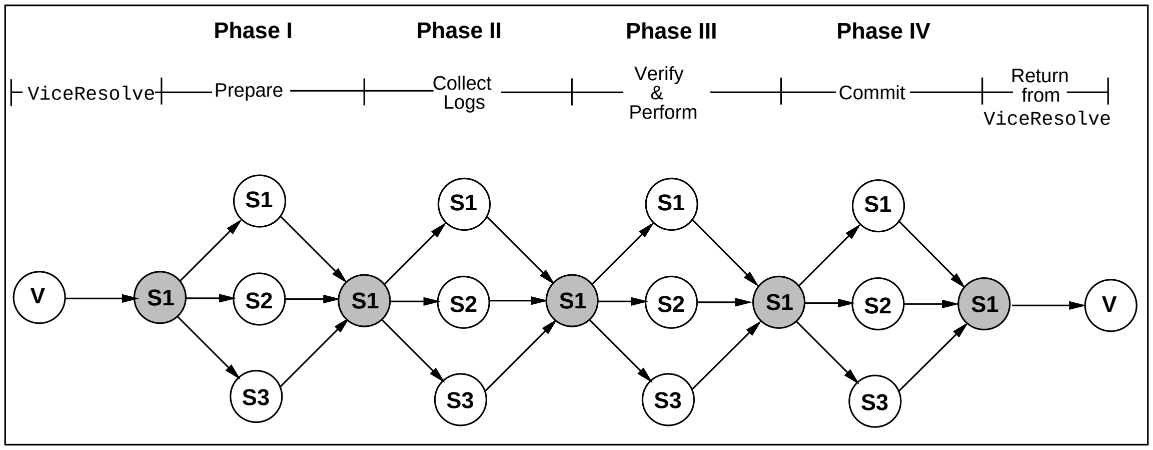
Correctness sometimes requires more than one directory to be resolved at a time. This is because some partitioned transactions, like `renames`, may mutate multiple directories in the same transaction. Figure 4.1 shows one such example. To verify the correctness of such partitioned transactions, it is necessary to simultaneously resolve all directories involved in the mutation. Once again, to minimize latency, only the smallest set of directories with related updates are resolved together. This set of directories, forming a transitive closure, is computed dynamically at resolution time. The smallest possible transitive closure for a directory $D$ is $D$ alone. In this case, all partitioned updates $D$ participated in are independent of other directories. Since transactions cannot span volume boundaries, the largest possible transitive closure includes all directories from the volume in which $D$ resides.



The figure shows a subtree with five objects `A, B, C, D` and `E`. The number in brackets is the identifier of each object. The two transactions `rmfile` and `rename`, corresponding to the Unix commands `rm B/E` and `mv C/D B/E`, are executed at only one replica on behalf of user `u`. In order to resolve directory `B`, `C` must also be resolved, and vice-versa. This is because the `rename` transaction, mutates both `B` and `C` atomically.

Figure 4.1: Transaction Example Requiring Multiple Directories to be Resolved Together

Another requirement for correctness is that the path to the root of the volume, from the directory



This figure shows the sequence of RPCs during resolution. The client **V** invokes resolution by nominating server **S1** as coordinator. The four phases of the protocol are executed at three subordinate servers **S1**, **S2** and **S3**. The node labeled **S1** is shaded when the server is acting as coordinator and unshaded when it is acting as subordinate.

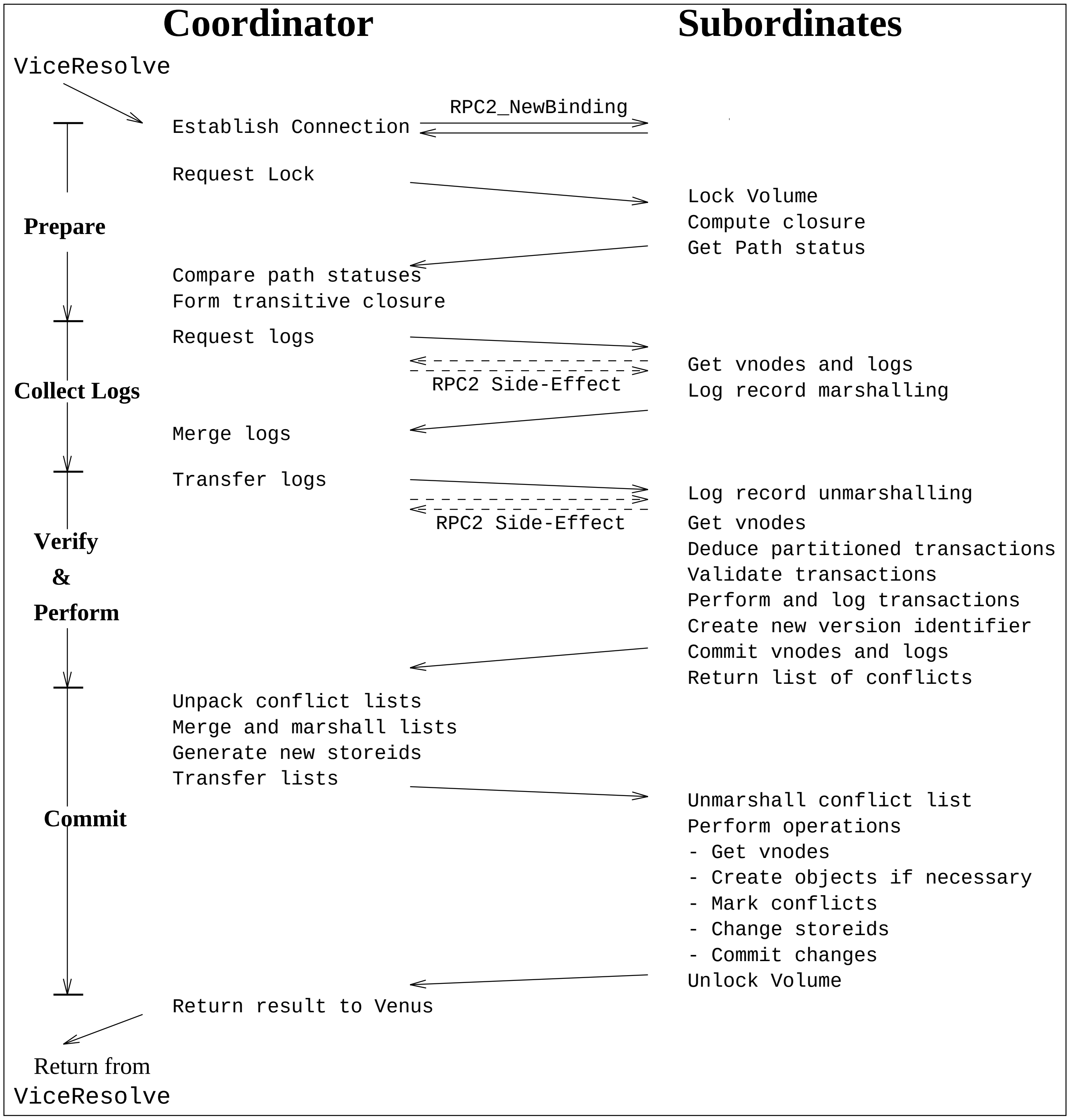
Figure 4.2: RPC Traffic During the Resolution Protocol

being resolved, is conflict-free. This top-down policy ensures that the directory being resolved exists at all accessible replicas. For example, if a directory was created and updated during a partition, its replicas will exist at all sites only after its parent has been resolved. If an ancestor has diverging replicas, the resolution current underway must be deferred till the ancestor is resolved.

To summarize, Coda uses a lazy, top-down resolution scheme for better scalability and correctness respectively. To minimize latency, it limits the objects resolved in each invocation of resolution. As a result, the descendants of a directory being resolved are resolved separately as and when they are accessed by the user.

## 4.2 Protocol Overview

The resolution protocol is coordinator based and is initiated by a `ViceResolve` RPC from Venus to one server in the AVSG. The RPC has a single parameter, the `fid` of the directory with diverging replicas, which is denoted by $D_{res}$. The coordinator, i.e. the server which services the RPC, is chosen randomly by Venus from the directory's AVSG. An alternative strategy could use Venus to coordinate this protocol, just like it manages connected updates. However, this isn't feasible due to the untrusted nature of the client and the need for performing resolution regardless of the protection on a directory.



This figure shows the sub-steps performed at the coordinator and each subordinate during the four phases of the protocol. The solid arrows show the RPC messages exchanged between them. The dashed arrows show the bulk data transfer used for transferring logs. Time moves downwards in this figure.

Figure 4.3: Details of Each Phase of the Resolution Protocol

The protocol design minimizes the number of phases, or message exchanges between servers, for performing resolution. Any resolution protocol requires at least three phases: one to distribute or collect the partitioned state of the replicas, and two to modify and commit distributed state at the different servers. Coda's resolution protocol uses one additional phase to compute the transitive closure of $D_{res}$. An alternative strategy, that avoids computing the transitive closure, would resolve all directories with diverging replicas in the volume containing $D_{res}$. Such an aggressive strategy would reduce the number of phases to three, but impose peak demands on servers after a failure ends. As mentioned before, this is antithetical to Coda's goals of scalability.

The four phases of Coda's directory resolution protocol are: *Prepare*, *Collect Logs*, *Verify and Perform*, and *Commit*. Each phase is performed in lock step at all the subordinate servers. Figure 4.2 shows the message exchange between the coordinator and the subordinates and Figure 4.3 shows the sub-tasks executed during each phase of the protocol. The Prepare phase performs two sub-tasks. First, it checks which servers in the VSG are available to participate in the resolution. Second, it computes the transitive closure of $D_{res}$, i.e. the set of directories that will be resolved in this invocation of resolution. The logs of these directories are collected by the coordinator in the second phase and then redistributed to all subordinates during the third phase. Each subordinate uses the logs to verify the resolvability of the partitioned operations and to re-execute them. The fourth phase, Commit, is responsible for marking the status of each replica so that normal operation can continue. If the partitioned operations were successfully re-executed in the previous phase, then the relevant directory's replicas are marked with identical version stamps and they can be used once again to service user requests. Otherwise, the replicas are prepared for manual repair.

The following sub-sections describe the tasks performed during each phase of the protocol.

### 4.2.1 Phase I: Preparing for Resolution

This phase consists of work done between the time a `ViceResolve` RPC is received by the coordinator and the time each subordinate starts processing logs. Its main purpose is to prepare the servers for performing resolution efficiently. It consists of four steps that are described below.

- **Establishing a connection** - First, the coordinator attempts to establish an authenticated connection with all servers in the VSG of $D_{res}$ by using a `RPC_NewBinding` request. Servers that respond become the subordinates and will use the connection just established for all communication with the coordinator. If some servers are not accessible, i.e. the bind request returns a timeout error, the coordinator does not send any more messages to them.

- **Locking the volume** - After establishing connections, the coordinator requests each subordinate to lock the volume containing $D_{res}$. This lock prevents other resolves or mutating transactions in the same volume from executing concurrently. However, it does not exclude non-mutating transactions, `readstatus` and `readdata`, within the same volume. This maintains high read availability but requires ensuing phases of resolution to lock individual objects if and when they are modified.

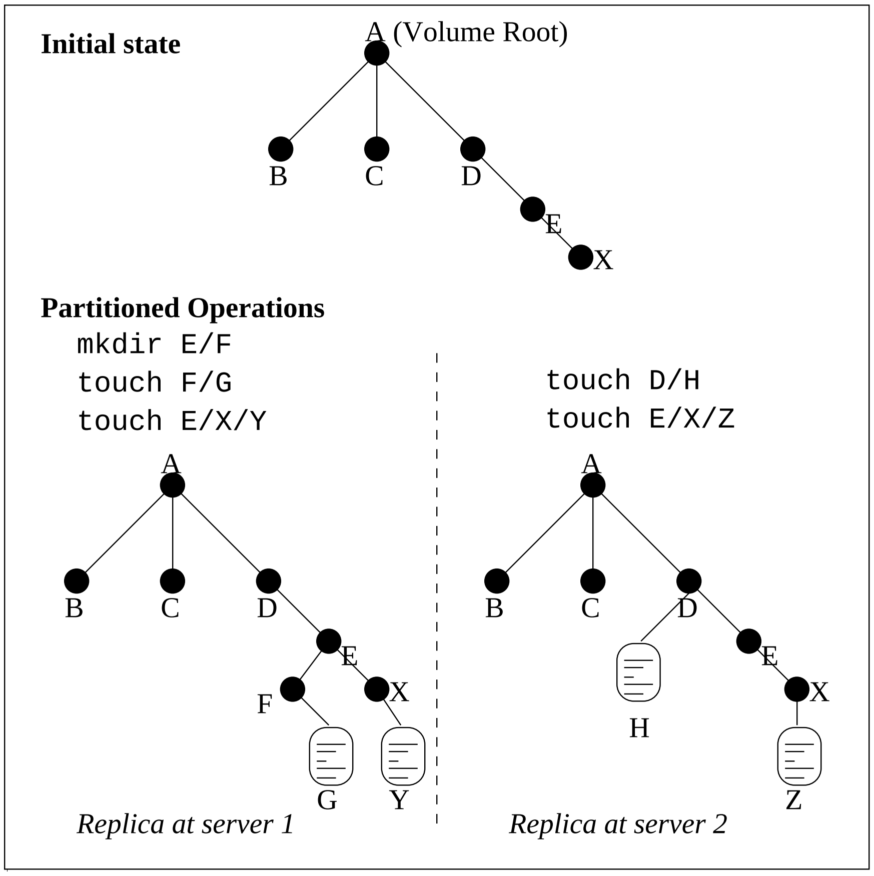
  If a subordinate is successful in locking a volume, it returns three items. First, it returns a list of directories that are in the transitive closure of $D_{res}$. The transitive closure is computed by an iterative process. Initially, the closure consists of one element, $D_{res}$. More directories are added to the set by scanning the resolution log of $D_{res}$ and including other directories it shares a transaction with. In the next iteration, the logs of the newly added directories are scanned and other directories they share transactions with are added to the closure. The process terminates when no directories are added to the closure in an iteration.

  The second item returned by each subordinate is the `fid` and `status` of the ancestors, up to the volume root, of each directory in the transitive closure. The status of a directory is the identity of the last transaction that modified it. Finally, each subordinate returns the total size of the resolution logs for all directories in the volume containing $D_{res}$. This value will be used by the coordinator during the next phase.

  The protocol continues only if all accessible servers lock the volume successfully. Otherwise, the coordinator requests the lock be released at servers that succeeded. The protocol is aborted and an error condition indicating the volume is busy is returned to Venus. Venus retries the call after a short time interval.

- **Comparing status** - If all subordinates lock the volume successfully, the coordinator compares the replicas of all directories in the closure and their ancestors. This is done for two reasons. First, to ensure that the replicas of $D_{res}$ are still diverging because they may have been resolved since the client detected the need for resolution. Second, to enforce top-down resolution. If the replicas of some ancestor of $D_{res}$ have different status values, i.e. they are diverging, then resolution of $D_{res}$ is aborted. Instead, resolution is initiated for the topmost ancestor with diverging replicas. This ancestor is found using the list of `fids` returned by each subordinate.

  There are two special cases the coordinator must address separately. First, some directories in the closure may have diverging replicas as well as an ancestor-descendant relationship. For example, a `mv A/B A/C/D` command during a partition requires directories `A` and `D` to be resolved simultaneously. The coordinator must recognize that top-down resolution cannot be enforced in this case and treat it as an exception. Second, as shown in Figure 4.4, $D_{res}$ may not exist at a subordinate. The subordinate is able to lock the volume but cannot return any status or closure entries. In this case, resolution



This figure shows the state of a volume subtree before and after a partition. If resolution is invoked first for directory F, server 2 cannot return the status of F or its ancestors to the coordinator. However, server 1 returns the status of A, D, E and F. Therefore, resolution is started for F's parent directory, i.e. E. E's resolution will discover that directory D must be resolved first. Note, if resolution is first invoked for X, both servers will return the status of A, D, E and X to the coordinator, and resolution will be invoked for D.

Figure 4.4: Enforcement of Top-down Resolution

is not being performed top-down since the parent of $D_{res}$ has diverging replicas. The coordinator recognizes this condition, finds the highest ancestor with diverging replicas using information returned by the other subordinates, and then initiates resolution for that directory.

- **Computing the closure** - The transitive closure of $D_{res}$ may vary from one subordinate to another since each server has participated in different partitioned transactions. Thus, the set of directories to be resolved together, called the *global transitive closure*, can only be constructed at the coordinator by taking the union of transitive closures returned by the subordinates. To complete the computation, the coordinator must request each subordinate for the closure of all directories in the global transitive closure. This iterative process could take several rounds and stops only when the closures returned by the servers are identical, i.e. taking their union does not add a new directory to the global transitive closure. Since at least one new directory is added to a closure in any iteration, the number of iterations needed to compute the global transitive closure is bounded by the number of directories in the volume.

Every iteration of the closure computation requires a message exchange between the coordinator and the subordinate servers. For efficiency reasons, the coordinator implementation stops the closure computation after the first iteration. The benefits of this optimization come at a cost: the closure thus computed may be incomplete. Consider a volume with two replicas $V_1$ at server $S_1$ and $V_2$ at server $S_2$. Partitioned cross directory renames in $V_1$ require directory $A$ to be resolved with $B$ and directory $C$ to be resolved with $D$. On the other hand, partitioned updates in $V_2$ require directory $A$ to be resolved with directory $C$. During the resolution of directory $A$, the closure computed by the coordinator in the first iteration consists of directories $A$, $B$ and $C$ ($\{A, B\} \cup \{A, C\}$). If the computation is stopped at this iteration, then the closure is incomplete since it does not include directory $D$ even though it includes directory $C$.

In practice, this problem occurs rarely since renames involving multiple directories are infrequent ($<$ 2% of all directory mutations) and the probability of a rename during a partition is even smaller. However, in the event that subsequent phases of the protocol discover this problem, the partitioned updates are considered un-resolvable and the relevant directories marked in conflict. In the example above, directories $C$ and $D$ will be marked in conflict when the servers performing resolution discover a rename involving these two directories in $C$'s log and do not find $D$ in the transitive closure. This technique does not compromise correctness but reduces the usability and availability of the system. However, the reduction in usability is far outweighed by the performance benefits of a simpler and more efficient implementation.

Recall from above that the status of all directories in the transitive closure of $D_{res}$ is returned to the coordinator by each subordinate. Since the closure returned by each subordinate may be different, the status of the replicas of some directories in the global transitive closure may not be available at the coordinator after the first message exchange. In the example above, the status of $B_2$ is not returned by $S_2$ in the first iteration. To obtain the status for such directories in the closure, if any, the coordinator may exchange one more message with the subordinates.

### 4.2.2 Phase II: Collecting Logs

This phase is responsible for collecting logs of all directories in the global transitive closure of $D_{res}$. It consists of three steps.

- **Request log** - This phase begins with the coordinator requesting each subordinate to return the logs of directories in the global transitive closure. The request is made via an RPC named `ResFetchLog`. Since the closure is not known to the subordinates, it is sent as a parameter of the request. In preparation for receiving logs, the coordinator allocates one buffer per subordinate. The size of this buffer is subordinate specific and was obtained in the previous phase. It is the total size of all resolution logs in the volume containing $D_{res}$. Though this is an overestimation of the space needed, it guarantees the log returned by each subordinate will not overflow the buffer.
- **Log record marshalling** - When a subordinate receives the log request, it verifies that the volume lock is still valid. If so, it locks all the directories in the closure and collects their logs. The logs may have a tree structure and have absolute pointers between records. In preparation for sending the logs to the coordinator, they are marshalled into a linear in-memory buffer and the data and pointers converted into a host-independent byte-order. Log records belonging to a directory are stored contiguously in memory and are ordered according to their age, oldest first. Furthermore, each log record includes the subordinate's address so that its source is identifiable once it gets distributed to the other servers. This information is crucial for identifying the partitioned updates missed by each replica.
- **Merge logs** - Once the logs are marshalled into a linear buffer, they are transferred back to the coordinator as a side-effect of the `ResFetchLog` RPC using the bulk-transfer mechanism, SFTP, of the RPC package. The coordinator receives each log in its pre-allocated buffer. Next, it merges the logs in preparation for the next phase. Since the logs are self contained and already in host-independent byte-order, they are merged simply by concatenating them together into a large buffer.

This marks the end of the second phase of resolution. Now, the coordinator is ready to redistribute the logs it has collected, back to the subordinates.

### 4.2.3 Phase III: Verify & Perform

This phase is the heart of the resolution protocol and is responsible for two tasks. First, it checks the resolvability of the partitioned updates to ensure correctness and second, it re-executes the partitioned updates missed at each server. Both these tasks are performed at all subordinates using the logs collected by the coordinator in the previous phase. The steps used to achieve these tasks are described below.

- **Log shipping** - The coordinator initiates this phase by invoking an RPC in parallel to each subordinate. One of the parameters of the RPC is the size of the log that needs to be transferred. Each subordinate allocates a buffer of that size and fetches the log from the coordinator using SFTP.

- **Log record unmarshalling** - Each subordinate unpacks the newly received log into the individual records but maintains certain groupings/orderings on them. Records are grouped based on the server where they were created. Within each group the records are ordered in two ways. First, they are sub-grouped by the directory they belong to. Second, they are sorted in the order of their creation. Therefore, not only does each record contain the server and directory it belongs to, but it also contains a version-stamp that increases monotonically within each volume replica as transactions are executed. The order of records in the sorted log is the order in which they will be examined in ensuing steps of this phase. By the end of this step, each subordinate has the transaction history of each directory replica involved in the resolution.

- **Deducing unique partitioned operations** - The log received by each subordinate contains redundant entries of two kinds. First, it contains records of partitioned transactions that have already been executed by this subordinate server. Second, it contains duplicate records of partitioned transactions that were executed with an AVSG of two or more servers. For resolution, each subordinate ignores records of the former kind and uses only one copy of the latter kind. To recognize the redundant records, the entire log is sorted based on transaction identifiers and adjacent duplicate records removed.

- **Locking vnodes** - Recall that the volume lock acquired in the first phase does not exclude read requests within the volume containing $D_{res}$. Therefore, this phase must lock directories exclusively before they can be modified. Each subordinate scans the record groups and composes a sorted list of directory `fids` that are involved in any transaction. This list is then used to lookup and exclusively lock the directory descriptors, called vnodes, in fid-order. All server threads enforce fid-order locking to prevent cycles in the wait-for graph and thus avoid deadlocks. If the lookup of a vnode fails because the directory was removed or wasn't created at this volume replica, the server still continues processing the list. Later steps of this phase will decide if this condition is fatal.

  Once a vnode has been locked, a copy of it is inserted into a list that will be used by ensuing steps of this phase. Changes to the vnode are made to this copy and committed to stable storage only when the phase completes.

| Transaction Type | Integrity checks |
|---|---|
| `chown`[*directory, user*] | *directory* vnode exists |
| `chmod`[*directory, user*] | *directory* vnode exists |
| `utimes`[*directory, user*] | *directory* vnode exists |
| `setrights`[*directory, user*] | *directory* vnode exists |
| `link`[*directory, name, file, user*] | *file* vnode exists<br>*file* vnode exists in *directory*<br>*name* does not exist in *directory* |
| `unlink`[*directory, name, file, user*] | *file* vnode exists<br>*name* exists in *directory*<br>*name* is bound to *file's* vnode |
| `rename`[*directory1, name1, directory2, name2, object, user*] | $name1 \in directory1$<br>$object \in directory1$<br>*name1* is bound to *object*<br>linkcount of *object* is one |
| `mkfile`[*directory, name, file, user*] | $name \notin directory$<br>*file* doesn't exist |
| `mkdir`[*directory1, name, directory2, user*] | $name \notin directory1$<br>*directory2* doesn't exists |
| `mksymlink`[*directory, name, symlink, user*] | $name \notin directory$<br>*symlink* doesn't exist |
| `rmfile`[*directory, name, file, user*] | $name \in directory$<br>*file* vnode exists<br>*name* bound to vnode of *file* |
| `rmdir`[*directory1, name, directory2, user*] | $name \in directory2$<br>*directory2* vnode exists<br>*name* bound to vnode of *directory2* |
| `rmsymlink`[*directory, name, symlink, user*] | $name \in directory$<br>*symlink* vnode exists<br>*name* bound to vnode of *symlink* |

Table 4.1: Integrity Checks Made for each Directory Transaction Type

- **Transaction checking, performing and logging** - During this step, each subordinate iterates over each log record, validates it and then executes the contained transaction. Validation performs four kinds of checks.

  – Certification: The goal of certification is to ensure that the transaction is serializable with respect to the update history of the objects being modified. It does so by comparing the current value of the objects in the transaction's read-set to their value when the transaction was executed during the partition. The latter value is available in the log record. Consider, for example, a partitioned `rmfile` transaction. The record for this transaction contains the file replica's version number when it was removed. Certification uses this version number to check if the file replica, that wasn't removed, has changed (due to a `store` transaction) since the partitioned `rmfile` transaction was executed. If this is the case then the `rmfile` transaction is not serializable with the `store` transaction.

- Integrity: This check ensures that performing the transaction will not violate any of the system invariants. For example, the name component in a `mkfile` transaction should not exist in the parent directory. Further, the file's vnode should not exist in the volume. The integrity tests for each transaction type are listed in Table 4.1.
- Closure completeness: Recall from Section 4.2.1 that for efficiency reasons the closure computed during resolution may not be complete. This check ensures that all directories accessed by this transaction are in the computed closure and are also being resolved in this instance of the protocol.
- Resource: This check ensures that the transaction can be executed without overflowing the volume's quota.

If a transaction is validated successfully, its mutations are performed tentatively on the volatile copy of the referenced vnodes and directory pages. Callback promises to clients are broken, and a log record for this transaction is created in volatile memory and appended to a tentative list. The identifier of the original partitioned transaction is reused for this log record to ensure this operation will not be repeated in future resolutions. However, the identifier isn't installed on the directory itself since a new one will be generated just before the changes are committed to stable storage.

If validation fails for a transaction, objects in its write set are marked in conflict and the server continues validating ensuing transactions in the log. Validation of an ensuing transaction fails if it reads or writes an item marked in conflict. Therefore, only those ensuing transaction that do not access any object in conflict can be resolved automatically. The advantage of this policy is it reduces the burden on the user by minimizing the number of transactions he needs to resolve manually. An alternative strategy in which resolution stops validating any more transactions after one fails validation would unnecessarily burden the user with the resolution of transactions whose read or write sets include completely different data items.

- **Marking status** - Once all transactions have been validated and performed, each subordinate installs a new unique storeid for each modified object. Having a different storeid for each replica makes the execution of the transactions appear as though they were performed at each server in an independent partition. Therefore, each subordinate can commit its changes regardless of the success of resolution at the other servers. If the coordinator crashes after this phase, the protocol can be simply restarted with a new coordinator.

- **Committing vnodes and logs** - In this step, the subordinate makes the changes to the vnodes and directory pages permanent. All changes, including the addition of new log records to the resolution log, are written to stable storage in a single transaction using a failure-atomic procedure. Vnode locks, obtained at the beginning of this phase, are relinquished as a side-effect of the transaction commit. Implementation details of transactions and their failure-atomicity are described in Section 4.4.1.1.

- **Returning conflicts** - A list of all the objects that were marked in conflict during this phase is returned to the coordinator by each subordinate. Each entry in the list contains the name and `fid` of the object in conflict along with the `fid` of its parent directory.

  The reason for returning the conflicts to the coordinator is to ensure that they are propagated to all replicas of the object. Conflicts arising from a failure in transaction validation, may not be detected at all subordinates. The following example shows how this might happen. In partition $P_1$, a `rmfile` transaction $T_1$, removes a replica of file `bar/foo`. However, in partition $P_2$ a `store` transaction $T_2$, updates a different replica of `foo`. Because of Coda's top-down resolution policy, only $T_1$ is considered during the resolution of `bar.` $T_1$ cannot be certified at the subordinates $\in P_2$, because the status of `foo` has changed due to the prior execution of $T_2$. However, subordinates $\in P_1$ do not detect the conflict because $T_2$ isn't considered during this invocation of the protocol.

### 4.2.4 Phase IV: Commitment of Resolution

The goal of this phase is to finalize the results of resolution. If all partitioned histories of a directory were validated and executed successfully, then its replicas have identical contents but different storeids. The storeids must be matched before a client can service any requests. If the histories couldn't be resolved successfully, then the directory's replicas are still diverging and must be made inaccessible until they are repaired manually. The steps taken by the coordinator and subordinates to achieve these tasks are described below.

The coordinator performs three steps:

- **Unpack and collate conflicts**: The list of conflicts, returned by each subordinate in the previous phase, are merged and duplicate entries are removed.

- **Generate new version identifiers**: The coordinator creates a list of new storeids for directories in the closure without any conflicts. It generates unique storeids by appending a monotonically increasing counter to its Internet address.

- **Marshal lists**: After creating the two lists above, the coordinator marshals them into an in-memory buffer. This buffer is transferred to each subordinate server using SFTP.

Once the buffer is transferred, each subordinate performs three steps:

- **Unpack list of conflicts and version identifiers** - The two lists contained in the buffer are unmarshalled. Every directory in the resolution closure belongs to one of these lists.
- **Perform the operations** - This step is responsible for installing new storeids on the vnodes of directories that were successfully resolved and marking the vnodes of the remaining directories in conflict. For fault-tolerance reasons these changes are made atomically in stable storage using a method similar to that described in Phase III. It consists of the following five sub-steps.
  - **Lock vnodes of objects**: Vnodes of all objects that need to be modified are locked exclusively and copied from stable storage to a temporary copy in memory.
  - **Create objects if necessary**: As shown in the `rmfile` example above, some objects in the list of conflicts may not exist at a subordinate. These objects are created and their names inserted tentatively into their parent directory.
  - **Change vnode status blocks**: The status block of each object's vnode is tentatively changed, either with a new storeid or with a flag indicating the conflict. Further, callbacks are broken for clients that may have cached any of these objects since the end of Phase III. Note that installing a new storeid invalidates all cached copies of the directory and every client is forced to re-fetch it on the next access.
  - **Put objects back in stable storage**: All the changes made in the previous two sub-steps are committed atomically to stable storage in a single transaction. Further, the locks obtained on vnodes are released.
- **Unlock the volume** - Each subordinate releases the volume lock, thus allowing all transactions to access objects in the volume containing $D_{res}$.

The coordinator communicates the success or failure of the protocol to the client as a return-code of the `ViceResolve` RPC. If the code indicates that resolution of $D_{res}$ succeeded, then Venus can fetch its new contents and continue to service the file system request that triggered resolution. In the case of a failed resolution, the error is propagated up to the user application that made the request.

## 4.3 Serializability Issues

The goal of certifying each transaction during Phase III of the resolution protocol is to verify the serializability of partitioned updates. The basic idea is to use the update histories, i.e. the resolution logs, to check if the partitioned execution of each transaction is equivalent to its execution in a serial *global* history, that includes updates from all partitions. The history of partitioned transactions and the serial global history are equivalent iff they produce identical final system states starting from the same initial state. Resolution succeeds only if all servers can verify the serializability of the partitioned transactions.

Previous approaches for testing serializability of partitioned operations, for example Davidson's method [9], verify the existence of one serial history that is equivalent to the complete set of partitioned histories. Such approaches are *centralized* in that they collect and verify serializability of the partitioned histories at one host. Then, the serial history is propagated to all servers. If some server cannot maintain the serial ordering of transactions, it undoes the culprit transactions and re-executes them according to the correct serial order. This isn't possible in the Coda inferred transaction model, since transaction undo/redo information isn't available. Instead, Coda uses a *distributed* approach. Each server tests the equivalence of the partitioned transactions to a different serial history: its own transactions are serialized before partitioned transactions from any other server. Therefore, if resolution is successful, not only are the partitioned histories serializable but they also have multiple serial equivalents. Furthermore, as we will see later, all the serial histories yield equivalent final states.

The following sub-sections describe the methodology for testing the equivalence of histories and formally prove the correctness of the mechanism used by directory resolution. First, we introduce the notation that will be used in the rest of this section.

### 4.3.1 Notation and Assumptions

A server is denoted by $S_i$ and the partition it belonged to is denoted by $P_i$. We assume there are $s$ servers $S_1, S_2, \ldots, S_s$ and thus the number of active partitions with at least one server is $\leq s$. The history of updates performed by server $S_i$ in a partition $P_i$ are denoted by $h_i$ and the transactions comprising $h_i$ are denoted by $t_{i_1}, t_{i_2}, \ldots, t_{i_n}$. Since $h_i$ is serializable (Section 3.2.3), it has an equivalent serial history denoted by $\hat{h}_i$. $\hat{h}_i$ and $h_i$ will be used interchangeably to refer to the same history. $h_{mp}$ is used to denote the multi-partition history consisting of updates from all partitioned histories, i.e.

$$h_{mp} = h_1 \cup h_2 \cup \ldots \cup h_n$$

During resolution, a partitioned transaction is replayed at sites that missed it due to a failure. To distinguish the execution of the original partitioned transaction from its execution during resolution, the latter is denoted by upper-case letters. Thus, the transactions from $P_i$ performed by $S_j$ during resolution are denoted by $T_{i_1}, T_{i_2}, \ldots$ etc. and the history of transactions from $P_i$ is denoted by $H_i$. The "$\cdot$" operator is used to specify the execution order of transactions contained in a serial history. $T_1 \cdot T_2$ specifies that $T_1$ commits before $T_2$ commences. The "$\cdot$" operator can also be used between two histories, for example, $H_1 \cdot H_2$, to specify that all transactions $\in H_1$ are committed before any transaction $\in H_2$ is started. During resolution each server $S_i$

verifies serializability by testing if $h_{mp}$ is equivalent to a serial history denoted by $H_{1C_i}$. [1] The ordering of transactions in each of the serial histories $H_{1C_1}, H_{1C_2}, \ldots H_{1C_s}$ will be defined in Section 4.3.3.

The symbol $d$ will be used to denote data items (files or directories) in the system. A special symbol $d_0$, will be used to denote the value of a data item in the initial or pre-partition state. $d_{Op}$ will denote the value of the data item after executing an operation $Op$. The operation can either be a transaction $t_{i_j}$ or $T_{i_j}$, or be a history $h_i$ or $H_i$. Thus $d_{t_{i_1}}$ is the value of $d$ after executing the first transaction in $h_i$ and $d_{h_i}$ is the value of $d$ after executing all transactions in $h_i$.

### 4.3.2 Equivalence Testing

One method for testing the equivalence of two histories, is using the *view-equivalence* criterion described by Bernstein et. al. [3]. Two histories $h_1$ and $h_2$ consisting of the same set of transactions are said to be view-equivalent if and only if they satisfy the following two conditions. First, each transaction $t_{1_i} \in h_1$ and its corresponding transaction $t_{2_i} \in h_2$ must have the same reads-from relationship. That is, $t_{1_i}$ reads the data item $d$ written by transaction $t_{1_j}$, iff $t_{2_i}$ reads $d$ written by transaction $t_{2_j}$. Second, if $t_{1_i}$ is the last transaction to write $d$ in $h_1$, i.e. $d_{h_1} = d_{t_{1_i}}$, then the corresponding transaction $t_{2_i}$ is also the last one to write $d$ in $h_2$, i.e. $d_{h_2} = d_{t_{2_i}}$. If all transactions in $h_1$ and $h_2$ satisfy these two conditions, the system state they generate is equivalent. Since each transaction reads the same value in both histories, it also writes the same value. Furthermore, since the same transaction writes the final value of a data item modified in either history, the system state is guaranteed to be equal.

**Version-id certification** View-equivalence can be implemented by associating a *version-id*, with each object. A version-id uniquely identifies the last transaction to modify the object. A new version-id is installed every time an object is modified. Therefore, when a transaction commences its view consists of the version-ids of all objects it accesses. View-equivalence can be tested by comparing the version-id of each object accessed by a transaction in the two histories. This process is called *certification*. If all transactions in each history can be certified successfully, then the histories are view-equivalent, otherwise they are not.

An advantage of version-id certification is that its space cost is small and independent of the object's size. In Coda, a version-id of 64-bits is sufficient to uniquely identify all transactions. An overhead of 64 bits/transaction is acceptable if the size of the item being modified is much greater than 8 bytes. This is true for files since they are typically more than a few hundred bytes long and a file mutating transaction rewrites its entire contents. Even though a directory is typically a few kilobytes in size, transactions read/write only the relevant entries not its entire

[1] $C$ in $H_{1C_i}$ is an abbreviation for one-copy and $H_{1C_i}$ should be read as the one-copy history at server $S_i$.

contents. Therefore, the system must maintain one version-id for every entry i.e. $256^{256}$ per directory. Furthermore, the size of a directory entry is typically in the range of tens of bytes and consequently, the space overhead for maintaining version-ids may be significantly high. Maintaining version-ids at a coarser granularity, for example, one per directory, increases the likelihood of false conflicts. As shown in Figure 4.5, certification of multi-partition histories operating on the same directory will fail even if the view of the transaction in the single-partition and multi-partition history is the same.

Pre-partition state: `foo (fid 1.1) (Version-id 249)`

| **Partition 1** | **Partition 2** |
|---|---|
| Transaction-id 275 | Transaction-id 359 |
| `mkdir (1.1, bar, 3.2, 2336)` | `mkdir (1.1, baz, 5.5, 122)` |
| Reads version-id `foo: 249` | Reads version-id `foo: 249` |
| Writes version-id `foo: 275` | Write version-id `foo: 359` |

Neither single-partition history is view equivalent to the multi-partition history

| $H_{one-part_1}$ | $H_{one-part_2}$ |
|---|---|
| T(275) (Reads from T(249)) | T(359) (Reads from T(249)) |
| T(359) (Reads from *T(275)*) | T(275) (Reads from *T(359)*) |

The figure shows two partitioned transactions `mkdir foo/bar` (id:275) and `mkdir foo/-baz`(id: 359). The identity of the last transaction to modify `foo` before the partition is 249, and both partitioned transactions have a reads-from relationship with this transaction. Using object-level version-ids, this reads-from relationship cannot be maintained in a serial history containing both partitioned transactions. Two possible serial histories are shown in the figure. In the serial history labelled $H_{one-part_1}$, transaction 359 reads-from transaction 275, not from transaction 249. Similarly in history $H_{one-part_2}$, transaction 275 reads-from tranaction 359, not from tranaction 249. If version-ids were associated with the individual items of directory `foo`, then each transaction would read-from a transaction that previously wrote the entries `bar` or `baz` respectively. This reads-from relationship would still hold in the single-partition history.

Figure 4.5: Problems with Object-level Version-ids

**Value-certification** Due to the shortcomings of version-id certification, Coda uses an alternative certification strategy, called *value-certification*. The basic idea is to change the certification mechanism to compare the value, instead of the version-id, of each item read by a transaction in the two histories. For this purpose, values of items changed by a transaction are recorded,

| **Partition 1** | **Partition 2** |
|---|---|
| T1: `mkfile (1.1, core, 2.2, 2336)` | T3: `mkfile (1.1, core, 4.5, 122)` |
| T2: `rmfile (1.1, core, 2.2, 2336)` | T4: `rmfile (1.1, core, 4.5, 122)` |

This history is value-certifiable because the value read by a transaction, i.e. the value of `d.data[core]`, in either serial history T1·T2·T3·T4 or T3·T4·T1·T2, is the same as the value of `d.data[core]` read by a transaction in the history above. However, this history is not version-certifiable because its transactions can never have the same reads-from view as transactions in either serial history. This is true even if version-ids are maintained at the directory item level, because each transaction modifies the same item (`d.data[core]`).

Figure 4.6: A History that is Value-certifiable but not Version-certifiable

implicitly and explicitly, in each partitioned history log. As in version-id certification, value certification declares two histories to be equivalent only when their respective transactions read, and consequently also write, the same value for each data item.

Intuitively, logging values for certification should require more space than storing version-ids. However, this isn't the case in practice. Logging records the value of only those items that are modified in the transaction history. Therefore the space overhead is proportional to the number of transactions and the size of the modified data items. Maintaining a version-id, on the other hand, has a fixed overhead for every independently modifiable object, regardless of whether the object is modified in any history. Since the number of transactions is much smaller than the number of objects in the system and the size of directory items is typically in the 10-40 byte range, recording values for certification is feasible.

In comparison to version-id certification, value-certification recognizes more multi-partitioned histories having an equivalent single-partition history. Clearly, all version-certifiable histories are value-certifiable. Furthermore, partitioned histories containing *identity subsequences,* that are not version certifiable, can sometimes be certified by value (Figure 4.6). An identity subsequence consists of transactions $T_i, \ldots, T_j$ such that the state of the system after $T_j$ is committed is identical to the state before $T_i$ began. Figure 4.6 shows an example two-partition history containing identity subsequences. Such histories are a common occurrence in the Unix environment and are caused by the execution of bad program binaries.

Another advantage of value-certification is it doesn't suffer from the false conflict problem. The history log records precisely what has changed and nothing more. Certification fails when a transaction reads different values for an item in the two histories, corresponding exactly to those situations when the histories are indeed not equivalent.

### 4.3.3 Certification Methodology During Resolution

During resolution, the goal of certification is to verify correctness of the $n$ partitioned histories $h_1, h_2, \ldots, h_n$ of each directory being resolved. Recall that the correctness criterion is the equivalence of the multi-partition history $h_{mp}$, to a serial execution of the same histories in a single partition. That is, the final state after executing $h_{mp}$ serially and the states after executing $H_{1C_1}$ at $S_1$, $H_{1C_2}$ at $S_2$, $\ldots H_{1C_s}$ at $S_s$, are all identical if started from the same initial state. Here $H_{1C_i}$ is the serial history consisting of the original partitioned transactions $\in h_i$ followed by $H_1, H_2, \ldots, H_{i-1}, H_{i+1}, \ldots, H_n$. The ordering of transactions in $H_{1C_i}$ is $\hat{h}_i \cdot \hat{H}_1 \cdot \hat{H}_2 \cdot \ldots \cdot \hat{H}_{i-1} \cdot \hat{H}_{i+1} \cdot \ldots \cdot \hat{H}_s$.

The equivalence of $h_{mp}$ and $H_{1C_i}$ is tested by each server $S_i$ using the value-certification scheme described in the previous section. Certification is performed incrementally at each server $S_i$ in the order specified by $H_{1C_i} - \hat{h}_i$. $\hat{h}_i$ does not need to be certified because $S_i$ has already executed that history during the partition. The read-set of each transaction $T_{j_k}$ is certified by value and its updates performed temporarily. This ensures that transactions that have a read-from relationship with $T_{j_k}$ can be certified successfully. If all transactions $\in H_{1C_i}$ certify successfully, then the updates are committed atomically to stable storage. In this case, $H_{1C_i}$ is a single-partition serial equivalent of $h_{mp}$.

Each server $S_i$ tests the equivalence of $H_{1C_i}$ with $h_{mp}$ independently. Resolution succeeds only if all servers successfully certify the serial histories. In this case, the histories $H_{1C_1}$, $H_{1C_2}, \ldots$, $H_{1C_s}$ are all equivalent to one another and are the serial equivalent of $h_{mp}$. If some transactions cannot be certified successfully, the objects they access are marked in conflict. In this case, user intervention is needed to decide the correct serial ordering of transactions.

**Theorem 4.1** *If each server $S_i$ from partition $P_i$ can successfully certify all transactions $\in H_{1C_i}$, then the system state after executing transactions in the order specified by $H_{1C_i}$ is identical at all servers, $S_i, i = 1, \ldots, s$.*

PROOF:
Assume that each server $S_i$ is able to successfully certify all transactions $\in H_{1C_i}$. Then we will show that the final system state is guaranteed to be the same at all servers.

**Lemma 4.1.1** *Each transaction executed in $H_{1C_i}$ reads and writes the same values as it reads and writes when executed in $h_i, i = 1, \ldots, s$.*

Each server $S_i$ doesn't re-execute $h_i$, therefore this claim is trivially true for transactions $\in h_i$. Since each transaction $T \in (H_{1C_i} - h_i)$ is certified by value, it can be executed only if each data item has the correct value. More specifically, $T \in H_j$ can be executed only if the value of each data item $d$ it reads matches the value of $d$ when $t$ was executed in $h_j$. If each transaction reads the correct values, it also writes the correct values since it is deterministic.

**Lemma 4.1.2** *Assuming certification succeeds at all servers, one of the following is true for each data item $d$ in the system:*

*4.1.2.1 $d$ is not modified in any partition*

*4.1.2.2 $d$ is modified in exactly one partition*

*4.1.2.3 If $d$ is modified in more than one partition, then the final value of $d$ in each partition must be the same as the initial pre-partition state, $d_0$.*
Suppose not. Let $d_{h_i} (\neq d_0)$ be the final value of $d$ after it is modified in partition $P_i$. Since $d$ is also modified in another partition $P_j$, let $t_{j_k}$ be the first transaction in $P_j$ to read $d$. The value of $d$ read by $t_{j_k}$ is $d_0$. At resolution time, $T_{j_k}$ will not be certifiable at $S_i$ because the value of $d$ is $d_{h_i}$ and not $d_0$. Contradiction.

**Lemma 4.1.3** *One of the following is true for $d_{H_{1C_i}}$, i.e. the value of a data item $d$ after executing the transaction history $H_{1C_i}$:*

*4.1.3.1 $d_{H_{1C_i}} = d_0$.* This follows from 4.1.2.1 and 4.1.2.3 above.

*4.1.3.2 If $d_{H_{1C_i}} \neq d_0$, $d_{H_{1C_i}}$ is written by the same transaction that wrote the value last in the partitioned history that modifies $d$.* Assume $d_{H_{1C_i}} \neq d_0$. From Lemma 4.1.2, $d$ is modified in exactly one partition, say $P_j$. In $\hat{h}_j$, there is one last transaction to modify $d$, say $t_{j_{last}}$. Since transactions are certified in the order specified by $\hat{H}_j$, the transaction $T_{j_{last}}$ corresponding to $t_{j_{last}}$ is also the last transaction to modify $d$ in $H_{1C_i}$.

The theorem statement follows from the three lemmas as follows. At each server $S_i$, each data item $d$ is in the pre-partition state $d_0$, or is in a different state $d_{h_i}$. If $\exists i, d_{h_i} \neq d_0$, from Lemma 4.1.3 it is clear that $d$ is modified in exactly one partition $P_i$ and is last written by transaction $t_{i_{last}}$. From Lemma 4.1.3, the same transaction ($T_{i_{last}}$) writes $d_{h_i}$ at each server. From Lemma 4.1.1, $T_{i_{last}}$ writes the same value at each server, regardless of whether it is executed as a part of $h_i$ (at server $S_i$) or as a part of $H_{1C_j}$. Thus $d$ has the same final value at each server. □

From the description of the certification mechanism it is evident that if some partitioned histories are declared resolvable they are guaranteed to be serializable. If there are $n$ partitioned histories, there are $s$ equivalent serial histories, $H_{1C_1}, H_{1C_2}, \ldots H_{1C_s}$ (from Theorem 4.1). If the histories are not serializable, then they will not be certifiable or resolvable.

The resolvability of a set of transactions from the same partition is sensitive to the order in which they are certified. It is important that transactions within the same history $h_i$, be certified in the order defined by the equivalent serial history $\hat{h}_i$. Otherwise, the history may falsely be declared to be uncertifiable. Consider a partitioned history that contains two transactions `rmdir foo` followed by `mkdir foo`. During resolution, it is imperative that these two transactions are certified in this order. Certification in the opposite order, i.e. `mkdir foo` followed by `rmdir foo`, is guaranteed to fail.

Unlike transactions from the same partition, transactions from different partitions can be certified in any order without affecting the resolvability of the partitioned histories. This is proved in the following corollary.

**Corollary 4.2** *The order in which the partitioned histories $H_1, H_2, \ldots, H_n$ are certified does not change the resolvability of the updates in $H_1 \cup H_2 \cup \ldots \cup H_n$.*

PROOF:
To certify $H_1 \cup H_2 \cup \ldots \cup H_n$, server $S_i$ certifies $H_j$ after $H_i$ since it has already committed transactions $\in h_i$ (its partitioned history) and they cannot be undone. Similarly, server $S_j$ certifies $H_i$ after $H_j$, since it has already committed transactions $\in h_j$. The partitioned histories are resolvable if and only if certification succeeds at all servers. In other words, $H_1 \cup H_2 \cup \ldots \cup H_n$ is resolvable iff $\forall i, j$ both $H_i \cdot H_j$ and $H_j \cdot H_i$ are certifiable. Thus the order in which the histories are certified does not affect their resolvability. □

In other words, the partition histories are commutative with respect to resolvability. This corollary simplifies the implementation of certification since history logs can be examined in any order as long as the intra-history ordering of transactions is preserved.

Recall from Chapter 3, that a group of histories was resolvable if and only if it was serializable. If the partitioned transactions are such that the histories containing them are serializable but not commutative, then the system will declare a conflict. Figure 4.7 shows one such example and this shortcoming exists due to the inferred transaction model. Since Coda does not provide explicit transactions, it has no way of undoing them. If two histories $H_i$ and $H_j$ are serializable but are not commutative, i.e. $H_i \cdot H_j$ is certifiable but $H_j \cdot H_i$ is not, then it is necessary for $S_j$ to undo some transactions in $H_j$ and redo them in the correct serial order after the transactions from $H_i$. Since transaction undo isn't feasible, directory resolution is conservative and wrongly declares such histories to be unresolvable. In our experience, such situations have been rare in practice and the advantages of a simpler implementation have outweighed these shortcomings. In the future, if the system is extended to support explicit transactions, such situations will become common and it will be necessary to address this issue.

## 4.4 Implementation Details

This section focuses on three main aspects of the implementation, *fault-tolerance* of the resolution protocol, the structure of the *resolution log* and some optimization details.

| **Partition 1($H_1$)** | **Partition 2($H_2$)** |
|---|---|
| $T_{1_1}$: `setrights (1.1, 2336)` | $T_{2_1}$: `link(1.1, foo, 2.3, 2336)` |
| Read Set={1.1, 1.1.rights[2336]} | Read Set={1.1, 1.1.rights[2336],1.1.data[foo], 2.3} |
| Write Set={1.1.rights[2336]} | Write Set={1.1.data[foo], 1.1.length, 2.3.linkcount} |

The figure shows two transactions executed in separate partitions. $T_{1_1}$ corresponds to a `chmod bar` operation, and $T_{2_1}$ correspnds to a `ln bar/baz bar/foo` operation. The `fid`s of `bar` and `bar/baz` are `1.1` and `2.3` respectively. $H_2 \cup H_1$ is serializable ($H_2 \cdot H_1$ is the equivalent serial order) but not resolvable because $H_1 \cdot H_2$ is not certifiable. Certification fails at servers in partition 1, since the value of an item (`1.1.rights[2336]`) read by $T_{2_1}$ has been modified by $T_{1_1}$. Note, that $H_2 \cdot H_1$ is certified successfully at servers in partition 2.

Figure 4.7: Histories that are Serializable but not Certifiable

## 4.4.1 Fault-tolerance of the Resolution Protocol

The resolution protocol is susceptible to three kinds of failures: coordinator and subordinate crashes, and network failures. Since a network failure appears as a coordinator crash to the subordinate server and vice-versa, the discussion below describes only the mechanism used to survive server crashes. Recovering from a server crash is particularly difficult since its persistent store could be in a partially modified and inconsistent state.

An obvious method of providing tolerance to server crashes during the protocol is using *distributed transactions*. The coordinator starts a distributed transaction involving all subordinates during the first phase of the protocol, and commits it during the last phase. The commit procedure typically takes two rounds of messages - a *prepare* to commit request followed by a *finalize* message. All mutations during the execution of the protocol are made within the scope of this transaction. Permanence and atomicity of updates are guaranteed by the distributed transaction facility. As a result, the implementation of the protocol is simplified. In the event of a crash, the protocol can be restarted or completed by the coordinator.

The disadvantage of using distributed transactions is that the failure of one server at critical points in the protocol can block other servers for arbitrary long periods of time. For example, a coordinator crash between the *prepare* to commit, and *finalize* requests might force the subordinates to keep vnodes locked and make them unavailable until the coordinator recovers. This is antithetical to Coda's philosophy of providing the highest possible availability. As a result, distributed transactions are not a viable approach for providing fault-tolerance in the context of Coda.

Coda uses an alternate approach for providing fault-tolerance, consisting of two distinct but complementary techniques:

- First, each server has a mechanism to make *local* updates using a failure-atomic procedure. This mechanism is local in the sense that it provides fault-tolerance independent of the state of other servers.
- Second, each server uses timeouts to avoid *blocking* due to remote server crashes. In the event of a crash of a critical server during the execution of the distributed protocol, the protocol is *restarted* without that server, if possible.

Unlike distributed transactions, these techniques do not hamper availability since the servers never block for arbitrary long periods of time. However, this advantage comes at a cost: each phase of the protocol must be made *idempotent* so that it can be restarted easily.

These two techniques used in Coda are described in greater detail in the following sections.

#### 4.4.1.1 Local Fault-tolerance

Fault-tolerance of mutations during resolution is complicated by two factors. First, multiple data structures are associated with most objects. For example, a directory has data pages and a vnode containing its meta-data. Second, each mutation typically changes both the data and meta-data of the object it modifies. For example, adding a new entry to a directory also changes its `length` or `linkcount` field in its vnode. Atomicity and permanence of these updates is imperative for providing fault-tolerance. A typical way of achieving these in Unix environments is outlined in the following five steps:

- Map the vnodes and data-pages of each object to separate Unix files.
- Make frequent calls to `fsync()` to guarantee all dirty data in the buffer cache is written out to disk.
- Enhance the kernel's understanding of the dependency between operations so that they are written out in the correct order.
- Organize the object layout on disk to minimize the chance of recovering to an inconsistent state.
- Finally, use special recovery code that can recognize all possible states after a crash and has the knowledge to decide which consistent state to move to.

Two problems with this object specific approach are its ad-hoc nature and more importantly its inflexibility. The ad-hoc nature makes it difficult to verify correctness of the algorithm since every combination of operation ordering and fault occurrence must be considered. Inflexibility of the approach makes the task of adding a new data type to the persistent store very difficult. A new data type changes the list of possible states for the persistent store after a failure and may require a complete rewrite of the recovery algorithm.

Instead of using object specific recovery techniques, Coda uses a simple and general fault-tolerance facility called RVM [50] for fault-tolerance. RVM allows an application to dynamically map segments of persistent store into its address space and update it in a transactional manner. Applications see a consistent state as long as all updates are performed within the scope of a transaction and all transactions transition the system from one consistent state to another. Crash recovery is handled entirely by RVM. RVM is implemented as a library and is linked in with applications that need fault-tolerance. As shown in Figure 4.8, the RVM library is interposed between the kernel that provides the functionality for reading/writing to persistent storage and the application that requests the updates.

The next few sections present the details of RVM and its use in the context of the Coda servers.

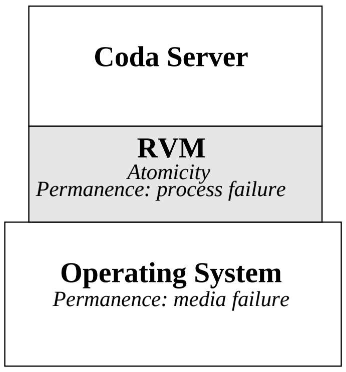


Figure 4.8: Layering of Functionality Using RVM

### The RVM library

RVM is designed for Unix applications with persistent data that must be updated in a fault-tolerant manner. RVM, like its predecessor, the Camelot distributed transactional facility [13], provides the *recoverable virtual memory* abstraction using a write-ahead log to synchronously flush modifications to disk. Recoverable virtual memory refers to regions of an application's virtual address space to which transactional updates can be made. Unlike Camelot, RVM is built without extensive operating system support. It is more portable and efficient than Camelot. However, it provides only two of the three properties of transactions, i.e. atomicity and permanence but not serializability. If needed, serializability can be enforced by the higher layers of the application software. Similarly, RVM does not provide any support for nesting or distribution of transactions and leaves it up to the higher layers to do so. These simplifying assumptions allow the library to provide efficient and fast transaction services. Table 4.2 shows the main routines that interface an application with RVM.

RVM manages the persistent store of the application in *segments*. The backing store of a segment, called the *external data segment*, may be a Unix file or a raw disk partition. An application explicitly maps *regions* of a segment into its virtual memory using the `map` operation. The map operation specifies the segment and the range of virtual memory addresses to which it must be mapped. To prevent aliasing, the `map` call ensures a region is never mapped to more than one area of memory at the same time. Furthermore, mappings cannot overlap in virtual memory. Once an application has finished using a segment, it can be `unmapped` from the virtual address space. Typically, an application maps regions at start up and unmaps them just before termination.

Once a region is mapped, addresses in its range can be used in transactional operations. Atomicity of mutations to that region is achieved by bracketing the updates in a pair of `begin_transaction` and `end_transaction`. A `begin_transaction` returns a transaction identifier (`tid`), that can be used by the application for explicitly aborting the transaction, using an `abort_transaction`, or informing RVM about its updates using a `set_range` call. By using a `set_range`, the application can specify which old-values RVM should record to handle aborts and which values it should write to the log at commit time. The *change records* are appended to the write-ahead log at transaction commit time.

### Structure of recoverable storage at the server

Each Coda server has a single recoverable data segment. The segment is divided into two unequal regions. The first region, the *recoverable static area*, contains recoverable global variables and pointers to the roots of the *recoverable heap* contained in the second region. The recoverable heap is much bigger than the first region and it allows storage to be dynamically allocated. Figure 4.9 shows the layout of the server's address space after the recoverable segment is mapped in. The segment is mapped in soon after the server starts and is unmapped just before shutdown. Since the pointers in the recoverable data structures contain absolute addresses, it is imperative that the segment is mapped at the same address every time. For this purpose, the first page of the recoverable segment contains the address this segment should be mapped to. This information is obtained by temporarily mapping the first page of the segment.

The data stored within a recoverable segment at each server is grouped by volume. As shown in Figure 4.10, a server maintains the *meta-data* of its volumes, files and directories in recoverable

| RVM Interface Routine | Description |
|---|---|
| *Initializing and Mapping Operations* | |
| `initialize` | initializes state internal to the package. |
| `map` | maps a recoverable segment region into the address space of the application. |
| `unmap` | unmaps a previously mapped region. |
| `terminate` | cleans up state before terminating a session with the package. |
| *Transactional Operations* | |
| `begin_transaction` | initiates a transaction. |
| `set_range` | defines a modification range pertinent to an active transaction. |
| `abort_transaction` | aborts an active transaction. |
| `end_transaction` | commits an active transaction. |
| *Log Control Operations* | |
| `flush` | forces buffered log records for committed transactions to the RVM log on disk. |
| `truncate` | applies change records in the RVM log to the recoverable segment(s), then discards those records. |

These are the main exported routines of the RVM library. A few additional routines are exported for setting options, collecting statistics, and so on. Consult the RVM manual [31] for further details.

Table 4.2: RVM Library Interface Routines and Descriptions

memory. Directory contents, and resolution logs are also stored in the recoverable segment. The volume meta-data contains administrative information, like its `name`, `creation date`, `quota` information etc. Since this data has a long life and a fixed length, it is stored in the static recoverable area. Directory and file vnodes contain information like the `fid`, name, length etc. Since the number of vnodes cannot be determined *a priori* and they are often short-lived, they are stored in the recoverable heap. Directory contents and resolution logs are also stored in the heap since their length changes dynamically.

Unlike directory data, a file's data isn't stored in recoverable storage because doing so is impractical: the number of files and the amount of data they contain is significantly higher than directories. Furthermore, the fine-grain byte level atomicity of updates provided by RVM is needed only for directory mutations. Consequently, a server stores a file's data in a UFS *container* file and records the container's inode number, in the file's vnode in recoverable

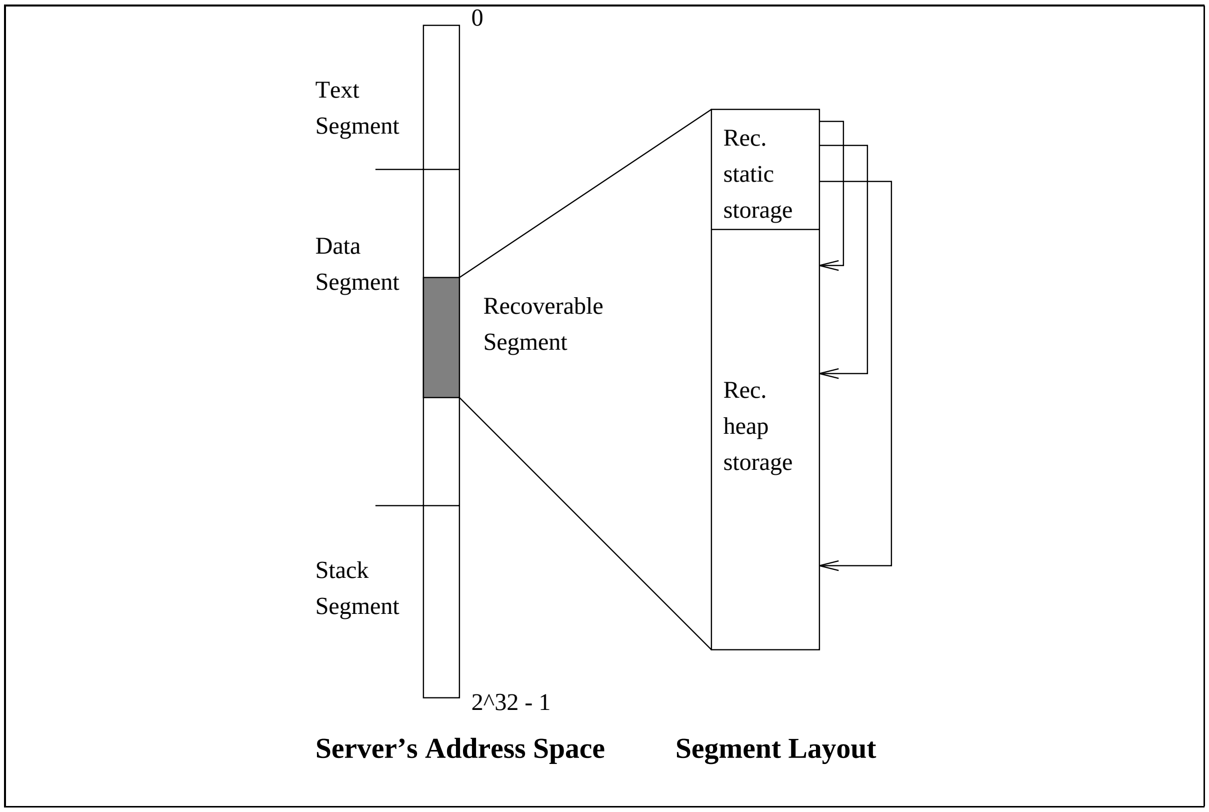
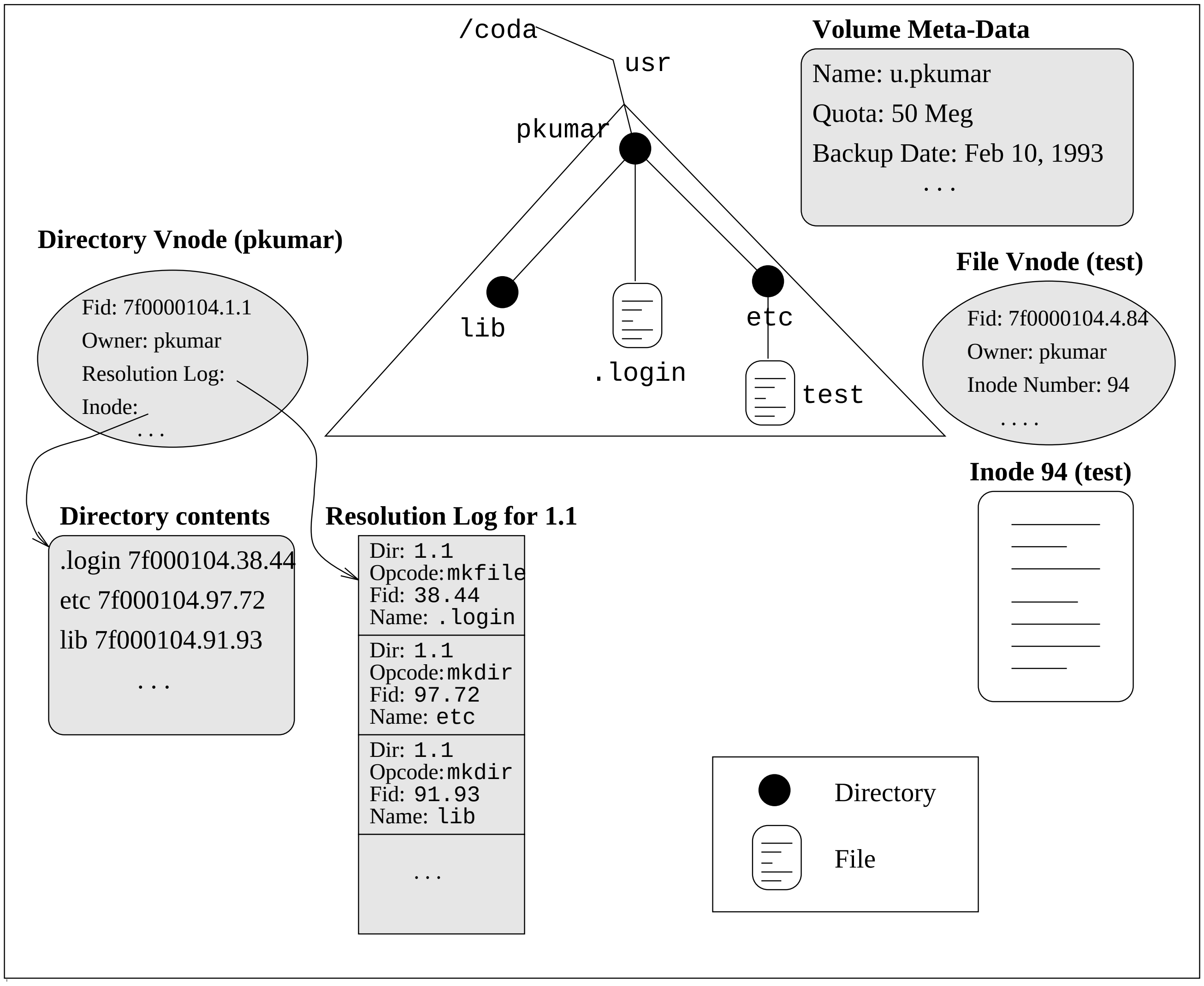


Figure 4.9: Organization of the Server's Address Space

storage. Since files are almost always written in their entirety, a simple shadowing technique can provide the fault-tolerance guarantees. Details of this technique are provided in the next section.

The size of a server's recoverable segment is fixed when the segment is initialized. However, the amount of recoverable storage used by each volume may grow and shrink over its lifetime. If the server exhausts the recoverable storage, another external segment can be initialized and added dynamically. However, in our experience, we have not had a need to extend the size of the recoverable segment. For 2 gigabytes of disk storage for container files, a 75 megabyte recoverable segment has been sufficient – 1 megabyte for the static region and 74 megabytes for the heap. Table 4.3 shows space requirements of three types of volumes: user, object and project. As expected, the ratio of the recoverable storage used by a volume to its total size is smallest for the object volumes since the size of object files and binaries is much higher than average.

The servers currently use a 10 megabyte write-ahead log. To ensure that the log has sufficient space to flush each transaction's changes at commit time, it is truncated periodically in the background. In our environment, truncation is triggered when the log is 50 percent filled. The



This figure shows the meta-data and data layout of a volume named `u.pkumar`. The volume is mounted at `/coda/usr/pkumar`. Shaded blocks represent data stored in recoverable virtual memory. For brevity, the structure of only one directory (the volume root) and one file (test) is shown. The directory's vnode, contents and resolution log are shown in detail. Note that the contents of the file named `/coda/usr/pkumar/etc/test` are stored directly on disk under a Unix file inode whose number is stored in its vnode stored in RVM.

Figure 4.10: Volume Data Structures in Recoverable Memory

time needed to truncate the log varies, depending on the kinds of mutations it contains. This is not a major concern since new transactions can append data to the log while it is being truncated.

| **Volume Name** | **File** | **Dir.** | **Dir.** | **Res.** | **RVM** | **RVM** | **RVM** |
|---|---|---|---|---|---|---|---|
| | **Vnodes** | **Vnodes** | **pages** | **Log** | **usage** | **+UFS** | **Total Disk** |
| **User Volumes** | | | | | | | |
| `u.pkumar` | 300 | 214 | 465 | 76 | 1,055 | 36141 | 2.9 |
| `u.satya` | 440 | 227 | 504 | 74 | 1,245 | 67921 | 1.8 |
| `u.dcs` | 233 | 142 | 322 | 15 | 712 | 62368 | 1.1 |
| **User Object Volumes** | | | | | | | |
| `u.pkumar.objs` | 45 | 42 | 88 | 12 | 187 | 44910 | 0.4 |
| `u.satya.objs` | 98 | 81 | 180 | 21 | 380 | 56326 | 0.7 |
| `u.dcs.objs` | 209 | 57 | 176 | 5 | 447 | 118478 | 0.4 |
| **Project Volumes** | | | | | | | |
| `p.coda.src` | 121 | 71 | 156 | 17 | 365 | 31901 | 1.1 |
| `p.trace` | 67 | 51 | 119 | 19 | 256 | 32137 | 0.8 |
| `p.odyssey` | 61 | 51 | 117 | 12 | 241 | 6551 | 3.6 |

This table shows the RVM usage in Kilobytes for 9 volumes. The space used by each volume's vnodes, directory pages and resolution log is shown in columns 2-5. Column 6 shows each volume's total RVM consumption. Column 7 shows the total space consumed by each volume. This number includes the RVM usage and the space used to store the contents of the volume's files in the UFS. The last column shows each volume's RVM usage as a percentage of the total space it uses. A user volume contains a user's home directory and his personal data/source code files etc. An object volume contains only object files and binaries. Project volumes contain data files, source code files and some binary files.

Table 4.3: RVM Usage for some Volumes in the Coda File System

### Transactional updates

Each phase of the resolution protocol modifies persistent store using the following five steps: lock relevant vnodes, validate/certify updates, perform updates, commit changes and unlock vnodes. These steps are enclosed within a `begin_transaction` and `end_transaction` pair, and all modifications to data in recoverable storage are recorded using RVM's `set_range` facility. This method ensures atomicity and permanence of updates to data items stored within RVM but not for persistent objects stored directly on disk, i.e. containers files. Therefore, each server needs to augment the functionality provided by RVM with data-specific techniques to achieve fault-tolerance. These techniques address two important issues: undoing mutations on container files if the whole phase is explicitly aborted and more importantly recovering from a crash that leaves the persistent store in an inconsistent state.

**Explicit aborts** As mentioned before, the server uses a shadowing technique to undo mutations involving containers. The execution of each phase is set up such that it can be aborted explicitly only during the first three steps. If a file's data needs to be changed, a shadow container is created and used to store its new contents. If the transaction aborts, its updates are "undone" by removing the shadow container. The pointer to the container in the file's vnode is changed only after the third step. Finally, the old container is removed at the end of the phase after the call to `end_transaction` completes successfully. If a file needs to be removed, its container on disk is marked for removal but deleted only at the end of the phase. If the phase aborts explicitly, the delete is undone by un-marking the container.

**Recovering from crashes** The Unix file system does not guarantee permanence of files across system crashes. For example, a file containing a partially written disk block will be removed by the `fsck` utility when the system restarts. Since container files are stored in the UFS, each server must check the state of all such files whenever it recovers from a crash. The purpose of this check is to ensure mutual consistency between the RVM state and the container file state. The module in the server responsible for verifying this consistency is called the *salvager*.

The salvager performs two tasks. First, it checks if each file vnode's container file exists. If not, it creates a new container and marks the object in conflict. This makes the file inaccessible until the problem is corrected by the user, and prevents the damage from spreading. The second task performed by the salvager is removing garbage container files, i.e. containers that are no longer referenced by any vnode. This condition may arise, for example, if a crash occurs soon after a file's vnode is removed but before its container is deleted. [2]

The data-specific techniques described above assume that commitment of a transaction guarantees the permanence of its updates. For example, deleting the marked file after its vnode has been removed assumes that the vnode will never reappear. Making this assumption simplifies the code but also has a disadvantage: the server is unable to utilize some of the performance optimizations provided by RVM. For example, the server cannot use the `no_flush` option for transaction commits. The `no_flush` option allows an application to delay flushing a transaction's mutations to disk and to improve its performance by batching writes of several transactions.

[2]Since the container files do not have any parent in the UFS hierarchy, the `fsck` program on the servers is modified to not garbage collect them.

#### 4.4.1.2 Global Fault-tolerance

This section describes the mechanism used by Coda to tolerate coordinator and subordinate crashes during the resolution protocol. The technique addresses two issues: (a) the recovery of a crashed coordinator or subordinate and (b) a subordinate's tolerance to coordinator crashes and vice-versa.

Neither the coordinator, nor the subordinate store any protocol-specific state in stable storage. Therefore, they recover using the normal server recovery mechanism. A newly recovered server has no knowledge if it was participating in a resolution protocol when it crashed and does not try to re-establish communication with any server. Since a subordinate modifies vnodes in persistent storage during the third and fourth phases of the protocol, care must be taken to ensure that the protocol will function correctly if it is restarted after a premature abort due to a crash.

At each subordinate, updates during the third phase are made in a manner indicating the protocol has not completed. The vnodes are marked with a version stamp unique to each subordinate. Thus, the replicas of the directory being resolved still appear diverging to a client, even if certification has succeeded and their contents are identical. This guarantees that the protocol will be restarted whenever the user accesses the directory again. Marking the vnodes with a unique version stamp is not necessary during the fourth phase, since it is the last phase of the protocol. The protocol doesn't need to be restarted if the subordinate crashes after this phase. However, if the crash occurs before the phase is complete, local fault-tolerance guarantees the server will recover to the state that existed at the end of phase three, and the protocol will be restarted as discussed above.

To maintain mutual consistency between the state of a directory replica and its resolution log, the updates made by a subordinate during the protocol execution are recorded in the resolution log. This invariant is necessary to ensure that a subordinate will not perform an update multiple times if the protocol is restarted after a crash. Local fault-tolerance at each subordinate guarantees that the directory and log updates are performed atomically.

The coordinator avoids blocking due to subordinate crashes, by using an RPC timeout mechanism. A subordinate crash is detected when an RPC made to that server returns a timeout error. The timeout interval in the current implementation is fifteen seconds by default, but can be changed at server startup time via a command-line argument. When a subordinate crashes, the coordinator executes the remaining phases of the protocol with the subordinates that are still accessible. Since a subordinate has no knowledge of other subordinates participating in the protocol, no special machinery is needed at the subordinate servers that are still participating in the protocol.

A coordinator crash does not affect the subordinate's execution. Once a subordinate receives an RPC request to perform a particular phase, it executes that phase in isolation [3] and returns its result to the coordinator when execution completes. Furthermore, the subordinate pessimistically assumes that a coordinator crash might occur and therefore records all its updates in the resolution log. If the coordinator crashes while the subordinate is servicing its request, the result of the request is discarded.

Recall that a coordinator locks the volume, containing the directory being resolved, for the duration of the protocol's execution. Therefore, if a coordinator crashes each subordinate must unlock its volume replica. Otherwise, the restarted protocol will block in the first phase. Since a subordinate never makes an RPC to the coordinator, it uses a daemon thread that probes the latter periodically to ensure it is still accessible. If the coordinator does not respond to the probe, the volume locked on its behalf is unlocked. A problem with this approach is it is not tolerant to errors. For example, a bug in the coordinator's implementation may cause it to enter an anomalous state in which it responds to probes but does not perform any useful work. If the volume is locked when the coordinator enters this state, then the volume will remain locked and unavailable until the subordinate servers are restarted. To overcome such shortcomings, the volume locks are timer based, i.e. the lock expires after a fixed time interval. This limits the duration for which a volume may be unavailable due to an error in lock management. The lock expiration interval must be longer than the time needed to resolve any directory. To safeguard against premature expiration of a volume lock, the subordinate checks the validity of the volume lock at the end of each phase of the protocol. If the lock is not valid, the protocol is aborted and restarted. In the current implementation, the lock expiration interval is pessimistically set to ten minutes. In our usage experience, this interval has been more than sufficient for all invocations of resolution.

The responsibility of restarting the protocol when a coordinator crashes lies solely with the client. A coordinator crash is detected by the client when the `ViceResolve` RPC returns with a timeout error code. If the client can access more than one replica of a directory and they are diverging, it restarts the resolution protocol by chosing one of the accessible servers as the coordinator. Of course, the protocol can proceed only if the volume is not locked.

### 4.4.2 Resolution Log

As mentioned earlier, the resolution protocol uses a data structure called the resolution log that is maintained by each server. The resolution log contains a history of all directory operations performed at a server. Logging directory mutations at the server is feasible and practical because (a) there are only a few and fixed number of possible directory operations and (b) these operations are visible at the server-client interface. As a result, the server can log individual

[3] The subordinate might transfer data from/to the coordinator using SFTP. If the coordinator crashes during the data transfer, SFTP returns with an error condition and the subordinate aborts the execution of the whole phase.

operations efficiently requiring only few tens of bytes per record. The next few sections describe how the resolution log is managed by each server.

#### 4.4.2.1 Log Structure

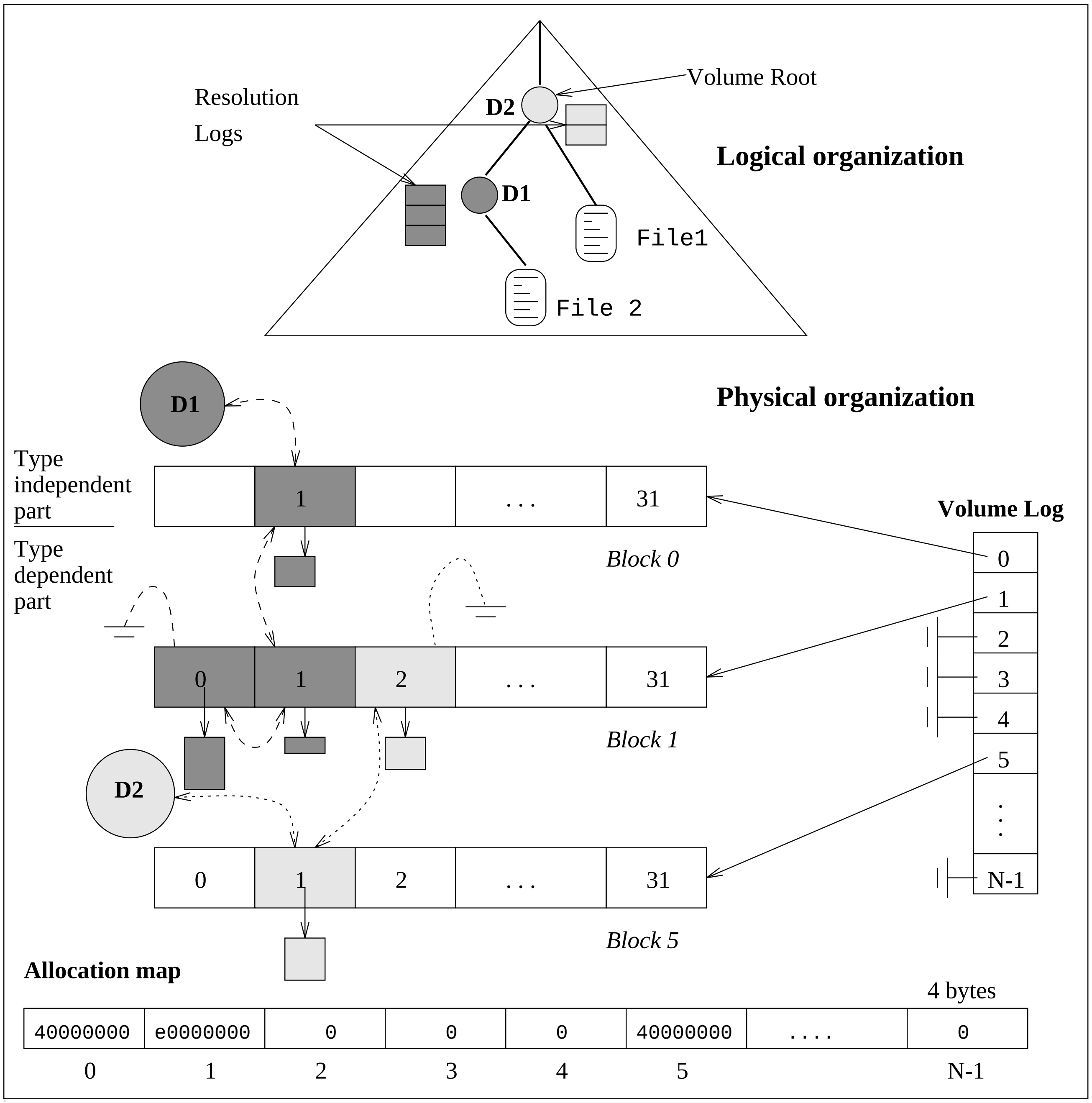
The resolution log is stored in persistent storage so that it can survive crashes. For correctness, log modifications must be made in a fault-tolerant manner. Not only should each modification be permanent but should also be atomic with respect to the directory update it represents. This is achieved by placing both the resolution log and directory contents in RVM and modifying them within the same transaction. Since the resolution log records operations and RVM's write-ahead log records values, placing the resolution log in RVM combines the well-known strengths of operation logging and value logging.

Each server maintains a separate resolution log for each volume replica it manages. The decision to do so was motivated by three considerations. First, a per-volume log achieves a reasonable balance between resource usage and efficiency. A single log per server would have achieved better utilization of RVM by reducing fragmentation, but would have given the server no control over the usage of RVM by individual users. At the other extreme, a per-directory log would have been more efficient since irrelevant entries would not have to be examined during resolution. But that approach would have resulted in much greater internal fragmentation of RVM space. A second consideration is that a per-volume log is consistent with Coda's policy of associating disk quotas with volumes. A final consideration is that the operands of system calls in Coda may span directories but never a volume boundary. Consequently, a volume is the smallest encapsulating unit whose log is guaranteed to contain all the information needed to resolve an update.

The physical organization of the resolution log meets two requirements. First, it makes efficient use of log storage. Second, it supports efficient recording of updates during normal operation, as well as efficient traversal of log entries during resolution.

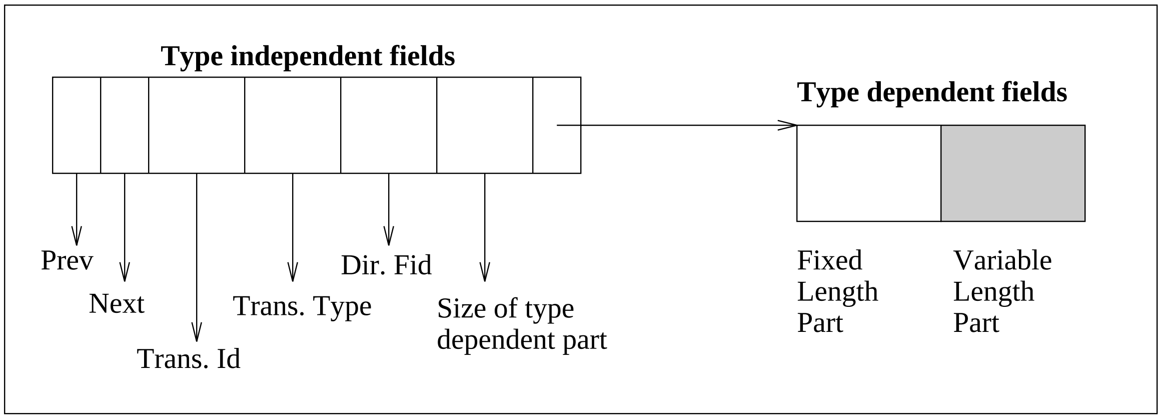
The first requirement is met by organizing the log physically on a per-volume basis and designing it in a manner that conserves storage. The volume log is divided into smaller *blocks*, each containing a fixed number of records. A block is added dynamically when needed, and reclaimed when none of the records it contains are being used. An *allocation map*, stored with each volume log, records the free entries in every block. The algorithm for reclaiming log records is described in Section 4.4.2.3. This organization conserves log storage in two ways. First, it does not require the log to be pre-allocated. Second, it provides a simple and lightweight compaction scheme: blocks are reclaimed independent of the state of adjacent blocks. In the worst case, each block has only one allocated record and the ratio of used space to allocated space would be 1/N, where N is the number of records per block.

The second requirement is met by organizing the log logically on a per-directory basis. A per-directory log is realized as a doubly-linked chain of log entries embedded in the volume



This figure shows the logical (top) and physical organization of a volume's resolution log. The logical organization shows a volume's subtree containing two directories **D1** and **D2** with their respective resolution logs, and two files `file1` and `file2`. Files do not have any resolution logs. The physical organization shows the block structure of the volume's log and the chaining of records belonging to **D1** and **D2**. The allocation map shows the records being used. Note that blocks 2, 3 and 4 have been reclaimed even though block 0, 1 and 5 have valid records.

Figure 4.11: Logical and Physical Organization of a Volume's Resolution Log



This figure shows the type independent fields of each log record. Type specific fields of every record contain data items of two kinds: fixed length and variable length. The latter typically consists of name strings.

Figure 4.12: Log Record Format

log. Figure 4.11 shows the logical and physical organization of the resolution log for a volume. Recording a directory update consists of finding a free entry in the allocation bitmap, setting the bit, filling the fields of the corresponding record and then linking the record at the end of the directory's log. A logical per-directory log is useful, because during resolution it is usually sufficient to examine the log entries of only the directory being resolved. Only on rare occasions, when the transitive closure includes multiple directories is it necessary to examine the logs of other directories.

#### 4.4.2.2 Record Format

There are nine types of log records for the different kinds of Coda directory transactions. As shown in Figure 4.12, each record is divided into two parts: a type independent part and a type dependent part. The type independent part contains fields common to all log records: pointers to next and previous records of the same directory, the corresponding transaction's type and identifier, the `fid` of the directory this record belongs to, and the length and address of the type dependent part. The type dependent part of each record consists of two kinds of items. The first kind consists of fixed length items, like `fids` of objects being deleted or created due to the transaction. The second kind, that are stored at the end of the record, are variable length and consist of names of the mutated children.

Each log record contains the information needed to certify the transaction it contains. This includes the read/write and the increment/decrement set of the transaction. Table 4.4 shows the fields in the type-dependent part of each log record.

| **Transaction type** | **Field names** |
|---|---|
| `chown, chmod, utimes,store` | transaction type, new owner, mode, author, mtime, version information |
| `mkfile` | new fid, owner, name |
| `mksymlink` | new fid, owner, name |
| `link` | fid, version information, new name |
| `mkdir` | new fid, owner, name |
| `unlink` | fid, name, version information |
| `rmfile, rmsymlink` | fid, version information, name |
| `rmdir` | fid, version information, log pointer, name |
| `rename` | type (source or target), other directory's fid fid, version information, deleted target's fid, version information, log pointer old name, new name |

Table 4.4: Log Record Types and their Type-dependent Fields

Three record types, corresponding to the `rmfile`, `rmdir` and `rename` transactions, are more complex than the rest. Records for the delete transactions contain the *state* of the object being deleted, in addition to its name and `fid`. A file's state is captured by its version-vector. A directory's state, on the other hand, is captured by the contents of its log, a pointer to which is stored in the `rmdir` transaction record. The deleted object's state is used during resolution to ensure that the delete transaction is serialized after all other transactions for that object.

The record corresponding to a `rename` transaction is most complex since it has the largest read and write set. If the transaction involves two directories, then a record is inserted in each of their logs. Further, if the transaction removes an object, the state of the deleted object is captured in the record to check for serializability. Finally, `rename` transaction records belonging to the same directory are chained together to make the computation of the transitive closure efficient. Computing the closure of a directory involves finding all objects it participated in a rename transaction with. These objects can be found by following the links in the rename transaction records instead of scanning the entire log of the directory.

#### 4.4.2.3 Finiteness of Log Space

Conceptually, a resolution log contains the entire list of directory mutating operations on a replica since its creation. The discussion so far has ignored the fact that log space is finite. The servers address the limitations on log space in two ways: first, they prevent the log from growing too large and second, they reuse records if the log exceeds a certain limit. The maximum size for each volume's log is set by the system administrator.

**Keeping the resolution log small** If a user-initiated directory transaction is executed successfully at all replicas, then its replicas are guaranteed to be equal. This follows from two observations: first, the directory's replicas were identical when the transaction started, since it was certified successfully at all servers, and second, executing the same transaction at identical replicas results in identical final states. The record for the last successful transaction is identical at all replicas and is called the *latest common entry*, or LCE for short.

Recall from the previous sections that each transaction's log record is used to certify and execute it during resolution. If a directory's replicas are identical then their log records are useless for the purpose of resolution and records prior to the LCE can be discarded. The LCE itself cannot be discarded since it marks the point until which replicas were identical and is used to deduce the set of partitioned updates for each replica during resolution.

Confirmation that a recently spooled log record is the LCE is available from two sources. When user-initiated requests are executed, this information is available during the second phase of the *update protocol*. For updates performed during resolution, the list of sites where resolution was successful is distributed during the fourth phase of the protocol.

Coda's update protocol proceeds in two phases. During the first phase, the client requests each server to perform the update. Each server certifies the transaction, mutates the relevant objects, spools a log record to the resolution log and returns a code to the client indicating success or failure. During the second phase, the list of servers that succeeded in the first phase is distributed to all servers. If the transaction is committed successfully at all servers, then the log record for this transaction is the LCE and all previously spooled log records are discarded. Log space is reclaimed eagerly in the absence of failures and the number of records in a directory's log oscillates between two and one. In the presence of partitions/failures, the log grows as transactions are committed. It is truncated if and when a resolution that spans all replicas is successful.

**Log overflow** A long lasting failure, for example, due to server hardware crashes, or an active user during a partition, can fill the volume log. The problem can be avoided by increasing the volume's quota for the resolution log, but any practical implementation must limit the size to some reasonable level. Therefore, the key question is what does a server do when a volume log becomes full? One approach would be to disallow updates to that volume until all replicas are accessible and a resolve reclaims some log records. However, this is antithetical to Coda's goal of providing the highest availability. The approach used in Coda, is to allow updates to continue by overwriting entries at the head of the log. This causes the LCE of the resolution log of some directories to be overwritten, a condition that will prevent the resolution algorithm from deducing their partitioned updates. In this case, the resolution protocol will mark these directories in conflict. This strategy enhances update availability and provides an easily-understood tradeoff between resource usage and usability: the larger a log, the lower the likelihood of having to resort to manual repair. However, it would be a simple matter to make the choice between disallowing updates and overwriting log entries a volume-specific parameter.

Note, that the directory whose log entries are overwritten becomes unavailable until it is repaired manually. Since the availability of a directory affects the availability of all its descendants, it is important to prevent the log of a directory with many descendants from being overwritten. An example of such a directory is a volume's root directory and Coda's policy overwrites its log only as a last resort. The depth of a directory in the volume is another stochastic measure of the number of descendants – directories deeper in the volume subtree have fewer descendants. Therefore, the optimal log overwrite strategy is volume specific – it depends on the structure of the volume's directory hierarchy and the locality of updates exhibited by users of that volume.

### 4.4.3 Other Details

This section describes four minor details of the implementation: the first is necessary for correctness; the next is an optimization that improves the system's performance; the third increases the system's availability and the fourth improves both performance and availability.

#### 4.4.3.1 Resolving with Incomplete VSG

When resolution proceeds without all VSG members, partitioned updates must be repropagated when other members become accessible. To prevent a site from performing the same operation twice, Coda logs transactions during resolution with the identity of the original transaction. The log entry contains the same information as the original update's entry to ensure correctness of future resolutions even if the site where the original update was performed becomes inaccessible.

Log entries spooled during resolution do not provide the same guarantee as that provided by entries for client-initiated transactions: if two replicas' logs have the same log entry created during resolution, the replicas need not have been identical at that point. Therefore, the step of the protocol that finds the LCE to deduce the partitioned transactions, ignores log records spooled during resolution. To make these two kinds of entries distinguishable, log entries spooled during resolution use special opcodes. However, there is one exception to this rule. The log of a directory created during resolution is initialized with a record containing the opcode of the user-initiated `mkdir` transaction. This guarantees that a future resolution of this newly-created directory will correctly deduce its partitioned transactions.

#### 4.4.3.2 Performance Optimization: Connection Management

As mentioned earlier, the coordinator uses RPC to communicate with the subordinates. Each RPC request is made through a *connection* established via an `RPC_Bind` request. Recall from Figure 4.3, that the bind request is made during the first phase of the resolution protocol and can potentially be a high latency operation if the target of the request, i.e. a subordinate, is inaccessible. The delay is as long as an RPC timeout which is set to 15 seconds in our environment. To enhance performance, each server employs two mechanisms: first, it maintains a *cache* of connections with other servers; second, it maintains the state of each server by probing at regular intervals. The bind request is made only if a server is accessible. The former mechanism improves performance by avoiding the cost of a bind request every time the protocol is started. The latter mechanism avoids the potential delay if a server is inaccessible. Of course these optimizations come at their cost: extra space is needed to maintain the state of each server and the cache of connections; network resources are consumed to probe the servers periodically. However, the overhead is negligible in the Coda environment since the number of servers is assumed to be small.

#### 4.4.3.3 Availability Optimization: Twin Transactions

In some situations, as shown in Figure 4.13, the same transaction might be executed in multiple partitions. For example, the user might initiate a `rmfile(core)` transaction during a partition, then move to another partition, rediscover the `core` file and initiate another `rmfile(core)` transaction. Such transaction pairs are called *twin* transactions [25] because they perform identical operations. Their certification is guaranteed to declare a conflict since their read and write sets are identical. However, if the model treats twins as one transaction and requires each server to execute only one of them in the global serial history, then the conflict can be eliminated.

Coda recognizes three kinds of twin transactions: twin renames, twin links and twin deletes, i.e. `rmfile`, `rmdir` and `rmsymlink` transactions. Note that twin creates are impossible in the Coda model because an object cannot be created in multiple partitions since its `fid` depends on the server where it is created. However, the `link` transaction is an exception to this rule since it does not create a new `fid`.

The servers use a simple technique to recognize twin transactions. During resolution, if a transaction T cannot be certified, but the value of its write-set is already reflected in the system, then a twin transaction exists for T. In this case, T is ignored and the server continues processing the rest of the log. This approach does not guarantee finding all twin transactions: if the write-set of either twin transaction is modified by any transaction that appears later in the history, then the twin transactions will appear to be in conflict.

| Partition 1 | Partition 2 |
|---|---|
| **T1**: `rmfile (d, core, 2.2, u1)` | **T2**: `rmfile (d, core, 2.2, u2)` |

These histories are not resolvable because **T1** and **T2** conflict: their read and write sets are identical.

Figure 4.13: A History with Twin Transactions

| Partition 1 | Partition 2 |
|---|---|
| **T1**: `mkfile (d, core, 2.2, u1)` | **T3**: `mkfile (d, core, 4.5, u2)` |
| **T2**: `rmfile (d, core, 2.2, u1)` | |

These histories are not resolvable because **T1** and **T3** conflict: each creates a file named core in the same directory `d`. However, if the identity sequence **T1, T2** is not certified during resolution, then **T1, T2, T3** are resolvable.

Figure 4.14: A History that is Resolvable only if the Identity Sequence is Ignored

#### 4.4.3.4 Performance and Availability Optimization: Identity Sequences

File system activity often exhibits a *cancelling* behavior, i.e. a user initiates a transaction to undo the effect of a previously executed transaction. A common example is the creation of a temporary file followed by its deletion. Sometimes, the cancelling behavior is exhibited by a sequence of updates containing more than two transactions. As mentioned before in Section 4.3.2 (on page 56), such sequences are called *identity sequences*. An identity sequence consists of transactions whose final write values are identical to their initial read values.

Intuitively, records of transactions belonging to an identity subsequence could be removed from the resolution logs without changing the resolvability of the partitioned histories. Doing so reduces the work and improves the performance of the third phase of resolution. Furthermore, as shown in Figure 4.14, ignoring an identity sequence sometimes increases the number of histories that are accepted as resolvable. This increases the availability of the system. However, for correctness, records belonging to an identity sequence must be inserted in the resolution log of every replica: if the protocol proceeds without all VSG members, a tail of the identity sequence might need to be propagated when other members become available.

Recognizing an identity sequence has high space and time overheads due to the combinatoric explosion in comparing read/write sets of arbitrary transactions. The servers try to minimize this overhead by limiting the sequences they recognize to the simplest form, namely a `create-object(foo)` followed by a `delete-object(foo)`, and no transaction in between these two operations accesses the object `foo`. The object `foo` can be a file, directory or symbolic link. A create-delete pair is found by grouping the history of transactions according to the object they access and then ordering them by their commit time. Using this strategy strikes the right balance between resource cost and performance benefits: the simplest identity sequences are found easily and at the same time provide most of the availability benefits.

# Chapter 5

# Automatic File Resolution

The goal of file resolution is to make diverging replicas of a file identical. Like directory resolution, file resolution requires semantic knowledge of the data to perform its task. But, unlike directory resolution, file resolution cannot use a uniform system-wide policy because (a) the structure and semantics of a file's data can be unique and (b) the system has no knowledge of these semantics. Consequently, only the user or the application that modifies the file has enough knowledge to resolve it. Programs that incorporate application-specific knowledge to perform resolution are called *Application-Specific Resolvers* (ASR). Thus, the goal of automatic file resolution is to invoke an appropriate ASR transparently when needed.

A file's replicas often differ because a subset of them are *stale* due to a missed update rather than because of concurrent updates in multiple partitions. Usage experience with Coda has shown that the former reason accounts for 99% of all cases in which file replicas differ. This is because files are rarely shared for writing in the Unix environment and consequently they are not susceptible to concurrent updates in multiple partitions. Unlike a file with diverging replicas, a file with some stale replicas can be resolved without any semantic knowledge of its data - its *latest* replica can be distributed to all servers in its VSG. Since this situation occurs often in practice, Coda addresses it using a special technique, based on *version-vectors* [39]. This technique is a purely syntactic approach and is more efficient than a semantic approach like application-specific resolution. The version-vector approach has two goals: first, to determine if a file's replicas differ due to staleness and second, to recognize the replica with the latest data and propagate it to all VSG members.

The rest of this chapter is divided in three parts. It first describes the version-vector approach. Next, it focuses on the design and implementation of the interface used to specify and invoke ASRs. Finally, two example applications that use the ASR interface are described.

## 5.1 Syntactic Approach: Version-Vectors

For every replica of a file, each server maintains an array, called a *version-vector*. The number of elements in a version-vector is equal to the number of file replicas and each element counts the number of updates made at the corresponding replica.

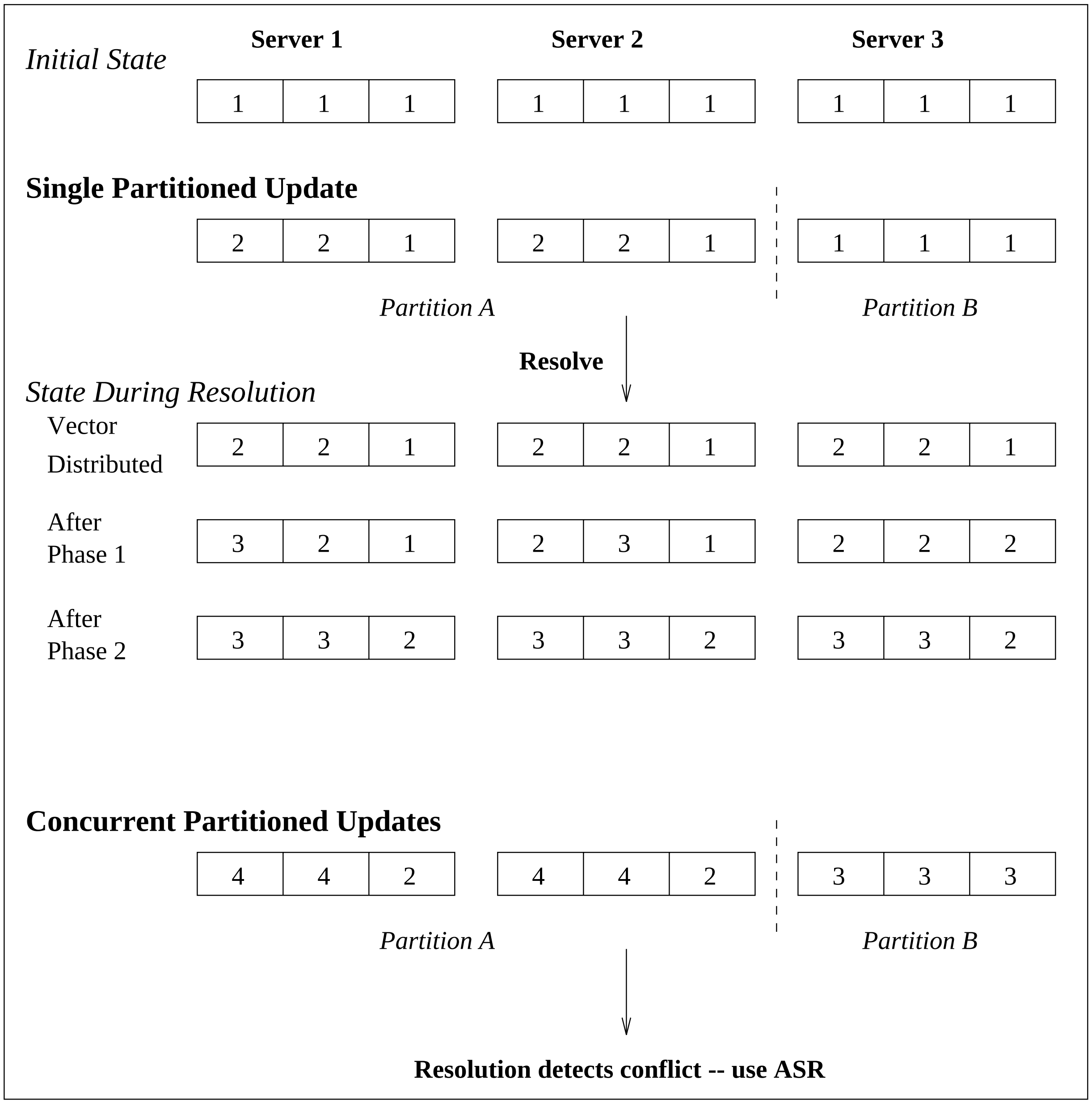
When a file is created, its version-vector elements are initialized with the value 1. For example, the initial version-vector state of a file `foo` replicated at three servers is `[1 1 1]`, and this vector is stored with all three replicas. The vector elements are incremented as the replicas are updated. Recall from Chapter 2 (Section 2.2.2 on page 11) that an update is propagated from the client to all accessible replicas in parallel. An update transaction is performed in two phases. In the first phase, Venus requests all servers in the AVSG to execute the transaction - each server certifies and performs the update. During file-update transactions, i.e. `store` and `setrights`, each server also increments the vector element that counts its own updates. During the second phase, Venus distributes the list of servers that completed the first phase successfully. Each server increments the vector elements corresponding to the replicas that were updated successfully in the previous phase. Consider an update for file `foo` mentioned above: after the first phase, the version-vector values of the three replicas are `[2 1 1]`, `[1 2 1]` and `[1 1 2]`; after the second phase, all three replicas have an identical version-vector, `[2 2 2]`.

Since Coda does not use the standard two-phase commit protocol [17, 28] to propagate updates from clients to servers, the update count in a version-vector is not always accurate. For example, if a server's reply message at the end of the first phase is lost, Venus assumes the operation failed at that server. As a result, none of the other servers increment the counter corresponding to the "failed" server in their vectors, even though the "failed" server itself increments its own counter. But this inaccuracy is harmless because it is always *conservative*. In other words, a server may fail to notice an update but it will never erroneously assume that a replica was updated. A server's estimate of updates to itself is, of course, always accurate.

In the absence of failures, the version-vectors of a file's replicas will be identical after each update. However, if a file is updated during a partition, then its replicas' vectors will be different. Figure 5.1 shows the state of the version-vectors of a file's replicas when it is updated during a partition. Venus invokes resolution for a file when it detects that the file's replicas have different version-vectors. As in directory resolution, Venus appoints one of the servers in the AVSG as the coordinator of the file resolution protocol. The coordinator collects the vectors of all accessible file replicas and compares them pair-wise. Comparing two vectors $V_1$ and $V_2$ yields one of the following results:

1. $V_1$ is identical to $V_2$, in which case the corresponding replicas are identical.

2. $V_1$ *dominates* $V_2$ (or vice-versa), i.e. every element of vector $V_1$ is greater than or equal



This figure shows the version-vectors of a file's replicas when they are modified during a partition and resolved thereafter. The file has three replicas. The figure shows two scenarios: first, only two of the three replicas are modified in a partition and the file is successfully resolved when the partition ends; second, the replicas are updated concurrently in two partitions and can be resolved only by an ASR.

Figure 5.1: Using Version-vectors for File Resolution

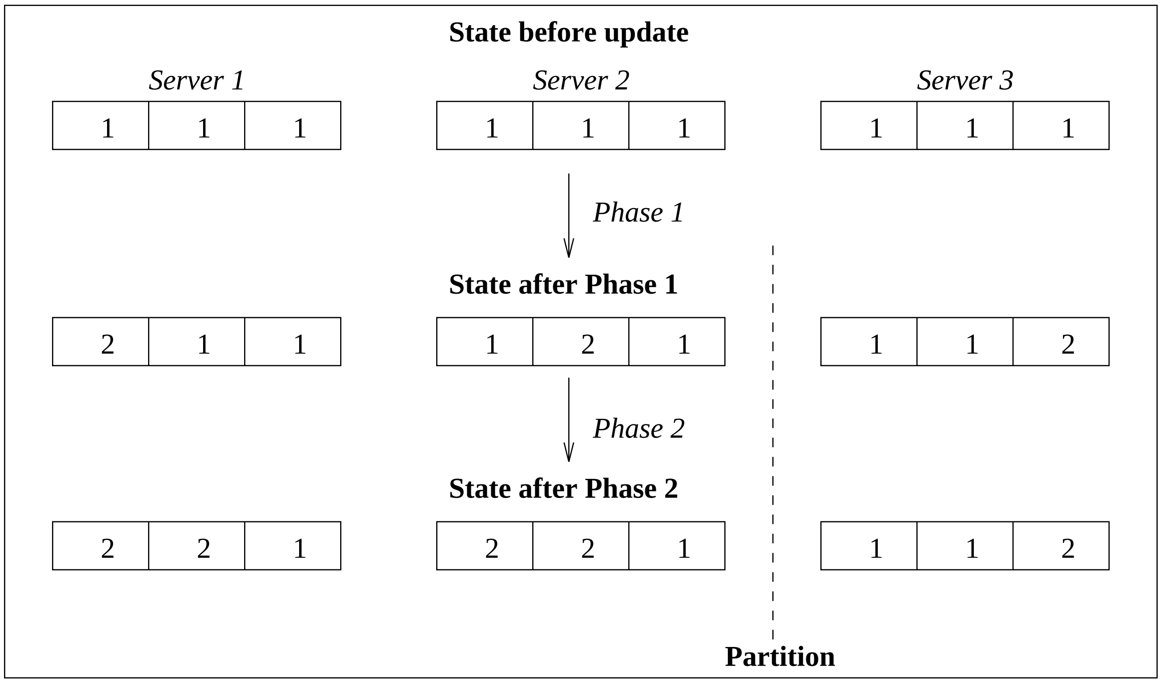
to the corresponding element of vector $V_2$. In this case, the replica with vector $V_1$ (or $V_2$, if $V_2$ dominates $V_1$) has the newer version of the file's data.

3. $V_1$ *conflicts* with $V_2$, i.e. at least one element of $V_1$ is greater than the corresponding element of $V_2$ and vice-versa. In this case, the two replicas are diverging.

If one replica's vector dominates the vectors of all other replicas, then that replica contains the latest version of the file's data. This replica's contents and version-vector are distributed to all servers in the AVSG by the coordinator using a two phase protocol. This protocol is similar to the one used by Venus to perform updates: during the first phase the coordinator collects the dominant replica's contents and version-vector and then distributes them to all the AVSG servers; the list of subordinate servers that completed phase one successfully is distributed during the second phase. The state of the version-vectors of a file during the two phases of resolution is shown in Figure 5.1. If the coordinator finds a pair of conflicting vectors, then an ASR is needed to perform the resolution. Details of the ASR mechanism will be provided in the next section.

The version-vector technique described above is space and time efficient. Its space requirements are modest - a file with $n$ replicas, requires extra space for storing $n^2$ integers (for $n$ vectors with $n$ integers). If $s$ is the number of servers in the file's AVSG, the time spent by the coordinator for collecting the vectors is $O(s.n)$ and the time needed to compare the vectors is $O(s^2.n)$ – there are $C_2^s$ (or $O(s^2)$) vector pairs and comparing each pair requires $n$ comparisons. Furthermore, the time needed to collect and distribute the dominant file replica with $b$ bytes is $O(s.b)$. Thus, the total time needed for resolution is $O(s^2.n + s.b)$. Since $s$ and $n$ are typically less than 4 in our environment, the dominant term for most files is $s.b$, the time needed to propagate a file with $b$ bytes to $s$ servers.

Since the vector elements are only a conservative estimate of the real update count, their comparison during resolution is also conservative. It never falsely declares equality or dominance but may declare a *false conflict* when none exists. Figure 5.2 shows an example of a false conflict. A common situation in which false conflicts are reported occurs when resolution is invoked between the two phases of the file update protocol. At this point of the update protocol, each server has incremented only its own update count and the vectors corresponding to any two replicas appear conflicting, even though the replicas themselves are identical. This situation, illustrated in Figure 5.2, occurs frequently if a file is write or read shared by multiple clients. In our environment, false conflicts occur more frequently due to a user accessing her files from multiple clients, rather than due to multiple users sharing a file. For example, users often update a program file on their desktop machine but use a different client as a compile engine. Due to a delay between the two phases of the update protocol, the recently stored file appears to have diverging replicas when accessed from the second client. To recognize this frequently occurring situation, each server augments the basic version-vector with the identity of the last transaction to update the file replica. This identifier is called the *storeid*. If the vectors of two replicas are conflicting, but their storeids are identical, then their contents are equal and the replicas are *weakly-equal*. If the resolution coordinator detects weak equality amongst a group of replicas, it makes their version-vectors identical.



This figure shows how lost messages result in false conflicts. The file being updated is replicated at 3 servers. The reply message from Server 3 during the first phase of the update protocol is lost due to a network partition. As a result, during the second phase, Server 1 and Server 2 do not increment the update count for Server 3. Even though the file's replicas are identical, the version-vector from Server 3 conflicts with vectors from both Server 1 and Server 2.

Figure 5.2: False Conflicts with Version-vectors

## 5.2 Semantic Approach: Application-Specific Resolution

As mentioned before, a semantic method for file resolution is needed when the file's replicas are diverging due to concurrent updates in multiple partitions. The updates comprising the multi-partition histories of the file's replicas are non-serializable and the main task of resolution is imposing a serial order on the updates in the combined history.

One method of "performing" this task is forcing the user to perform compensation at each diverging replica. While acceptable as a method of last resort, this method has two problems.

First, the task of resolution can be quite onerous to the user because he must understand the semantics of updates made by all other users who also updated the file. For example, consider a file `foo.c`, write-shared by two users. Before either user can decide the serial order of the updates, each user must understand the code fragment updated by the other user. The other problem with this approach is that restricting the resolution task to users limits the availability of the file – it is inaccessible until a user who understands the semantics of the file's data has performed the compensation.

In many important cases, an application may itself have enough knowledge to perform resolution. Consider an appointment calendar maintained in a file with two replicas. A user makes appointments and updates one replica during a network partition while a secretary adds appointments to the other replica. The two updates are not serializable when viewed as mutations at the file granularity. However, if the appointments do not overlap, a utility program, i.e. the calendar's resolver, could merge the updates. Electronic project diaries and check-books shared by users in the same organization are other examples of data that provide an increased opportunity for write-sharing. Fortunately, applications that manipulate such data have well-defined methods for performing updates and can be extended to perform application-specific resolution.

Coda's design for application-specific resolution concentrates on three issues: the methodology for invoking an ASR when needed, the interface for specifying a resolver for a file, and the execution of the ASR. It does not address the design of the ASR itself. The rest of this section focuses on these three issues.

### 5.2.1 ASR Mechanism: Design Considerations

*Transparency* is the most important requirement for the ASR invocation mechanism. Forcing the user to run an ASR manually should be avoided since doing so exposes him to the shortcomings of optimistic replication. One obvious strategy is to have the server coordinating the file resolution protocol invoke the ASR when it detects conflicting vectors for the file's replicas. However, this is undesirable in Coda for two reasons. First, security constraints make it inadvisable to run arbitrary ASR programs at the servers - recall from Chapter 2 that Coda servers are required to execute only trusted software. Second, scalability considerations suggest that the work be off-loaded to the client.

Instead, Coda relies on the client to invoke and execute an ASR. This strategy allows the reuse of some machinery already present at the client to implement ASR operations – performing pathname resolution to access replicas of the file being resolved and collecting them from the servers. As in directory resolution, Venus invokes the ASR *lazily*, i.e. only when a user accesses a file with diverging replicas. Transparency is maintained by suspending the user's request until the ASR completes its execution. The only noticeable effect seen by the user is a longer pause for servicing the file request. This pause could be eliminated by implementing an aggressive policy that executes an ASR for each file with diverging replicas, as soon as the partition heals. However, as in directory resolution, an aggressive policy would lead to recovery storms. Furthermore, our usage experience with Coda shows that an ASR is invoked rarely enough to obviate the need for such policies.

To assist Venus in finding the resolver corresponding to a file, Coda provides an interface for users to specify this relationship *a priori*. *Simplicity* and *flexibility* were the foremost concerns in the design of this interface. Simplicity is important to make the ASR mechanism usable; flexibility simplifies changing the ASR specification for debugging or maintenance purposes. To satisfy these conditions, resolvers for files in Coda are specified using rules similar to those used by the Unix `make` utility. Most Unix users are already familiar with such rules and should find them easy to use. This approach is flexible because the ASR specification for a file can be changed by simply modifying its rule. Obviously, this approach assumes the `make` syntax is acceptable to most users. An alternative strategy would extend the system call interface to allow application writers to specify the resolver for each file as it is created. This information would be maintained along with each file and be transparent to the user. But the resolver could be changed only via the system call interface and be less flexible than Coda's rule-based approach.

*Security* is one of the design considerations for the execution of the ASR because it uses client workstation resources. Since an ASR may need to access arbitrary files and directories, one obvious strategy is to ignore security constraints and allow it to access any object. However, doing so violates Coda's security guarantees since ASRs are user programs that cannot be trusted. Coda's strategy is to execute the ASR as a separate process on behalf of the user who made the ASR-triggering file request. All requests from this process are subject to the same access checks as any other request from that user. The advantage of this scheme is the scope of the damage is limited to those objects that are accessible to the user whose request triggered the ASR. Of course, the disadvantage is that it does not provide the highest availability since an ASR may fail because it has insufficient privileges to modify a critical file. The reduction in availability is acceptable since only the file with diverging replicas is inaccessible. This is contrary to Coda's philosophy for directory resolution where the price of a failed resolution is much higher and unacceptable since none of the descendants of the unresolvable directory are accessible.

*Fault-tolerance* of the ASR execution is another important design consideration. *Atomicity* and *isolation* of updates made by an ASR are the main guarantees needed for providing fault-tolerance. Atomicity is necessary to avoid partial ASR updates resulting from a client crash during its execution. Otherwise, the partial updates themselves need to be resolved after recovery. Isolation is important for two reasons: first, to ensure a file is not being resolved simultaneously by multiple ASRs, and second, to prevent applications from using incomplete results of an ASR. For example, consider a calendar that is managed in two files `A` and `B`, and both files have diverging replicas. The ASR resolves file `A` successfully and a calendar manager opens this file while the ASR is executing. Then, due to some condition in file `B`, the ASR decides to undo its changes to file `A`. The user, who has already seen the ASR's updates to file `A`, will get an inconsistent view if and when the calendar manager opens file `B`. Details of the fault-tolerance mechanisms for the execution of an ASR will be provided in the next section.

### 5.2.2 ASR Mechanism: Design and Implementation

Viewed from a high level, the ASR mechanism works as follows. On every cache miss Venus verifies that all replicas of the file being accessed are identical. If the replicas' version-vectors indicate that they diverge, Venus searches for an ASR for this file using the rules specified by the user. If an ASR is found, it is executed on the client workstation. The ASR's mutations are performed locally on the client's cache and written back to the server atomically after the ASR completes. The application that requested service for the file is blocked while the ASR is executing. This mechanism for running an ASR is loosely analogous to a *watchdog* as proposed by Bershad and Pinkerton[4]: it extends the semantics of the file system for specific files to be resolved transparently when needed. The rest of this section describes the details of the invocation and execution of the ASR.

#### 5.2.2.1 ASR Invocation

To execute an ASR on the client, Venus makes a request to a special process called the ASR-`starter`, as shown in Figure 5.3. As its name implies, the `starter` process is responsible for finding and executing an ASR for a file. No ASR can be executed unless a `starter` process is executing on the same host as Venus. This policy provides one method of disallowing ASRs from executing on workstations whose primary users are concerned about security.

The functionality of the `starter` process could be provided as a library within Venus. However, we chose not to do so for two reasons. First, Venus code is complex and its binary image is already very large, about 5 Megabytes. Second, the design and development of the `starter` was greatly simplified - its compile time was much smaller and it could be tested independent of Venus. The disadvantage of this approach is a higher latency for starting an ASR. But, this is not a serious bottleneck since an ASR is invoked rarely.

The Venus process consists of multiple threads of control, and a subset of these threads, called *workers*, service file requests from users. The request to start an ASR is made by a worker thread when it notices that the file it is servicing has diverging replicas. This request is made via an RPC, called `Invoke`ASR, to the `starter` process. If the request is successful, the process-id (`pid`) of the newly started ASR is returned to the worker and it blocks awaiting the result of the ASR's execution. As a result, the user process being serviced by the worker thread also remains suspended until the ASR completes. If an ASR is not started due to an error, resolution fails and a conflict indicator is returned to the user process.

The `Invoke`ASR RPC has two parameters: the pathname of the file needing resolution and the identity of the user requesting service for that file.

The pathname of the file with diverging replicas is used by the `starter` process to find an appropriate ASR. Recall that an ASR for a file is specified using rules. The resolution rules are stored in a special file named `ResolveFile`, a la `Makefile`. Rules contained in a `ResolveFile` apply to all files in the subtree rooted at the directory containing the `ResolveFile`, except where overridden by another `ResolveFile` lower in the tree. This scoping mechanism is similar to lexical scoping used in programming languages and its advantage is that resolution rules are automatically inherited as new objects are created. For example, a user may have only one set of resolution rules in a `ResolveFile` in his home directory, which applies to all his files. Thus, to find the resolution rule for a file, the `starter` searches for a `ResolveFile` in its ancestors, bottom-up. The ancestors are obtained from the pathname parameter in the RPC request. The search stops when it finds a `ResolveFile`, or it reaches the system root, i.e. `/coda`. To simplify sharing of rule files, the `ResolveFile` entry in a directory can be a symbolic link. For example, an application-writer could provide a `ResolveFile` for the application's files. All users of this application can share the `ResolveFile` by creating a symbolic link to it in their personal directory containing the application's data files.

The second parameter of the `Invoke`ASR request, i.e. the user's identity, is used by the `starter` to execute the ASR on the user's behalf. Therefore, the `starter` process must have sufficient privileges to do so.

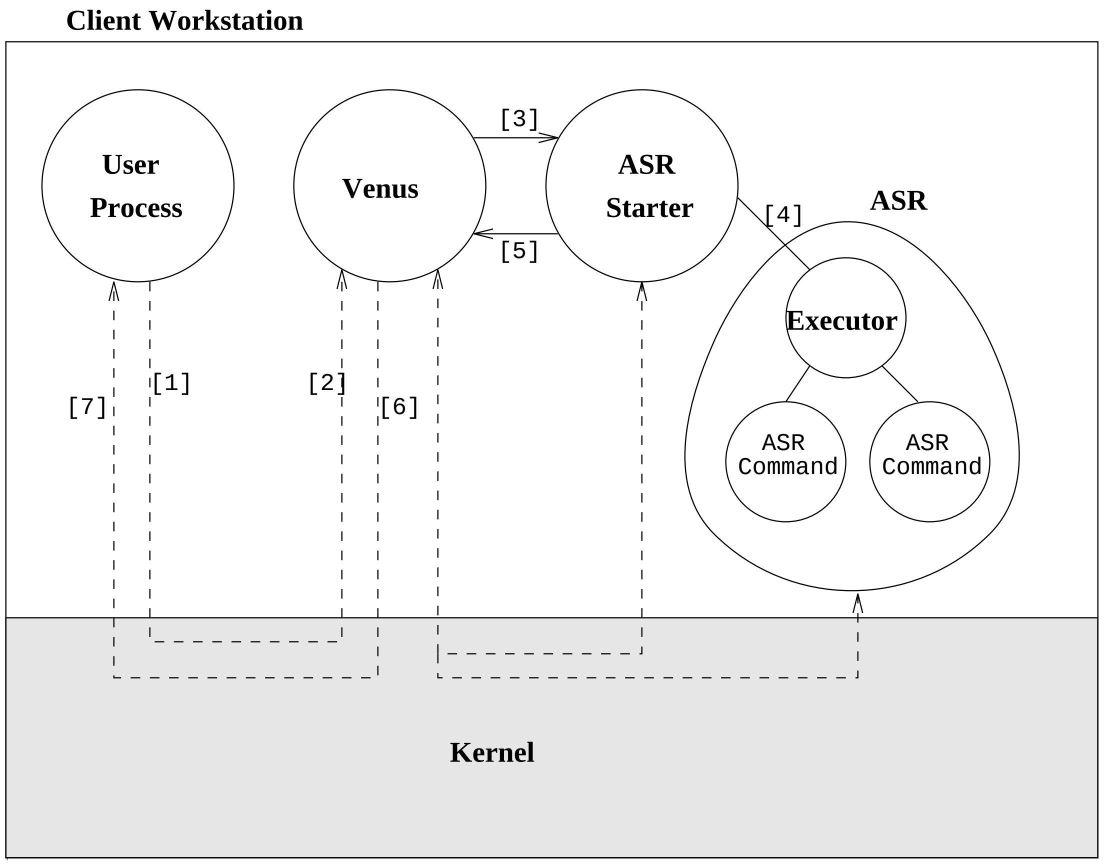
Once the ASR completes execution, its result is returned to Venus by the `starter` via an RPC called `Result_Of_ASR`. The `pid` of the recently completed ASR is also returned to Venus so that it can resume the worker waiting for this ASR. If the ASR succeeds, the worker retries servicing the user's request.

### 5.2.2.2 The ASR Rule Language

Like `Makefile`s, `ResolveFile`s contain multiple resolution rules separated by one or more blank lines. Each rule has the following format:

```
<object-list> : <dependency-list>
                <command-list>
```

The object-list is a non-empty set of object names for which this resolution rule applies. Object-names can contain C-shell wild-cards like "`*`" and "`?`", thus allowing one rule to be used for multiple files. For example, a rule that specifies `*.dvi` in its object-list applies to all files with a `.dvi` extension. Typically, such files are created by the LaTeX document-processing system



This figure shows the message exchange between processes involved in an ASR invocation on a client workstation. Seven events responsible for the ASR invocation and execution are labelled in the Figure: events [1] and [2] are the user request for a file that has diverging replicas; event [3] is the `Invoke`ASR RPC; event [4] is the execution of the ASR commands; the ASR result is returned to Venus via the `Result_Of_ASR` RPC labelled as event number [5]; events [6] and [7] return the result of the user request. The dotted arrows from the ASR and the `starter` show the message exchange to service their Coda requests.

Figure 5.3: ASR-Invocation

and require the same programs to be executed for their resolution. Allowing wild-cards in the object-list permits concise specification of the resolution rules.

The dependency-list contains a, possibly empty, list of object names whose replicas must be equal when the rule is utilized. This facility provides a useful check for resolvers that re-generate a file based on the contents of another *source* file. Consider once again the example of a file with a `.dvi` extension. Its contents can be re-generated by processing the source file with LaTeX. However, this strategy is usable only if the source file itself does not have diverging replicas, a condition that can be checked automatically by adding the name of the source file, i.e. the file with the `.tex` extension, to the dependency-list.

In principle it would be possible to recursively resolve objects in the dependency-list. However, to keep the implementation simple, an ASR invocation is aborted if any object in the dependency-list has diverging replicas. Otherwise, the system would need to detect cycles in the dependency-list. The disadvantage is an increased likelihood of conflicts due to a false declaration of resolution failure.

```
*.dvi:  $*.tex
        file-resolve $</$>
        latex $*.tex
```

For a file `/coda/usr/pkumar/foo.dvi,` the rule expands to:

```
        file-resolve /coda/usr/pkumar/foo.dvi
        latex foo.tex
```

And is executed only if the replicas of `foo.tex` are not diverging.

Figure 5.4: Macro-expansion of a Resolution Rule

The command-list consists of one command per line. Each command specifies a program to be executed along with its arguments. Since the object-list may contain wild-cards, some file-names that need to be passed as arguments to the resolver programs are not known until the rule is being utilized to invoke the ASR. Therefore, the rule-language provides macros that serve as place holders for these file-names. The language provides three `make`-like macros, `$*,` `$>` and `$<`. The `$*` macro expands to the string that matches the wild-card in the object-list; `$<` expands to the pathname of the parent of the file being resolved and `$>` expands to the file's name. These macros are expanded dynamically if and when a rule containing them is used to execute an ASR. Figure 5.4 shows the macro expansions using a simple example.

To find a rule that applies to a file `foo`, the `starter` process first parses the `ResolveFile`. The parser is implemented using `yacc`. Table 5.1 shows the grammar rules for the rule-language syntax. The string `foo` is matched against the names in the object-list of each parsed rule. The first rule containing a match is used as the resolution rule for `foo`. If no match is found or a syntax error is found in the `ResolveFile`, the `starter` assumes no ASR exists for `foo` and returns an appropriate error code to Venus. Before invoking the ASR the `starter` expands all macros in the dependency-list and the command-list of the rule and ensures none of the objects in the dependency-list have diverging replicas.

| | | |
|---|---|---|
| `Start` | $\rightarrow$ | **Blank-line** \| `Rule-list` |
| `Rule-list` | $\rightarrow$ | `Rule-list Rule` \| `Rule` |
| `Rule` | $\rightarrow$ | `Object-list` **:** `Dep-list` **New-line** `Cmd-list` |
| `Object-list` | $\rightarrow$ | `Object-list` **, Object-name** \| **Object-name** |
| `Dep-list` | $\rightarrow$ | `Dep-list` **Dependency-name** \| $\phi$ |
| `Cmd-list` | $\rightarrow$ | `Cmd-list Cmd Cmd-Terminator` |
| `Cmd-list` | $\rightarrow$ | `Cmd Cmd-Terminator` |
| `Cmd` | $\rightarrow$ | **Cmd-Name** `Arg-list` |
| `Cmd-Terminator` | $\rightarrow$ | **New-line** \| **;** |
| `Arg-list` | $\rightarrow$ | $\phi$ |
| `Arg-list` | $\rightarrow$ | `Arg-list Arg` |
| `Arg` | $\rightarrow$ | **Arg-Name** `Rep-specifier` \| **Replica-count** |
| `Rep-specifier` | $\rightarrow$ | $\phi$ \| **[** `Index` **]** |
| `Index` | $\rightarrow$ | **Integer** \| ***** |

Terminals of the grammar are shown in boldface.

Table 5.1: Grammar for the Resolution Rule-language Syntax

#### 5.2.2.3 Exposing Replicas to the ASR

Recall from Chapter 2 that Venus makes replication transparent to applications. The ASR, on the other hand, requires replication to be exposed since it needs to examine the individual replicas of the file being resolved.

One obvious strategy for exposing replication is making each replica accessible in a non-replicated region of the file system. Since the granularity for replication in Coda is a volume, this strategy would make each volume replica a separate entity in the non-replicated name space. For example, consider a volume named `u.pkumar` that is replicated at two servers and its root directory is normally accessible as `/coda/usr/pkumar`. The individual replicas of this volume could be `mount`ed at `/coda/nonrep/pkumar.rep0` and `/coda/nonrep/pkumar.rep1`. Using this strategy, the replicas of the file `/coda/usr/pkumar/foo` would be accessible as `/coda/nonrep/pkumar.rep0/foo` and `/coda/nonrep/pkumar.rep1/foo`. This approach is simple and needs no special implementation at the client. The disadvantage is the pathnames of the file's replicas are significantly different from the file's replicated pathname. This difference can be problematic if the ASR's execution depends on the replicated pathname of the file, for example, a resolver that uses `make` whose rule file contains absolute pathnames.

To overcome this shortcoming, Coda uses an alternative strategy. Before invoking an ASR, Venus explodes the file in-place into a *fake directory*. The directory is "fake" because it exists temporarily at the client that is going to execute an ASR but not at the server or other clients. Each replica of the file appears as a unique child of this fake directory and is named using the name of the server it is stored at. In the example above, replicas of file `foo` would be accessible as `/coda/usr/pkumar/foo/server1` and `/coda/usr/pkumar/foo/-server2` where `server1` and `server2` are the names of the servers that store the volume `u.pkumar`. Other files in this volume that have identical replicas still appear non-replicated to the ASR.

File replicas in a fake directory are accessible for reading via the normal UFS interface. However, the only mutating operation allowed on the fake directory is the `repair pioctl`, that takes the name of a "new" file as input. The "new" file can exist in the local UFS at the client or it can be one of the replicas of the file being resolved. Its contents are used to atomically set the contents of the diverging replicas to a common value. Once this operation succeeds, Venus collapses the fake directory back into a file.

For the duration of the ASR's execution, the volume containing the file being resolved is forced into a special *fakeify* mode. In this mode, any file in that volume with diverging replicas, is converted to a fake directory dynamically, i.e. if and when it is accessed. This functionality is necessary for an ASR that simultaneously resolves a group of files with diverging replicas. For example, a user's calendar might be maintained in multiple files, and its ASR needs to examine the replicas of all files to perform resolution. Of course, this approach assumes that all files mutated by the application are contained in the same volume.

To convert a file into a fake directory, Venus assigns it a special `fid` that is not assignable by any server. This ensures that no operation on this directory will erroneously force Venus to contact the server. The children of this directory are created as special symbolic links: the links are named using the names of the servers in the file's VSG; the content of each link is the non-replicated `fid` of the corresponding file replica. The symbolic link is special because it has the format of a mount point. Since Venus already contains special code to handle mount points, modifying it to access/fetch each file replica when the corresponding symlink is accessed was simple: all changes were localized to a small segment of the code.

Since the names of the diverging replicas are not known *a priori*, the rule-language syntax provides three additional macros, `[`$i$`]`, `[*]` and `$#`, that serve as *replica specifiers*. `[`$i$`]` is substituted by the name of the $i$th replica and `[*]` is substituted by names of all replicas separated by white space. The `$#` macro is substituted by the replication factor for the file being resolved and is useful for resolvers that need this number to correctly parse the argument list. Figure 5.5 shows the use of these macros with a simple example.

```
*.cb:
      merge-cal-replicas $< $# $>[*]
```

expands to

```
      merge-cal-replicas /coda/usr/pkumar 2 \
         mycal.cb/server1 mycal.cb/server2
```

This figure shows the macros expansion for a file whose replicated pathname is `/coda/usr/-pkumar/mycal.cb`. The file's volume is replicated at two servers, `server1` and `server2`.

Figure 5.5: Macro-expansion of a Resolution Rule with Replica Specifiers

#### 5.2.2.4 Execution of the ASR

The ASR is executed after all macros in the resolution rule have been expanded. Multiple programs may be executed in a single ASR invocation, since a rule's command-list can contain more than one command. Since these programs may take an arbitrary long time to execute, they are executed by a separate process, called the `command-executer`, that is `fork`ed by the `starter` process. The `starter` process, meanwhile, returns the `pid` of the `executor` to Venus. Executing the ASR from the `executor` allows the `starter` process to continue servicing other requests from Venus.[1] Once the ASR completes, its result is returned via the `Result_Of_ASR` RPC.

The `executor` runs with the identity of the user whose request triggered resolution. It executes the commands in the command-list sequentially, each in a separate process. If any command fails, the `executor` is aborted and an error returned to Venus.

Since Venus runs on multiple hardware platforms, the `executor` must have the ability to choose the binary appropriate for the client machine. To achieve this functionality, it uses the `@sys` pathname expansion capability of the Coda kernel. The kernel evaluates the `@sys` component, if present in a pathname, to a unique value on each architecture. This mechanism allows a single resolution command pathname to be used for any client machine. For example, the pathname `/usr/coda/resolvers/@sys/bin/mergecalendar`, evaluates to `/usr/coda/resolvers/pmax_mach/bin/mergecalendar` on the DECstation-5000 clients and to `/usr/coda/resolvers/i386_mach/bin/mergecalendar` on the Intel-386 clients.

[1]The `starter` is run as a part of the `advice-monitor` [11] that is used by Venus to obtain user feedback for caching strategies etc.

The rest of this section discusses security, robustness and fault-tolerance of the ASR execution.

**Security issues** Recall that an ASR is executed on behalf of the user who triggers resolution. This strategy strikes the right balance between availability and security. It does not provide the highest availability because an ASR can perform its task only if the user who triggered resolution has sufficient rights to modify the file being resolved. As mentioned in Section 5.2.1, the alternative strategy in which an ASR operates without any security checks would provide higher availability. However, this strategy would be insecure as Coda clients cannot be trusted. Executing an ASR on behalf of a special ASR-id is also unusable: the password of the ASR-id would have to be shared by all users who need to execute an ASR. Furthermore, the availability provided by this strategy would be limited only to those files that are writable by the ASR-id.

An advantage of executing an ASR on the behalf of the user is that the damage caused by a malicious ASR is limited only to those objects that are accessible to the user. However, the disadvantage is it exposes the user's personal data to the ASR and may be problematic if the ASR is malicious. For example, a user's private files are accessible to an ASR while it is executing. This problem is particularly serious because a user is unaware of it since an ASR is executed transparently. To safeguard against this problem, the user may disallow the execution of an ASR selectively, using several methods. First, shutting down the `starter` process prevents any ASR from executing. Second, the `cfs pioctl` interface to Venus allows the user to selectively turn off/on ASR execution for all files belonging to a volume. Finally, the user may specify a list of directories that is used to search for an ASR. An ASR is allowed to execute only if it belongs to one of the directories in this list. This strategy assumes these directories are protected and the user trusts all programs belonging to them. Of course, preventing the execution of an ASR using these methods results in loss of transparency in some cases.

**Robustness** A *mis-behaved* ASR may decrease the robustness of a client. For example, an ASR whose return code indicates success even though it is unable to resolve the file will cause the Venus worker thread to loop indefinitely: the worker, when resumed, will re-invoke the mis-behaved ASR since the file's replicas are still diverging. Resolvers with programming errors like infinite loops can end up starving user processes of critical resources. Recall, that a worker thread servicing the resolution-triggering user request is blocked until the ASR completes. Since the number of worker threads is finite, multiple executions of a mis-behaved ASR could block all worker threads and deny service to other users. Coda addresses these problems by limiting the execution time of an ASR to two minutes and the frequency of ASR invocations for an object to once every five minutes. In our limited experience, these default limits have been sufficient for most ASRs. The system allows these limits to be changed dynamically for specific objects via the `pioctl` interface for Venus. Of course, no statically set limit can be perfect: there will always be situations where a perfectly good ASR execution is aborted due to one of these limits being exceeded. In the worst case, some transparency will be lost since the user will be required to resolve the file manually.

**Isolation** To avoid multiple ASRs from resolving a file simultaneously, the `starter` ensures only one ASR is executing on a client workstation. A worker thread, requesting the execution of an ASR, is blocked if an ASR is already executing on the client. To avoid deadlock, all locks held by the worker are released before it is blocked. To maintain transparency of resolution, the worker requests the ASR execution again when it is resumed after the currently executing ASR completes. Of course, the worker does not wait, but returns with an error, if the request triggering an ASR is made by an executing ASR. For example, consider an ASR for file A, that accesses another file B. If file B also has diverging replicas then simply blocking the ASR request for B will result in a deadlock since the ASR for A will wait for itself indefinitely.

The above strategy only ensures local serialization of ASRs. It does not prevent multiple clients from running an ASR for the same file simultaneously. To detect this condition, Coda uses an optimistic strategy similar to that used for concurrency control of normal updates. Each ASR performs its updates in the client's cache assuming no other client is executing an ASR for the same file. However, when committing the updates made by an ASR, each server verifies that none of the objects in the ASR's write-set have changed since the ASR started executing. If this isn't the case, then none of the mutations made by the ASR are committed. Therefore, if multiple ASRs are resolving a file from different client workstations simultaneously, only the first ASR to finish succeeds.

Mutations performed by an ASR in the client's cache are undone if it doesn't complete successfully. To prevent other applications from accessing partial results of an ASR execution, Venus implicitly locks the volume, containing the file being resolved, exclusively for the ASR before invoking it. The volume remains locked while the ASR is executing and is unlocked implicitly when the `starter` makes the `Result_Of_ASR` RPC. This strategy assumes that an ASR mutates objects only in the volume containing the file being resolved. While an ASR is executing, any user request for an object in that volume is blocked. Recall that an ASR itself is executing on behalf of the user. Therefore, to distinguish between ASR and non-ASR requests, Venus uses the process-group mechanism of Unix. All the commands in the resolution rule are executed using the same process group number. This group number is returned to Venus by the `InvokeASR` RPC. Furthermore, the Coda kernel is modified to include in each request for Venus, not only the identity of the user on whose behalf the request is being made, but also the identity and group number of the process making the request. Any request for an object in the volume being resolved is blocked unless it originates from a process having the ASR's process group number. This strategy assumes the ASR processes do not deliberately change their group number while executing.

**Atomicity** An ASR execution may abort due to a variety of causes such as client failure, coding bugs in the ASR and exceeding ASR time limits imposed by the `starter`. Coping with the partial results of an aborted ASR execution can be messy, especially if they lead to further conflicts. To alleviate this problem, Coda ensures that the updates performed by an ASR are made visible atomically. This transactional guarantee can be provided by Venus because it knows the boundaries of an ASR computation (bracketed by `Invoke`ASR and `Result_Of_ASR` RPC requests) and because it can distinguish requests made by the ASR (using the process group mechanism described above).

Venus exploits the mechanism for *disconnected-operation* [25] to make ASR execution atomic. The basic idea behind disconnected-operation is to log all mutations made at a client while it is disconnected from a server. Then, this log is used to *reintegrate* the updates with the server when connection is re-established. The updates are committed at the server atomically, using the transaction mechanisms described in the previous chapter. To make the updates of an ASR atomic, its updates are not written through from the client's cache to a server but logged as in disconnected-operation. If the ASR aborts, its updates are undone by purging the log and the modified state in the client's cache. However, if the ASR completes successfully, its updates are reintegrated with the servers atomically.

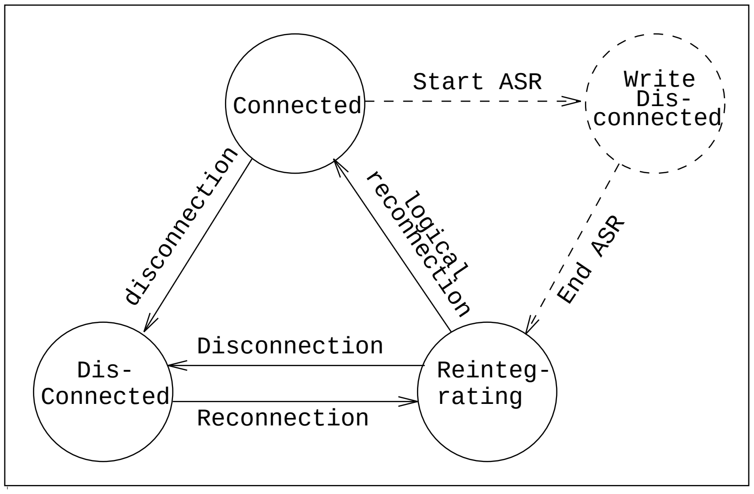
The implementation of disconnected-operation maintains a state variable for each volume. A volume may be in one of three states: connected (a.k.a. hoarding), disconnected (a.k.a. emulation) or reintegrating. In the connected state, all read requests for objects in the volume are serviced using the client's cache or the server if a cache-miss occurs, and all updates are written through the client's cache to the server.[2] In the disconnected state, all read requests in the volume are serviced using data from the cache and an error is reported if a cache-miss occurs. Updates are reflected on the local cache and recorded in a per-volume *modify-log*, but not propagated to the server. The reintegration state is transitory and exists only while a volume's modifications during disconnection are being propagated to the server. Details of the implementation of these three states are provided in Kistler's thesis [25].

The mechanisms for disconnected operation cannot be used directly for providing atomicity of updates made by an ASR since it may need to access objects that are not in the client's cache. Thus, to provide atomicity of updates made by an ASR, Venus maintains a fourth state for each volume, the *write-disconnected* state.[3] This state is a hybrid between the connected and disconnected states: all reads are serviced using the local cache or the server, but writes are not propagated to the server even if it is accessible. Two operations, *purge-log* and *commit-log* are provided to purge the mutations in the modify-log or to reintegrate them with the server, respectively. The volume containing the file being resolved is forced into the write-disconnected state before an ASR is invoked by Venus. The modified state transition diagram for a volume is

[2]File updates are written through to the server only when the file is closed.

[3]The dual of this state is read-disconnected. Both read- and write-disconnected states are collectively referred to as pseudo-disconnected states.

shown in figure 5.6. The volume remains in this state while the ASR is executing. Therefore, none of its changes are propagated to the server. When the result of the ASR execution is returned to Venus, it commits or aborts the ASR's mutations depending on whether the ASR succeeded or failed. Commit-log is invoked if the ASR computation was successful and the server is accessible: the volume first transitions into the reintegration state, when the ASR's updates are propagated to the server, and then into the connected state. Purge-log is used to undo the changes made by the ASR if its computation failed.



This figure shows the modified state transition diagram for a volume with the new *write-disconnected* state. The new state and its related transitions are shown in dotted lines/arrows.

Figure 5.6: Volume State Transitions

The write-disconnected state for a volume was implemented with small modifications to the implementation of the disconnected and connected states. Servicing a read in this state is similar to servicing a read in the disconnected state if the object being accessed is cached, and similar to servicing a read in the connected state if the object isn't cached. Servicing an update in this state and the disconnected state is similar except for one small difference - mutation of files with diverging replicas. Since such files are not cache-able, they cannot be modified in the disconnected state. However, the write-disconnected state must allow these files to be mutated since that is the motivation for executing the ASR. As described in Section 5.2.2.3, such files can only be modified via the `pioctl` interface. To support the `repair` operation in the write-disconnected state, a new record type is defined for the client modify-log. Furthermore, the reintegration routines at the server, that receive and process the modify-log, are also modified to recognize this record type and to perform the `repair` operation. The system provides a special wrapper program called `file-repair` for the `repair pioctl`. This program takes as input the name of the file with diverging replicas and the name of the file whose contents should replace the diverging replicas. If the latter parameter is missing, then all the diverging replicas of the file are truncated.

## 5.3 Example Application-Specific Resolvers

This section shows the use of the resolution rule language using two example ASRs used in the CMU/Coda environment: a resolver for a calendar application `cboard`, and a resolver for a file created by the `make` utility. The ASRs use different strategies to perform their task. The ASR for the `cboard` application resolves its files by merging their diverging replicas, while the resolver for the `make` application reproduces a file's data from its source files. Unlike the `cboard` resolver, whose actions are dependent on the contents of the diverging replicas, the actions of the latter ASR are independent of the contents of the file replicas being resolved.

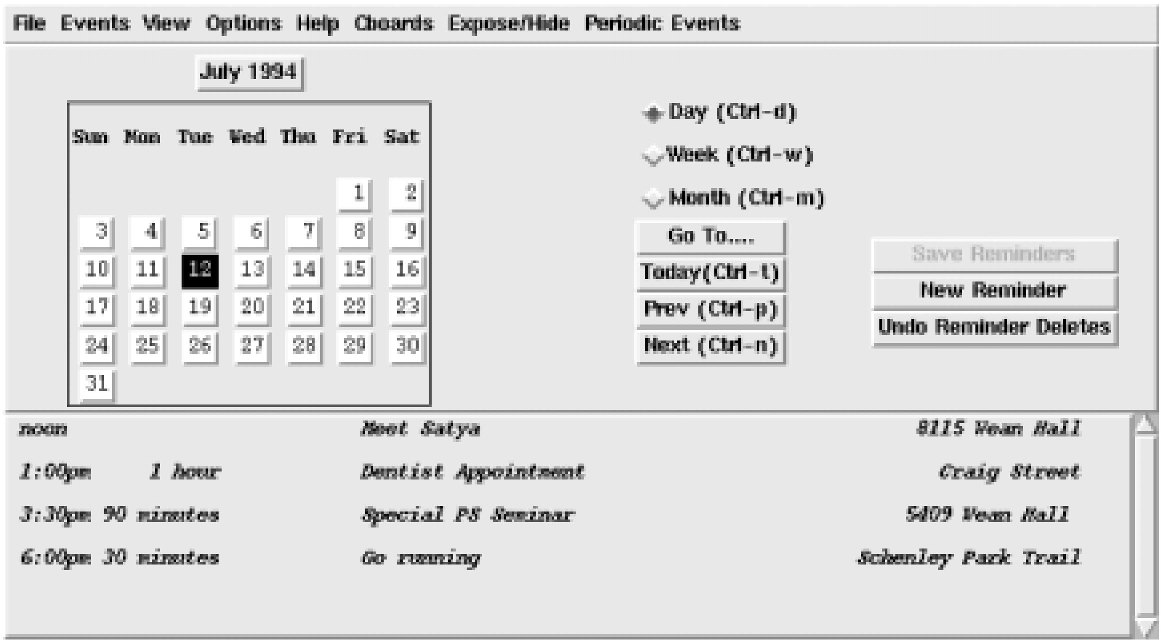
Before closing, this section also shows how the rule-language can be used to automatically invoke a file repair-tool when needed. The repair-tool uses feedback from the user to perform resolution.

The purpose of using these examples is to show different kinds of ASRs implemented in the Coda environment. It is not, by any means, an exhaustive list or an argument for completeness of the resolution rule-language.

### 5.3.1 Calendar Application

The `cboard` application maintains a calendar of events in a database. Typically, each user maintains her appointments in one or more databases. For example, a user may have a `personal` database, for making private appointments, and an `official` database that she shares with her secretary, for recording business appointments. In the CMU environment, the `cboard` application maintains an additional database, the `system` database, that contains announcements for public presentations in the university. This database is accessed in read-only mode by most users, except for the administrative staff responsible for posting announcements to this database. The `cboard` application allows events to be copied from the `system` database to a personal database, so that users can set up reminders for specific events. In general, the `system` database and a user's `official` database, exhibit some write-sharing and are thus prone to concurrent updates in multiple partitions.

Each database is maintained in two separate files, an *events* file and a *key* file. The former file is stored in ASCII format containing one record for each event. The key file, on the other hand, is stored in binary format and contains a hash-table index of the events file. The hash-table buckets are sorted based on the time of an event, so that the events for a particular date can be obtained efficiently. These two files are named uniquely using the name of the database concatenated



```
noon                      Meet Satya                 8115 Wean Hall
1:00pm     1 hour         Dentist Appointment        Craig Street
3:30pm 90 minutes         Special PS Seminar         5409 Wean Hall
6:00pm 30 minutes         Go running                 Schenley Park Trail
```

This figure shows the Tcl/Tk based interface for browsing events in the calendar. The menu is used to invoke functions like selecting a database, inserting and deleting events etc. The top frame shows the calendar and highlights those days whose events are being viewed in the lower frame. The user may view events for a day, week or month. The lower frame shows a one line summary for each event. Details for an event can be seen by clicking on its summary line. The application uses a dialog-box to remind users about an event and a window-based form to receive user input.

Figure 5.7: User Interface for the Calendar Manager

with special extensions. The events file uses a `.cb` extension and the key file uses a `.key` extension. The files storing the `system` database, for example, are named `system.cb` and `system.key`.

The `cboard` application provides tools for users to browse through a calendar database or mutate it. The user interface, implemented in Tcl/Tk [36] on the X11 windowing system from MIT, is shown in Figure 5.7. The application allows three kinds of mutations[4] on a database: *inserting* new events, *removing* cancelled events and *updating* modified events. A new event is inserted at the end of the events file of a database. To remove an event from the database, its record in the events file is invalidated, i.e. removed logically from the calendar, but not deleted physically from the file. Finally, updating an appointment is equivalent to invalidating its record and inserting a new one with the updated data. In all three cases, the hash-table's

[4]The `cboard` application provides other methods of updating a database. However, these methods are either not relevant to our discussion or are equivalent to one of the three methods described here.

buckets corresponding to the changed records are modified and both the key file as well as the events file are written to disk.

Since each calendar mutation modifies the events file and the key file, a concurrent partitioned update to the calendar causes both these files to have diverging replicas. Therefore, a resolution rule for the `cboard` application contains both these files in its object-list. The resolution rule for this application is shown in Figure 5.8. The dependency-list of the rule is empty, since the calendar's resolver does not require any other file to have non-diverging replicas. The resolver is executed in three steps. First, the `merge-cal` program merges the diverging replicas of the events file. The number of replicas and their names are passed in the argument list of this program. The merged events file and its corresponding key file are first written to a new database since the files with diverging replicas can be mutated only via the `repair pioctl`. The name of the new database is also specified in the argument list. In the rule above this new database is stored in `/tmp/newdb.cb` and `/tmp/newdb.key`. In the next two steps, the new database is used to atomically update the contents of the diverging files using the `file-repair` command provided by the system.

The `merge-cal` program, provided by the `cboard` application-writer, performs its task as follows. It parses the replicas of the events file and builds a hash-table index for each replica. A deleted event, that appears as an invalid record in the events file, is included in the index so that the resolver can disambiguate between recent deletes and new insertions. Once the indices are built, they are merged using a straightforward algorithm. First, a unique copy of each valid record is preserved; next, any record that has been invalidated in any replica is invalidated in the merged index. This algorithm guarantees that all insertions and deletions are preserved. Recall, that an update event appears as a deletion followed by a new insertion. Therefore, in case an event is updated simultaneously in separate partitions, the merged index will contain multiple records for this event, one for each partitioned update. The update mechanism could be easily changed so that the merge algorithm can detect concurrent partitioned updates and report a conflict. However, we chose not to do so for two reasons. First, changing the update algorithm required additions to the record structure that would prevent the `cboard` application from being backward compatible, an important requirement for a calendar manager. Second, in our experience with `cboard`, such updates occurs so rarely that this shortcoming has not caused any problem.

The `cboard` application was designed and implemented at CMU more than a decade ago. We chose this as one of our test applications due to its wide-spread use. There are approximately 90 users in the CMU SCS community using this application on a daily basis. The user-interface for the `cboard` application, when it was originally implemented, was based on ASCII terminal input/output. The more recent implementation, based on windows and Tcl/Tk and shown in Figure 5.7, is not as widely used as the first implementation. Currently there are 10 users using this version of `cboard` on a daily basis. We hope it will be used more widely as it becomes stable.

```
*.key, *.cb:
      merge-cal-replicas -n $# -f $</$*.cb[*] -db /tmp/newdb
      file-repair $*.cb /tmp/newdb.cb
      file-repair $*.key /tmp/newdb.key
```

Figure 5.8: Resolution Rule for the Calendar Application

From the above discussion it is clear that an ASR can be implemented for applications with structured data files. Examples of such applications include news-reading applications like `rn`, `gnus`, `messages` (Andrew) etc. and electronic-mail reading programs like `mh` and `mail` etc. Consider the `.newsrc` file maintained by news reading applications. This file contains a user's subscriptions and a list of article numbers he has read. This file is typically not shared by multiple users and is thus not prone to concurrent partitioned updates. However, it may be "shared" by a user with himself while reading news articles from two different client workstations. When this occurs, the file's resolver could merge the diverging replicas of the `.newsrc` file by computing the union of all articles read by the user.

### 5.3.2 Make Application: Reproducing a File's Data

Another ASR commonly used in the Coda environment is for files produced by `make`-like applications. A characteristic common to such files is their contents are produced by processing another *source* file with some application program. If such a file has diverging replicas, its contents can be reproduced by processing the source file again with the application program; assuming of course, that the source file itself does not have diverging replicas.

Files whose names have a `.dvi` extension or a `.o` extension are common examples of files produced by application programs without user intervention. The former are created by LaTeX and the latter are produced by the C compiler. The resolution rules for these two file types are shown in Figure 5.9.

Note that both rules have a non-empty dependency-list, which is typical for applications that regenerate the file's contents. To resolve diverging replicas of a file named `foo.dvi`, its ASR reproduces its contents by processing `foo.tex` with LaTeX, provided `foo.tex` itself does not have diverging replicas. Recall, that all mutating operations, except `repair`, are disallowed for a file with diverging replicas. Since LaTeX needs to modify `foo.dvi`, the file must either be repaired before LaTeX is executed, or LaTeX must be modified to issue the appropriate `pioctl`. We chose the former approach since it requires no modifications to LaTeX. Executing `file-repair foo.dvi` results in empty replicas for the file `foo.dvi`

```
*.dvi:  *.tex
      file-repair $</$>
      latex $*.tex

*.o:  *.c
      cc -c $*.c -o /tmp/$*..o
      file-repair $> /tmp/$*..o
```

Figure 5.9: Resolution Rules for the Make Application

and allows LaTeX to be executed without problems.

A similar methodology can be used to resolve a file named `bar.o` provided the source file `bar.c` does not have diverging replicas. Its contents can be reproduced by processing `bar.c` with the C compiler. Since the C compiler provides a mechanism for redirecting the compiled code to a file named differently from the source, the contents of `bar.o` can be first produced in a temporary file (`/tmp/bar..o`). Then this file is used as input to the `file-resolve` program to atomically set contents of all replicas of `bar.o`.

The two examples above show that a file, produced by a program execution, can be easily resolved transparently. The limitation of such ASRs is they fail if the source file has diverging replicas. Our experience with ASRs isn't sufficient to predict the frequency of this failure.

### 5.3.3 Automatically Invoking the Repair-tool

For situations in which an ASR execution fails, Coda provides a repair-tool that uses feedback from the user to resolve the diverging replicas of a file. This tool could also be invoked as an "ASR" for a file that does not have its own ASR. To achieve this functionality, the following rule could be added to the `ResolveFile`:

```
*:
    xfrepair $</$>
```

Since the name "`*`" in the object-list matches the name of any file, this rule must be specified at the end of the `ResolveFile`. Any rule that appears after this rule will never be used. `xfrepair` is the name of the program binary corresponding to the repair-tool. The tool takes as input the name of the file with diverging replicas, specified by the `$</$>` macro. The advantage of using this rule is the tool is invoked automatically when the divergence is detected, and the user can resolve the divergence immediately. Details of the design and implementation of the tool are deferred until the next chapter.

# Chapter 6

# Manual Resolution

The responsibility of performing resolution falls on the user when the automated facilities, described in the previous two chapters, discover conflicting transactions in the partitioned histories of an object. For example, two users may have created files with the same name in partitioned replicas of a directory. Or, a user and her secretary may have inserted different appointments for the same hour in partitioned replicas of a calendar file. In both cases, the partitioned transactions were correct when they were performed in isolation, but violate an invariant when they are merged. The directory updates violate the invariant requiring a directory to contain unique file names; the file updates violate the invariant requiring a calendar to never contain multiple appointments for the same hour. In either case, the user intervention is needed to merge the conflicting updates.

The term *repair* is used to refer to the manual resolution of an object. The system provides assistance to the user for repairing an object in two ways. First, it provides means by which a user can examine the state of each replica of the object being repaired. Second, it provides a *repair tool* that helps a user in finding the differences between replicas of an object and performing compensation at each replica.

This chapter describes the repair facility for the Coda file system in two parts. The design rationale for the facility is described first, followed by the details of its design and implementation.

## 6.1 Design Rationale

The repair facility is required to *preserve* the diverging replica states as evidence for the user and *notify* him about the conflict. This is achieved by marking the diverging replicas of the object with a special *inconsistency* flag. An inconsistent object cannot be accessed by normal applications using the Unix API. Thus, the damage is prevented from spreading and the state of the replicas is preserved. The user is “notified” about the conflict *lazily*, i.e. the conflict is discovered by a user only when he tries to access the object. Of course, a disadvantage of the lazy scheme is that a conflict may remain unnoticed for a long time.

An alternative strategy for preserving replica state, is moving the replicas to a special *conflict* region of the file system, (*a la* `lost+found` directory of the Unix file system). The Ficus system [20] uses this strategy and notifies the owner of the moved object via electronic-mail. This strategy has two disadvantages. First, it is inconvenient because the object disappears temporarily from the name space and remains unavailable until the owner repairs it. Other users who share this object are forced to wait for the owner even though they are capable of performing the repair. Second, moving the object causes some evidence for the conflict to be lost: the conflicting updates may have been made as part of an encapsulating computation that modifies other objects, but these objects are not moved to the conflict region if they don’t have any conflicting updates. Coda’s strategy makes the context for the conflict available to the user since the inconsistent object appears in the directory in which it existed at the time of the conflicting updates.

Although replication is normally transparent to the user, it must be exposed during the repair so that the user can examine the individual replicas of the object he is repairing. To expose the replicas of an object, the repair facility borrows the technique used for application-specific resolution, i.e. the inconsistent object is exploded in-place into a fake directory with each of its replicas appearing as a child of this directory. The in-place explosion of the inconsistent object is performed by Venus only when the user indicates that he is ready to repair the object.

The replicas that appear as children of the fake directory are accessible via the normal Unix API and can be examined and compared using Unix utilities like `diff`. However, normal applications are disallowed from mutating these objects to prevent any accidental destruction of evidence for the conflict. The only means of mutating an inconsistent object is a `DO_REPAIR` operation provided by Venus’ pioctl interface and is used by the repair tool to perform its function. Exporting this functionality as a pioctl allows it to be used by the repair tool as well as by applications that want to provide their own specialized repair facility. To make the interface general and extensible, the input to the `DO_REPAIR` operation is specified in a *repair-file* – new commands can be added easily to this file without modifying the interface to the operation.

Since the repair-file is constructed by a user, it may contain errors – for example, the repair commands may violate security constraints or system invariants. Therefore, each repair command must be *validated* before its updates are performed. Even after the validated repair commands have been executed, the replicas of the inconsistent object are not guaranteed to be identical. As a result, an additional responsibility of the system is to verify that the object being repaired has identical replicas before it is made accessible to normal applications.

The `DO_REPAIR` operation is performed at the servers on behalf of the user. It may be aborted if the server crashes or finds an incorrect command in the repair-file. In either case, undoing the partial mutations can be cumbersome, especially if multiple objects have been mutated. To make the `DO_REPAIR` operation *fault-tolerant*, the repair commands are executed within the scope of a transaction. Updates are made in memory as each command is executed and committed atomically to stable storage at the end of the `DO_REPAIR` operation. To prevent exposing partial results of the repair, an exclusive lock is held on the objects being mutated for the duration of the `DO_REPAIR` transaction. Atomicity of the updates guarantees the objects being repaired will be in a mutually consistent state if the operation is aborted.

## 6.2 Repair Tool: Design and Implementation

When a user requests service for an inconsistent object, Venus prints an advisory message on the console of the client workstation and denies access to the object. The user must repair the object before he can access its data.

Briefly, a repair session proceeds as follows. The user begins the session by requesting the system to expose the replicas of the inconsistent object. Venus explodes the object in-place into a fake directory, and makes each of its replicas accessible as a child of this directory. The user cannot mutate these replicas but he may examine them using any Unix utility to determine the repair strategy. The user's repair request is propagated to the servers by Venus. Each server validates and performs the commands requested by the user. If the repair is successful, Venus implodes the fake directory back into a regular object and fetches its status and contents from the servers. However, if the repair is unsuccessful, the object's replicas are still diverging and it remains in the fake directory state.

A user's repair request differs according to the kind of object being repaired. To repair an inconsistent file, the user prepares a file whose contents replace the diverging replicas. However, to repair a directory, the user provides a file containing the compensating updates to be performed on each replica. This distinction arises due to the difference in the structural properties of files and directories. Since directories are navigational objects, their structural integrity is crucial for the accessibility of their descendants. Therefore, to prevent a corruption in the directory structure, the servers never accept entire directory contents from clients but perform the directory operation themselves. On the other hand, for a file update or repair, they do accept its updated contents from the client.

The rest of this section describes the implementation details of the repair facility. Coda provides separate tools for repairing directories and files due to the difference in their repair methodology. However, the two tools use the same methodology for preserving and exposing the diverging replicas of an object. Therefore, this feature is described first. Then, the two tools are described in separate sub-sections.

### 6.2.1 Preserving and Exposing Replicas

An inconsistent object is presented to the user as a dangling symbolic link. Thus, applications are prevented from using or modifying inconsistent data. The target name in the symbolic link has a special format containing the `fid` of the inconsistent object. For example the result of the Unix command, `ls -lL`, for an inconsistent object `foo` would be

```
lr--r--r-- 1 root   27 Nov  4 09:35 foo -> @7f00021e.1c5.9e6
```

where `7f00021e.1c5.9e6` is the `fid` of the object `foo`.

When an application tries to access an inconsistent object, the request returns with an `ESYMLINK` error code. Instead of adding a new error code to the Unix API, we used an existing error code for backward compatibility. Since most Unix applications are expected to handle the `ESYMLINK` error code, they can be used unmodified in the Coda environment. Of course, the disadvantage of this approach is that users of these applications need to be trained to recognize this special symbolic link as a representation for an inconsistent object.

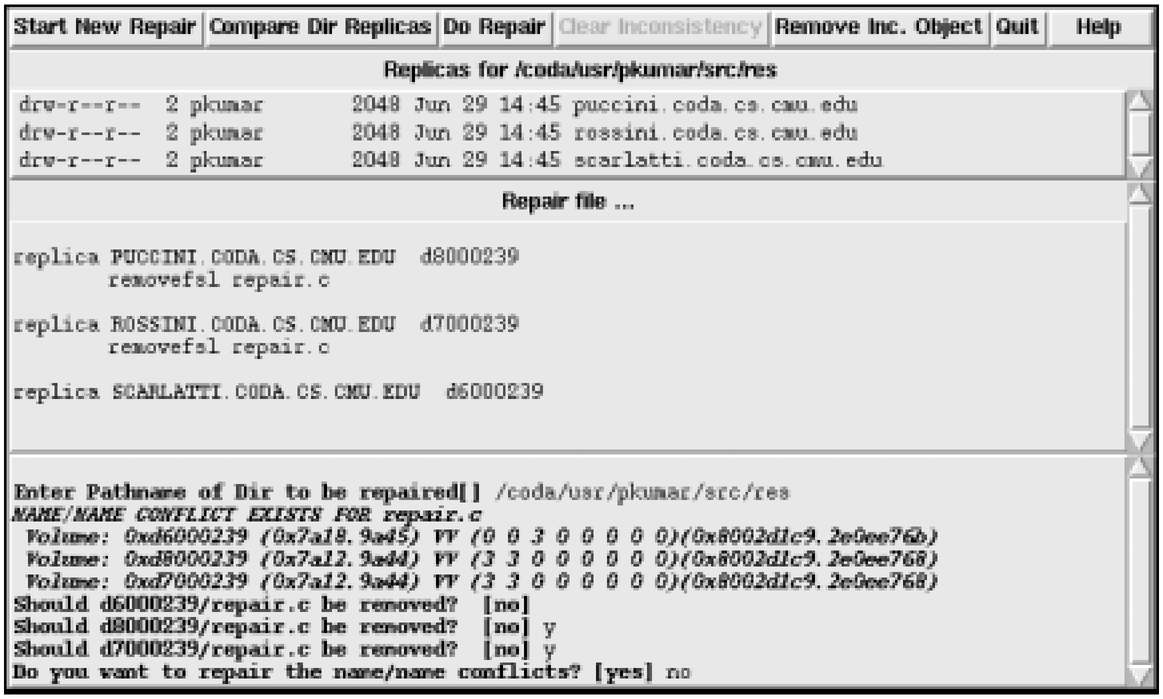
Since the client-server interface is not visible to applications, the servers use a distinct error code, `EINCONS`, to inform the client about the inconsistency. On receiving this error code, the client presents the object as a symbolic link as described above.

To perform the repair, users need to access the replicas of an inconsistent object. Therefore, once the replicas of an inconsistent object have been exposed to the user, the servers must service requests for that object instead of returning an error code. To distinguish between requests made after and before the replicas are exposed, Venus uses a different `fid` parameter for the requests.

Each replicated object in Coda has two fids: a *replicated* `fid` that is shared by all replicas of the object and a *non-replicated* fid that is unique to each replica. A request for an inconsistent object is serviced only if it is made using the non-replicated `fid`. Venus uses this `fid` to make requests for the inconsistent object's replicas after they have been exposed to the user. Requests that are made prior to this event use the replicated `fid` and result in an error condition. Note that no mutating requests are serviced for the individual replicas.

### 6.2.2 Directory Repair Tool

The directory repair tool user-interface, implemented using Tcl/Tk, is shown in Figure 6.1. Another version of the tool based on terminal input/output is also available for users not running a windowing environment. The tool provides three basic commands. The `Start-New-Repair` command allows a user to inform Venus that she is ready to repair an inconsistent directory. The tool confirms that the object is inconsistent and then passes the request to Venus using the `BEGIN_REPAIR` pioctl. As mentioned before, Venus explodes the directory in-place into a fake directory. Each replica of the directory is uniquely named using the name of the server it is stored at. For example, the replicas of an inconsistent directory named `foo`, that is replicated at two servers `server1` and `server2`, will be accessible as `foo/server1` and `foo/server2` after the in-place explosion.



```
drw-r--r--  2 pkumar      2048 Jun 29 14:45 puccini.coda.cs.cmu.edu
drw-r--r--  2 pkumar      2048 Jun 29 14:45 rossini.coda.cs.cmu.edu
drw-r--r--  2 pkumar      2048 Jun 29 14:45 scarlatti.coda.cs.cmu.edu
```

```
replica PUCCINI.CODA.CS.CMU.EDU  d8000239
        removefsl repair.c

replica ROSSINI.CODA.CS.CMU.EDU  d7000239
        removefsl repair.c

replica SCARLATTI.CODA.CS.CMU.EDU  d6000239
```

```
Enter Pathname of Dir to be repaired[] /coda/usr/pkumar/src/res
NAME/NAME CONFLICT EXISTS FOR repair.c
 Volume: 0xd6000239 (0x7a18.9a45) VV (0 0 3 0 0 0 0 0)(0x8002d1c9.2e0ee76b)
 Volume: 0xd8000239 (0x7a12.9a44) VV (3 3 0 0 0 0 0 0)(0x8002d1c9.2e0ee768)
 Volume: 0xd7000239 (0x7a12.9a44) VV (3 3 0 0 0 0 0 0)(0x8002d1c9.2e0ee768)
Should d6000239/repair.c be removed?  [no]
Should d8000239/repair.c be removed?  [no] y
Should d7000239/repair.c be removed?  [no] y
Do you want to repair the name/name conflicts? [yes] no
```

The topmost row of buttons is used to invoke the tool's commands. The `Clear-Inconsistency` command, that makes an inconsistent object accessible to normal applications, should be used with caution and therefore is disabled for normal use. The `Remove-Inc-Object` command is a macro to repair and delete an inconsistent object. The top frame displays the replicas of the directory being repaired, i.e. `/coda/usr/pkumar/src/res`. This directory is replicated at three servers `scarlatti`, `puccini` and `rossini`. The contents of any replica can be viewed by double-clicking on its name. The repair-file generated by the `Compare-Dir-Replicas` command is shown in the next frame. The file can be edited in the frame before it is used by the do-repair command. The bottom frame is used to obtain user input and display advisory messages.

Figure 6.1: User Interface for the Directory Repair Tool

The second command, `Do-Repair`, is used to request each server to perform compensating operations. The compensating operations are specified in a repair-file, whose name is supplied as a parameter of this command. The tool checks the repair-file for syntax errors before propagating the request to the servers. The repair-file has a special format containing multiple groups of commands separated by the name of the server they are to be executed at. Each group contains zero or more commands that are executed sequentially at each server. Figure 6.2

shows an example repair file.

```
replica GRIEG.CODA.CS.CMU.EDU cc000110
        removefsl admon.h

replica WAGNER.CODA.CS.CMU.EDU ce000110
        removefsl admon.h

replica HAYDN.CODA.CS.CMU.EDU cd000110
        createf advice.o 7f000116 2a2 206b
        createf adsrv.h 7f000116 2a8 206c
        mv lwp.c.BAK lwp.c 7f000194 535 4867 535 4867
```

The directory being repaired is replicated at three servers. `7f000116` is the replicated volume-id corresponding to the three volume-ids `cc000110`, `ce000110` and `cd000110`.

Figure 6.2: An Example Repair-file

To simplify the process of producing a repair-file, the tool provides a `CompareDirReplicas` command. The goal of this command is to compare a directory's replicas and generate a list of suggested compensating commands to be executed by each server. These commands are not executed by the servers automatically; the user has ultimate control over the list of commands sent to the server.

During the comparison, objects that exist at all replicas are ignored. Version-vectors of the child objects are used to distinguish recent subset creates from subset removes. For the former kind of objects, a compensating command to create the missing replicas is generated automatically. However, for the latter kind of objects, a compensating command to remove the existing replicas is confirmed with the user. There are four situations in which user confirmation is needed to decide the compensating command:

1. **Name/Name conflict**: In this situation, two or more objects with the same name are found in separate replicas of the same directory. The user decides which object to preserve.

2. **Remove/Update conflict**: In this situation, an object is removed from a subset of the directory's replicas. Some replicas of the object weren't removed by the automated resolution facility because they were modified since they were partitioned from the replicas that were removed. User confirmation is needed to decide if the updates are to be preserved or discarded.

3. **Rename/Rename conflict**: An object's replicas are renamed into different directories. The user decides which directory should contain the object.

4. **Access-control list updates**: The access-control lists of a directory's replicas were modified during a partition. The user decides the changes required to make the lists identical.

#### 6.2.2.1 Compensation at each Server

Compensation at each server is initiated by Venus making an RPC named `ViceRepair`. The parameters of this call include the `fid` of the directory to be repaired and the repair-file. The repair-file is shipped in its entirety. Venus could send each server only those commands corresponding to the replica it stores. However, the repair file is typically small, containing less than ten commands per replica. Thus the simplicity of transferring the entire file outweighs the overhead of sending extraneous information to each server. Table 6.1 shows the complete list of compensation commands and their parameters.

| | |
|---|---|
| `createf <name> <fid>` | Create the named file |
| `creates <name> <fid>` | Create the named symbolic link |
| `createl <name> <fid>` | Create a link to the specified object |
| `created <name> <fid>` | Create the named sub-directory |
| `removefsl <name>` | Remove the named file, link or symbolic link |
| `removed <name>` | Remove the sub-tree rooted at the named sub-directory |
| `mv <src-name> <tgt-name> \` | Rename object from `<src-name>` to <tgt-name> |
| `<src-dfid> <tgt-dfid>` | `<src-dfid>` is the parent's `fid` before the rename |
| | `<tgt-dfid>` is the parent's `fid` after the rename |
| `setowner <owner-id>` | Set a new owner-id for the directory |
| `setmode <mode-bits>` | Set the directory's mode-bits |
| `setmtime <time>` | Set the time a directory was last modified |
| `setacl <user> <rights>` | Set the user's rights in the access-control list |
| `delacl <user>` | Delete a user's rights |

All commands except `mv`, assume they are to be executed in the directory being repaired. Commands for deleting objects do not specify any `fid` because each name within a directory uniquely identifies the object.

Table 6.1: Command Syntax for Repairing a Directory

On receiving a repair-file, each server selects the commands for the replica it manages. Before executing the commands, it locks the vnodes of objects it will modify during the repair. The list of objects to be locked is computed from the parameters of the commands specified. Once the vnodes are locked, the server validates and executes the commands sequentially. To validate a command the server makes three checks:

- Security: This check ensures that a user can repair only those objects he has appropriate rights for.
- Integrity: This check ensures that executing the repair command will not violate any system invariant, for example, a directory must never contain duplicate entries.
- Resource: This check ensures that a repair command can be executed without overflowing the enclosing volume's quota.

Each command is executed only if it is validated successfully. The mutation is performed on a volatile copy of the object and committed atomically to stable storage only if all commands complete successfully. If some command fails validation, the repair session is aborted, its mutations are undone and an appropriate error code is returned to the client. The updates are performed atomically using RVM. Details of the implementation have been omitted here because this approach is similar to the method used by the automated resolution facility as described in Section 4.4.1.1.

An important implementation detail concerns the execution of a repair command that creates a directory. Consider a repair session in which the user requests the creation of a sub-directory `foo`. The request is prompted by the existence of `foo` at another replica and the user's wish to preserve the sub-tree rooted at `foo`. Should the server creating `foo` copy the entire sub-tree from the replica where `foo` already exists? Or, should the user be forced to recursively repair all directories in the tree rooted at `foo`? Using the former strategy could make the repair session very expensive while using the latter strategy would inconvenience the user immensely. Instead, Coda uses the resolution facility to populate the newly created directory replica.

The directory replica created during a repair session is empty and marked with a special *null* version-stamp. Such objects are called *runts*. The non-runt replicas of this directory, that existed before the repair, have a different version-stamp. Therefore, resolution will be invoked for this directory when a user tries to access it. The resolution protocol treats such objects specially – instead of using the resolution logs to perform compensating operations, it populates the runt-replica with the contents of one of the non-runt replicas. A sub-directory of the runt being populated is created as a runt and is populated only when it is resolved. Thus, in the example above, the tree rooted at `foo` is populated lazily, as and when a user accesses its descendants.

#### 6.2.2.2 Completing the Repair

Since the repair-file may be generated manually by a user, the replicas of a directory being repaired are not guaranteed to be identical after the compensating commands have been executed. In fact, some directories are repaired in several iterative steps. For example, the first iteration may delete some objects to repair a name/name conflict while the second iteration creates the missing replicas of the object that wasn't deleted. Therefore, after executing the compensating commands, each server stamps the directory replica it manages with a unique version number. This guarantees that a comparison of the replicas' version numbers indicates divergence. Furthermore, the inconsistency flag on each replica is cleared only after the directory replicas have been verified for equality.

Since Venus already has the machinery to fetch a directory's replicas from the servers, it is logical to make it responsible for verifying their equality. Unfortunately, this would compromise security since the client is not trusted. Therefore, this task is performed at the servers with the client being responsible only for its activation. If the compensation commands are executed successfully, Venus requests one of the servers to verify that the replicas are equal.[1] The appointed server fetches the directory's replicas from all servers in the AVSG and compares them byte by byte. If the replicas are equal, it generates a new version stamp for the directory and requests all servers to clear the inconsistency flag. Now, the directory can be accessed by normal Unix applications again.

#### 6.2.2.3 Repairing with Incomplete VSG

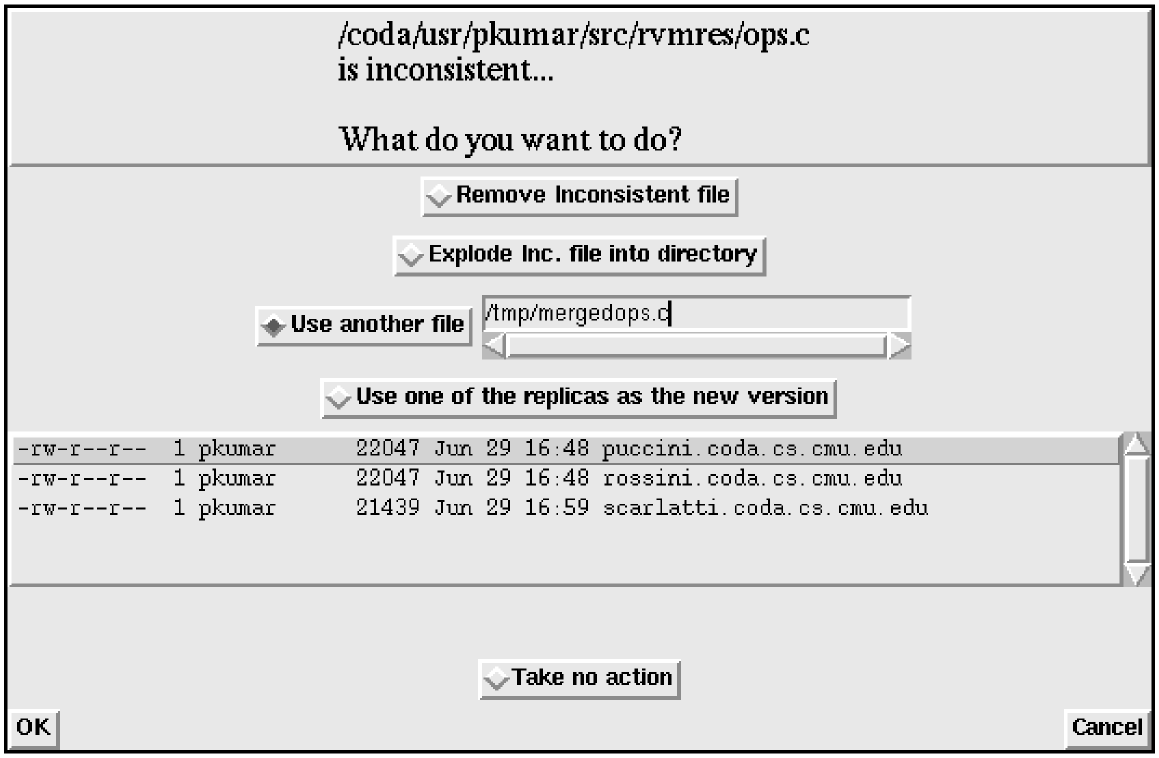
Directory operations executed during a repair are not recorded in the resolution log for two reasons. First, the reason for performing the repair may be an over-full resolution log and spooling more log records during the repair may be impossible. Second, the user may not want the operations propagated automatically to the inaccessible replicas. For example, a user may decide to remove a file `foo.c` while repairing a directory named `bar`. However he may want to preserve another replica of `bar/foo.c`, that was inaccessible during the repair. In any case, if a repair session does not include all VSG members, the directory must be repaired again when the AVSG expands.

To achieve this functionality, each server inserts a *repair-record* in a directory's resolution log while performing compensation. To ensure that a record can be inserted, it deletes all previously spooled records for this directory. Resolution is invoked when the directory's AVSG expands, since the version stamps of the repaired replicas are different. If the resolution protocol finds a repair-record in the resolution log of a subset of the replicas, it marks the directory with an

[1]This request is made using the `ViceResolve` RPC. The coordinator implementation recognizes a resolution request for an inconsistent directory as a request to verify the replicas' equality.

inconsistency flag, thus forcing the user to repair the directory again. However, it ignores the repair-record if the record is found in the logs of all replicas.

### 6.2.3 File Repair Tool


/coda/usr/pkumar/src/rvmres/ops.c
is inconsistent...

What do you want to do?

```
-rw-r--r--  1 pkumar      22047 Jun 29 16:48 puccini.coda.cs.cmu.edu
-rw-r--r--  1 pkumar      22047 Jun 29 16:48 rossini.coda.cs.cmu.edu
-rw-r--r--  1 pkumar      21439 Jun 29 16:59 scarlatti.coda.cs.cmu.edu
```

The tool displays the replicas of the inconsistent file and the list of options for the repair. The inconsistent file `/coda/usr/pkumar/src/rvmres/ops.c`, shown in the figure, is replicated at three servers `scarlatti`, `puccini` and `rossini`. The user chose to repair it with a new file named `/tmp/mergedops.c` that he created. Instead, the user could have chosen to use one of its replicas by clicking on the appropriate replica's name.

Figure 6.3: User Interface for the File Repair Tool

The user-interface of the file repair tool is also based on Tcl/Tk and is shown in Figure 6.3. The name of the inconsistent file is provided as a command line argument to the tool. Like the directory repair tool, it exposes the replicas of the file using the `BEGIN_REPAIR` pioctl. The file can be repaired in one of four ways. It can be

1. replaced by a new file prepared by the user, or

2. replaced by one of the file's replicas itself, or
3. removed, or
4. exploded into a real directory containing all the file's replicas. In other words, each replica of the file is now replicated at all servers in its VSG.

The basic operation used by the tool to implement these options is the `DO_REPAIR` pioctl provided by Venus. Recall that this operation takes two inputs: the name of an inconsistent file and the name of a repair-file. The contents of the repair-file are used to set the diverging replicas of the inconsistent file to a common value. It is obvious how the tool uses this pioctl to implement the first two options. To remove an inconsistent file, the tool uses a temporary empty file to perform the repair and then deletes the repaired file using the `unlink` system call. To implement the last option, it creates a temporary replicated directory and copies the replicas of the inconsistent file into this directory. Then it deletes the inconsistent file, as described above, and renames the temporary directory to have the name of the inconsistent file. Note that unlike the directory repair tool, the file repair tool does not provide a compare-replicas command, since a file's replicas can be compared easily using the `diff` utility.

A file repair request is propagated to the servers by Venus using the `ViceRepair` RPC. The parameters of this call include the `fid` of the inconsistent file, a version-vector that dominates the vectors of all accessible replicas and the new contents of the file. Note that the `ViceRepair` RPC was also used for a directory repair but the version-vector parameter is ignored in that case.

An optimistic concurrency control strategy is used to prevent multiple users from repairing a file simultaneously. A client makes a repair request assuming no other client is repairing the file. When performing the repair each server verifies that the version-vector of the file being repaired is submissive to the vector sent by the client. This test will fail, and the repair will be aborted, if the file has changed since the client initiated the `ViceRepair` request.

Due to considerations of security, the servers perform the repair only if the user has sufficient rights to modify the file being repaired. The contents and version-vector of the inconsistent file are updated using Coda's two-phase file update protocol, as described in Chapter 5. Since the file's replicas are updated simultaneously using the same repair-file, they are guaranteed to be identical at the end of the `ViceRepair` RPC. Thus, unlike directories, the inconsistency flag on a file is reset automatically at the end of the `ViceRepair` RPC. If the RPC returns successfully, Venus implodes the fake directory back into a file and makes it accessible via the Unix API once again.

# Chapter 7

# Evaluation

The resolution mechanisms presented in this thesis have been implemented as a part of the Coda file system. Coda is being used on a daily basis since 1991. Currently it supports over thirty-five users and this community is still growing. This chapter evaluates Coda's resolution mechanisms empirically and quantitatively. The empirical evaluation is based on evidence from real use of the system, while the quantitative evaluation is based on controlled experimentation using file system traces as well as synthetic benchmarks. Before discussing the evaluation, this chapter describes the implementation status of the resolution mechanisms.

## 7.1 Status and Usage Experience

At the time of writing, there are ten Coda servers storing approximately four Gigabytes of data. Access to this data is provided by approximately seventy-five clients. The resolution mechanisms have been very successful in making the system more usable. Over a period of 10 months, from September 1993 to June 1994, there have been 12,496 attempts to automatically resolve directories and 20,370 attempts to automatically resolve files. 98.9% of the former and 99.4% of the latter were completed successfully.

### 7.1.1 Evolution

Coda's resolution mechanisms were developed in several stages. The first version of the system contained only the simplest resolution mechanism, i.e. version-vectors. This version of the system could automatically resolve files that had been updated in a single partition. It could detect a directory with diverging replicas but was unable to perform the resolution automatically. Therefore, it included a repair tool so that diverging directory replicas could be resolved manually. This version of the system was operational by June 1990 and was used by four users. Experience with the system made it clear that a mechanism for automating the resolution of a directory was essential for usability.

A preliminary version of log-based directory resolution was operational by June 1991. This version of directory resolution stored resolution logs in volatile memory only. By doing so, the system did not have to address issues related to log storage management or fault-tolerance. However, this simplification came at a price – a directory could be resolved only if the servers storing its replicas had not been restarted since the updates occurred. This version of the system was used by eight people. Experience with the system convinced us that the storage space needed to log directory updates was not excessive and that a log-based approach for directory resolution was feasible in a real production system.

The latest version of the system which records directory updates in recoverable storage has been operational since Spring 1992. It is being used on a daily basis by a community of over thirty-five users. This implementation of directory resolution has two limitations – partitioned updates to directory access-control lists and partitioned rename operations involving multiple directories must be resolved manually. We chose not to implement the enhancements required to automate the resolution of these two operations for two reasons. First, these enhancements require a change in the recoverable data structures and a complete re-initialization of our system – a task that would inconvenience our users and have a high administrative cost. Second, the enhancements are not critical since these operations occur rarely in practice. Analysis of file system traces from our environment shows that less than 3% of all directory updates are cross-directory renames. The number of updates to directory access-control list cannot be estimated because these updates are not uniquely identifiable in the traces. However, anecdotal evidence suggests that such updates are rare in our environment.

Application-specific resolution of files was developed in two stages. The first stage was operational by June 1993 and only included the mechanism to find and transparently invoke an ASR. It did not guarantee fault-tolerance of the execution of the ASR. The fault-tolerance mechanism was implemented in the second stage that has been operational since May 1994. Since the ASR mechanism was made available to the user community only recently, its evaluation is based on limited experience. However, we hope to gain more experience with the ASR mechanism since it will also be used in an extension of Coda to support Isolation-Only Transactions [30]. The goal of this project is to provide users with transactions at the file-system level so that write-write as well as read-write conflicts can be easily detected. Once the conflict is detected, an ASR is used to resolve the conflict automatically.

The repair facility has been operational since the first version of the system was released in June 1990. At the beginning the repair facility did not provide the `CompareDirReplicas` command. Therefore, users were forced to create the repair-file manually. The next version of the repair facility that provided this command was implemented by December 1990. Recently, in May 1994, the repair facility was modified to provide a graphical user-interface in addition to the standard terminal based interface. The new interface is based on Tcl/Tk.

### 7.1.2 Coda Environment

The ten servers servicing the Coda community are divided into two groups of three servers each and another group consisting of four servers. The former two groups of servers, running the omega and beta versions of the server programs, store three-way replicated volumes. The last group of servers run the alpha version of the server programs and store two- to four-way replicated volumes.

The omega servers, running the most stable version of the software, store volumes containing the home directories of the Coda user community as well as the source trees for application software. In fact, the source trees for Coda and some related projects are stored entirely within the Coda file system. To prevent loss of data due to hardware crashes or catastrophic software errors, all data on the omega servers is backed up to tape every night.

Volumes stored on the beta servers contain files that can be reproduced easily – for example, object files produced by compiling source files and files belonging to applications like `gnu-emacs`, LaTeX etc. that are archived regularly and available freely on the Internet. Therefore, these servers can be reinitialized easily in the event of an unrecoverable failure. To stress test the beta server software, each member of the Coda group has a volume on these servers which he or she uses regularly to compile new software modules.

The alpha servers are used for testing new server releases and therefore store temporary volumes. The server software can execute on two hardware platforms – the Mips R2000/R3000 and the Intel x86 series. However, all server hosts in our environment are the 5000/200 series DECstations containing the Mips R3000 CPU. Each omega and beta server has a disk storage capacity of two Gigabytes. At the time of writing, each server contains about 1.7 Gigabytes of data.

There are two kinds of clients in the Coda environment – desktop and mobile hosts. The desktop hosts are mostly 5000/200 series DECstations while the mobile hosts are mostly DECpc 425SL laptops. In addition, there are a few 3100 series DECstations, Sun Sparcstations and IBM PS/2-L40 laptops in our environment. Servers and clients run version 2.6 of the Mach operating system [1].

### 7.1.3 Usage

The nature of partitioned updates and their subsequent resolution is influenced by the kind of data and its usage in our environment. There are currently 225 volumes managed by the Coda servers. Forty-three of these volumes are *user* volumes containing the home directories of users in the Coda community. To stress-test beta versions of the server software, members of the Coda group store object files in twenty *object* volumes managed by the beta-servers. The alpha servers manage 17 *scratch* volumes that are used to test new server software releases. 110 *Project* volumes contain files shared by project members, for example, project sources and libraries of various software modules. Seven projects store their files in project volumes, but most of these volumes are used for the Coda project itself. To support disconnected operation, certain critical applications like `gnu-emacs`, X11, C++, etc. are stored in 35 *application* volumes. Storing these applications in Coda allows users to utilize the small disks on mobile hosts effectively. For example, a user can cache only those X11 applications he wants to run while disconnected and still have sufficient space to cache other personal files also needed during the disconnection.

Ten out of the forty-three user volumes belong to members of the Coda group who use the system on a daily basis. Fifteen user volumes belong to users who use the system actively, i.e. they have two or three sessions of file-system activity per week, while eighteen user volumes belong to users who use the system occasionally. Although Coda's user community is quite large it is important to realize that it is homogeneous and consists of users that perform similar tasks. All of the users are researchers in a University environment. They use the file system primarily for two tasks – program development and document processing.

The frequency of resolution depends on the degree of write-sharing and the frequency of network partitions in the environment. Files contained in a user volume do not exhibit a high degree of write-sharing because they are usually modified by the owner of the volume. Application volumes also do not exhibit write-sharing for two reasons. First, they are updated infrequently, only when a new version of the application is released, and second they are typically updated by one user, the maintainer of the application software.

Objects belonging to project volumes exhibit some write-sharing between members of the project. However, we found the degree of write-sharing to be less than expected and can attribute two reasons for this. First, project members share files using a revision control system and make private copies of the shared files while they are developing the software. Thus, the shared copy of a file is updated only when the user is ready to check in its final version. Second, project members typically develop independent modules of the system and therefore modify independent sets of files contained within the same volume.

Even though a file or directory is write-shared infrequently between users, it is often "shared" by one user accessing it from different clients. For example a user developing a software module may use one client to run his windowing system and an editor, and use another client as the compile engine. This scenario is common in our environment since most Coda users have access to at least two clients, a mobile laptop computer and a desktop host; and, the mobile host is often connected to the network during the day to ensure that its cache contains the latest data.

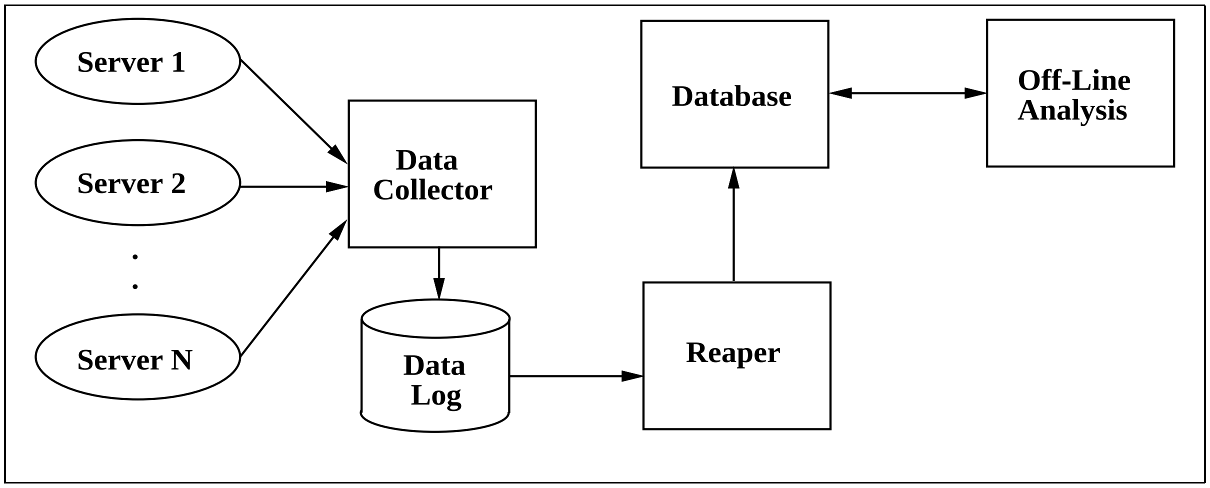


Figure 7.1: Data Collection Architecture

Anecdotal evidence indicates that partitioned activity is quite common in our environment. Partitioned activity occurs whenever only a subset of the servers in an object's VSG are accessible. A server may be inaccessible because of a network or software failure, or because it is shutdown for software maintenance. Coda's shutdown procedure ensures that at least one server in each VSG is accessible at all times. All these events provide ample opportunity for partitioned activity in our environment.

As a result of the usage patterns described above, resolution is triggered most often by intra-user cross-client sharing or partitioned activity during a server shutdown or failure. Since the updates triggering such resolutions originate from the same user, the likelihood of a conflict is low. This reason combined with the fact that write-sharing across users is rare in the Unix environment accounts for the high success rate of resolution.

## 7.2 Empirical Measurement Framework

To evaluate the techniques presented in this thesis, data was collected during real use of the Coda system. This section presents the framework for data collection. Results from these measurements are presented in the following two sections.

Figure 7.1 shows the architecture used to collect data from the Coda servers. Data resulting from each server's instrumentation is sent to a central data collector that stores the data in a log on its local disk. To minimize loss of data due to failures, each server sends data to the collector every two hours. A reaper processes this data and inserts the results into a database. Storing the data in a database allows us to answer various questions about the system's usage long after the data has been collected. The answers are provided by querying the database using SQL [7].

This framework was implemented by Brian Noble. Details of its design and implementation are provided in [35]. The Coda servers were instrumented using this framework for a period of ten months, from September 1993 to June 1994.

## 7.3 Evaluation of Directory Resolution

The success of directory resolution in practice can be measured by counting the number of instances in which it automatically resolves partitioned updates. Empirical results from our environment confirm that directory resolution is very effective. As shown in Table 7.1, 98.9% of all directory resolution attempts were successful. Note that 88.9% of the successful resolutions did not need to validate any transaction. In each of these instances the replicas of the directory being resolved were equal even though they appeared to be diverging. This event occurs frequently in our environment due to the intra-user sharing mentioned in the previous section – a user updates a directory from one client and accesses it from another client before the second phase of the update is complete. At that point, the directory's version-vectors[1] are unequal and make the replicas appear to be diverging even though they are not.

Although directory resolution is very successful in the Coda environment, it is important to understand that the Coda user community is homogeneous and not representative of users outside the research environment. Therefore, it is possible that resolution will be less successful in other environments, such as a large product development site, that exhibit different file usage patterns.

Directory resolution fails for two reasons: (a) because the partitioned transactions on the replicas are non-serializable or (b) because of shortcomings in the implementation including the scarcity of resolution log space. Table 7.1 shows the number of conflicts caused by each of these problems. Create transactions exhibited the highest number of conflicts. Anecdotal evidence suggests that these conflicts are often caused due to the creation of a file named `core` in the same directory in multiple partitions. The inability to resolve cross-directory renames caused only 54 resolution attempts to fail. Unfortunately, we cannot report the number of failures caused by partitioned access-control list operations since they were not counted by the servers. The number of instances in which the resolution log filled up was much higher than our expectation. Further analysis showed that the maximum log size for the corresponding volumes was erroneously set too low by the system administrators.

[1]Version-vectors for directories do not serve any purpose during resolution and are maintained purely for historical reasons.

| **Directory Resolutions** | Count |
|---|---|
| Attempts | 12,496 |
| Aborts | 58 |
| Conflicts | 82 |
| Successes | 12,356 |
| 0 transactions validated | 10,981 |
| $\geq$ 1 transaction validated | 1,375 |
| **Non-serializable Transactions** | |
| Create | 35 |
| Remove | 14 |
| Update (`store` etc.) | 10 |
| **Implementation Limitation Conflicts** | |
| Rename | 54 |
| Log Wrap | 45 |

This table shows the results of directory resolutions observed from September 1993 to June 1994. The resolution attempts are classified into three classes: those that had to be aborted, those that resulted in conflicts and those that succeeded. The number of non-serializable transactions observed in the same time period are also shown. The last group shows the number of resolution failures due to shortcomings in the implementation.

Table 7.1: Observed Directory Resolutions

An important point to note is that the number of conflicts reported in Table 7.1 is overestimated. A conflict is typically detected by more than one server participating in the resolution and therefore is counted multiple times towards the total.

Our implementation of directory resolution incurs both time and space overheads. The time overhead occurs at two points – first during the mutation to create and spool the resolution log records and second during the execution of the resolution protocol. The space overhead arises from the need to maintain logs at servers. The rest of this section answers the three obvious questions that follow from these observations:

- What is the effect of logging on system performance?
- How fast does the log grow during partitioned operation?
- How well does resolution perform?

### 7.3.1 Performance Penalty due to Logging

Spooling resolution log records slows down the servers since they need to do more work for each update transaction. Obviously, the increased latency of each mutation slows down the client also. This section answers two questions: How does the latency of each operation performed at a server with logging support compare with the latency of the same operation without logging support? And, what is the increase in latency perceived by the client?

#### 7.3.1.1 Methodology

To answer these two questions we used two sets of experiments.

The first set of experiments measured the latency of updates at a server. The latency of an update transaction is the elapsed time between the server receiving a request and sending the response to the client. Each experiment consisted of generating an update request at a client and measuring its latency at the server using a microsecond timer. Measurements were made for seven kinds of update transactions, `create`, `link`, `unlink`, `rename`, `mkdir`, `rmdir` and `symlink`. To measure the overhead of logging, each experiment was conducted in two configurations, i.e. with and without logging support at the server. Mutations were performed only on non-replicated objects in order to minimize the overheads due to replication.

To measure the latency of mutations perceived by the client, the second set of experiments timed a well defined task, the Andrew benchmark [22]. The benchmark takes as input a source sub-tree and operates on a target sub-tree in the file system to be measured. It performs its operations in the target tree in five phases: the *Makedir* phase creates sub-directories; the *Copy* phase copies files from the source tree; the *ScanDir* phase opens all directories and examines the status of all files; the *ReadAll* phase opens and reads all files; finally the *Make* phase compiles an application. The experiment consisted of running the benchmark at the client and measuring the elapsed time of each phase. It was conducted under two server configurations: with and without logging support at the server.

To minimize experimental error, extraneous user applications like X, and system daemons like `cron` and `atd` were killed for the duration of both experiments.

#### 7.3.1.2 Results

Since resolution logs are maintained in RVM, spooling a log record at the end of a transaction requires a few additional modify-records to be appended to the RVM log. Therefore the time overhead for logging should be small. Results from the first set of experiments confirm this fact. As shown in Table 7.2, the time overhead to log most directory mutations at the server is around 2 or 3 milliseconds. The two exceptions are the `rename` and `rmdir` transactions.

Recall from Table 4.4 on page 75, that the log record for the `rename` transaction is the biggest and most complex. Thus, the overhead for spooling this record is high. The `rmdir` transaction has a high overhead because it needs to scan and process the log of the sub-directory being removed.

| **Transaction** | **Server Configuration** | | **Overhead** | |
|---|---|---|---|---|
| **Type** | No Logging | Logging | Time | % |
| `create` | 25.0 (4.7) | 27.7 (1.7) | 2.7 | 10 |
| `link` | 23.3 (5.4) | 25.9 (5.0) | 2.6 | 11 |
| `unlink` | 25.1 (6.4) | 27.7 (6.3) | 2.6 | 10 |
| `rename` | 21.1 (4.6) | 26.2 (3.1) | 5.1 | 24 |
| `mkdir` | 32.8 (7.1) | 34.5 (4.5) | 1.7 | 5 |
| `rmdir` | 27.2 (6.2) | 32.5 (5.7) | 5.3 | 20 |
| `symlink` | 24.4 (5.0) | 27.5 (1.7) | 3.1 | 13 |

This table shows the time for each update transaction with and without logging at the server. The last two columns show the increase in latency for each operation. This data was collected using a DECstation 5000/200 with 32 Megabytes of memory as a server. All numbers representing time are in milliseconds and are the mean value from 15 trials. Numbers in parentheses are standard deviations.

Table 7.2: Time Overhead at Server due to Logging

Results from the second set of experiments show that the time overhead due to logging is not noticeable at the client. As shown in Table 7.3, the time for each phase of the Andrew benchmark as well as its total execution time in the two server configurations is comparable. The difference between the two configurations is less than one standard deviation of the mean value. This result is expected since the overhead due to logging is much smaller than the sum of the network latency and the time for executing the operation at the server. Experimental error, which is caused by a noisy network, completely masks the overhead due to logging.

### 7.3.2 Size of Log

To limit the growth of the log, Coda uses an aggressive policy to reclaim log space. Recall that log records corresponding to non-partitioned operations, i.e. operations that succeed at all servers in the VSG, are removed as soon as the operation succeeds. However in the presence of a partition, resolution logs grow linearly with the amount of work. To estimate the rate of log growth we made two sets of measurements. First, a trace-driven simulation was used to estimate the log growth in the presence of long partitions. Second, in order to verify the results

| Task | Server Configuration | | | | Overhead | |
|---|---|---|---|---|---|---|
| | No Logging | | Logging | | Time | % |
| Andrew Benchmark | 112.5 | (3.6) | 111.5 | (1.0) | -1.0 | -1 |
| MakeDir | 2.8 | (0.4) | 3.2 | (0.3) | 0.4 | 14 |
| Copy | 20.2 | (2.6) | 20.2 | (0.9) | 0.0 | 0 |
| ScanDir | 8.0 | (0.5) | 7.7 | (0.5) | -0.3 | -4 |
| ReadAll | 13.6 | (3.1) | 12.8 | (0.8) | -0.8 | -6 |
| Make | 67.8 | (1.4) | 71.9 | (4.7) | 4.1 | 6 |

This table shows the time spent in each phase of the Andrew Benchmark. The data was collected using a DECstation 5000/200 with 64 MB of memory as a client and a DECstation 5000/200 with 32 MB of memory as a server communicating over an ethernet. The time values are in seconds and show the mean value over 5 runs of the benchmark. Numbers in parentheses are standard deviations.

Table 7.3: Time Overhead at Client due to Logging

of the simulation, the servers were instrumented to measure the log growth in practice. Each of these studies is described below.

#### 7.3.2.1 Trace-driven Simulation

The trace-driven analysis is based on about 4GB of file reference traces[2] obtained over a period of 10 weeks from 20 Coda workstations. The usage profile captured in these traces is typical of research and educational environments. These traces were used as input to a simulation of the logging component of the resolution subsystem. The simulator assumes that all activity in a trace occurs while partitioned and therefore never simulates any log truncation that might occur in practice. It maintains a history of log growth at 15-minute intervals for each volume in the system. For each directory update in the trace, the simulator increments the corresponding volume's log length by the size of the log record that would have been generated by a Coda server. At the end of simulation, the average and peak log growth rates for each volume can be obtained from its history.

Table 7.4 shows the distribution of the long-term average rate of log growth over all the volumes encountered in our traces. This average is computed by dividing the final log size for a volume by the time between the first and last updates on it. The long-term log growth is relatively low, averaging about 94 bytes per hour.

Focusing only on long-term average log growth rate can be misleading, since user activity is often bursty. A few hours of intense activity during a partition can generate much longer

[2]The file-system traces were collected by Lily Mummert.

| Bytes per Hour | Percentage of Volumes |
|---|---|
| 0 to 100 | 66% |
| 100 to 200 | 20% |
| 200 to 300 | 5% |
| 300 to 400 | 7% |
| 400 to 500 | 2% |
| > 500 | 0% |

This data was obtained by trace-based simulation and shows the distribution of long-term average growth rates for 44 volumes over a period of 10 weeks.

Table 7.4: Observed Distribution of Long-Term Average Log Growth Rates

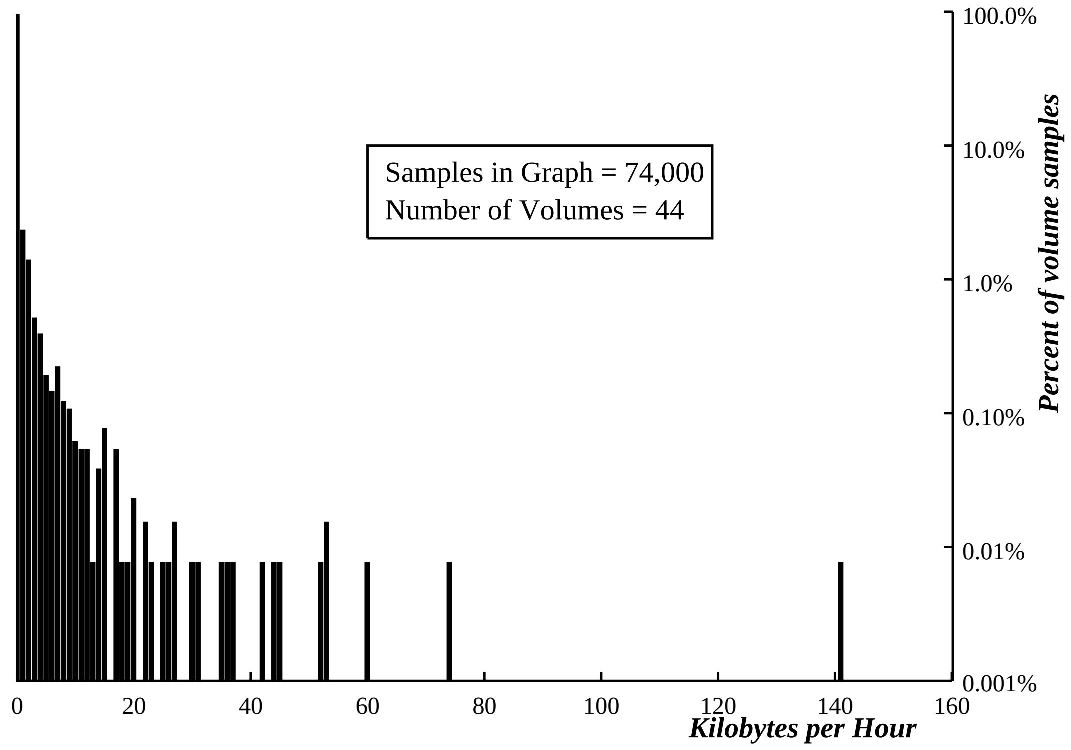
logs than that predicted by Table 7.4. To estimate the log length induced by peak activity, we examined the statistical distribution of hourly log growth rates for all volumes in our simulation. Figure 7.2 shows this distribution. Over 94% of all data points are less than 1KB, and over 99.5% are less than 10KB. The highest value observed was 141KB, but this occurred only once.

A worst-case design would have to cope with the highest growth rate during the longest partition. A more realistic design would use a log adequate for a large fraction of the anticipated scenarios. Since hourly growth is less than 10KB in 99.5% of our data points, and since an hour-long partition could have straddled two consecutive hours of peak activity, we infer that a 20KB log will be adequate for most hour-long partitions in our environment. More generally, a partition of $N$ hours could have straddled $N + 1$ consecutive hours of peak activity. Hence a log of 10$(N + 1)$ KB would be necessary. If a Coda server were to hold 100 volumes the total log space needed on the server would be $(N + 1)$ MB.

### 7.3.2.2 Measurements from the System

To corroborate the simulations, the space used by the resolution logs in practice was measured using the framework described in Section 7.2. The high-water mark reached for each volume's log each day was reported to the data collector. At the end of the day the high-water mark for each volume was stored in the database. The high-water mark is reset to the current size at the beginning of each day.

Figure 7.3 shows the distribution of the high-water mark reached each day by each volume's log. The mean of the distribution is 17.7KB implying that each volume has a modest sized log. Most of the observed high-water marks were under 200 Kilobytes with only a few outliers above the 250KB range. As predicted by the simulation, 99.5% of the resolution logs grow less than 240KB per day. However, some of the predictions made by the trace-driven simulation were gross over-estimates. For example, based on a peak log growth rate of



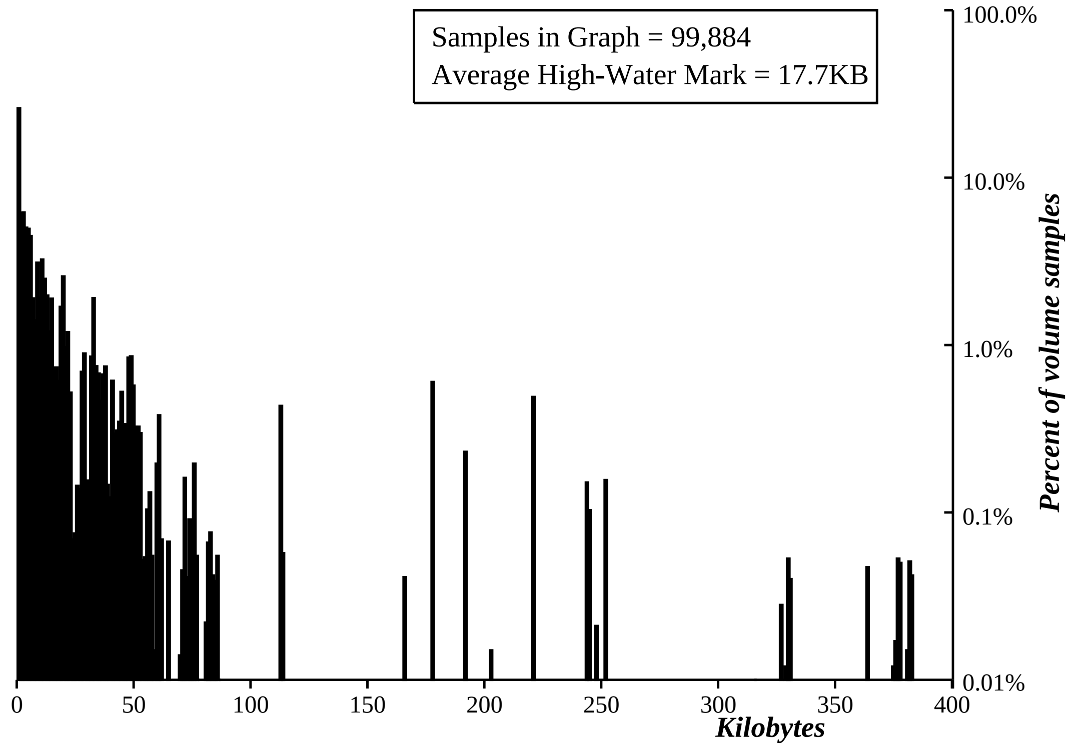
This figure shows the distribution of log growth rates for each hour for each volume reported by the simulations. The traces used for the simulation were taken from 20 workstations and spanned a 10 week period. The width of each histogram bar is 1KB. Note that the scale on the vertical axis is logarithmic.

Figure 7.2: Distribution of Hourly Log Growth

141KB/hour, the simulation predicted the maximum high-water mark to be $>$ 3.3MB. However, measurements from practice show that the maximum log size ever reached is 385KB. There are two explanations for this “anomaly”. In practice, it is unusual for heavy demand on the file system to last for an entire day. Furthermore, even if the demand is high, the resolution logs grow only when the servers are partitioned.

### 7.3.3 Performance of Resolution

A fair estimate of the time overhead due to resolution must account for the fact that resolution will take longer when there are more partitioned updates to resolve. Hence the metric we use in our evaluation is the ratio of two times: *resolution time* and *work time.* Resolution time is the elapsed time between detection of a partitioned update and return of control to the client after



This figure shows the distribution of maximum log size reached each day by each volume. The sizes were measured from resolution logs in actual use by the system from September 1993 to June 1994. The high-water mark is reset each morning to the current log size. The width of each histogram bar is 1KB. Once again, note that the scale on the vertical axes is logarithmic.

Figure 7.3: Daily High-Water Marks of Resolution Log Sizes

successful resolution. Work time is the sum of the elapsed times for performing the original set of partitioned updates.

Resolution time is perceptible to the first user to access a directory after the end of a network failure that resulted in resolvable partitioned updates. The elapsed time for failed resolution is less important, since it is swamped by the time for manual resolution.

An increase in partitioned activity lengthens phases 2 and 3 of the resolution protocol. Phase 2 takes longer because larger logs are shipped to the coordinator. Phase 3 takes longer because of an increase in the transmission time to ship a larger merged log to the subordinates, and because of an increase in the times at the subordinates for computing and applying compensating operations. An increase in the number of replicas also increases resolution time because communication overheads are higher, and the computing of compensating operations by subordinates takes longer.

| Physical Characteristic | Volume Type | | | | |
|---|---|---|---|---|---|
| | User | Project | System | BBoard | All |
| Total Number of Volumes | 786 | 121 | 72 | 71 | 1050 |
| Total Number of Directories | 13955 | 33642 | 9150 | 2286 | 59033 |
| Total Number of Files | 152111 | 313890 | 113029 | 144525 | 723555 |
| Total Size of File Data (MB) | 1700 | 7000 | 1500 | 560 | 11000 |
| Directories/Directory | 3.6 (13.4) | 3.0 (4.5) | 3.6 (10.4) | 6.8 (19.4) | 3.2 (8.3) |
| Files/Directory | 14.6 (30.6) | 16.2 (35.6) | 15.9 (36.9) | 66.9 (142.4) | 15.7 (34.5) |
| Hard Links/Directory | 3.7 (12.4) | 2.0 (1.5) | 4.0 (3.9) | 0.0 (0.0) | 3.4 (5.7) |
| Symbolic Links/Directory | 4.1 (10.1) | 3.4 (7.5) | 13.6 (45.3) | 6.0 (25.9) | 6.3 (24.9) |

This table, adapted from [12], summarizes various physical characteristics of system, user, project, and bulletin board ("bboard") volumes in AFS at Carnegie Mellon University in early 1990. This data was obtained via static analysis. The numbers in parentheses are standard deviations. The data in this table was collected by Maria Ebling.

Table 7.5: Sample Physical Characteristics by Volume Type

#### 7.3.3.1 Methodology

To quantify the above effects, we conducted a series of carefully controlled experiments using a synthetic benchmark. One instance of the benchmark, referred to as a *work unit*, consists of 104 directory updates. The execution of a work unit proceeds in three steps:

- create 20 new objects, consisting of 14 files, 4 subdirectories, 1 link and 1 symbolic link. These values approximate the composition of a typical user directory in our environment and were obtained from a static analysis of the AFS environment. Detailed results of the analysis are shown in Table 7.5. Due to the high variance in the number of links and symbolic links per directory, a work unit includes only one of each.

- simulation of editor activity on the newly-created files. This is done by creating, then removing, a checkpoint file for each file.

- simulation of a C++ compiler's activity on the newly-created files. For each such file, *foo.c,* a file *foo..c* is created; next, a file *foo..o* is created, then renamed to *foo.o;* finally *foo..c* is removed.

An experiment consists of first measuring the work time for performing a variable number of work units on each of $n$ partitioned replicas of a directory. Work-units are always performed sequentially, each unit creating a distinct set of files. When the work is to be performed at multiple replicas the client communicates with one server at any time. All work-units to be performed at that replica are completed before communication is established with another server. After the work is completed, the partitions between the replicas are healed, resolution is triggered, and the resolution time is measured.

We performed two sets of experiments, one involving partitioned work at one replica, and the other involving partitioned work at all replicas. In each set, we examined configurations involving 2, 3 and 4 replicas. For each configuration, we varied the load from 1 to 10 work units.

#### 7.3.3.2 Results

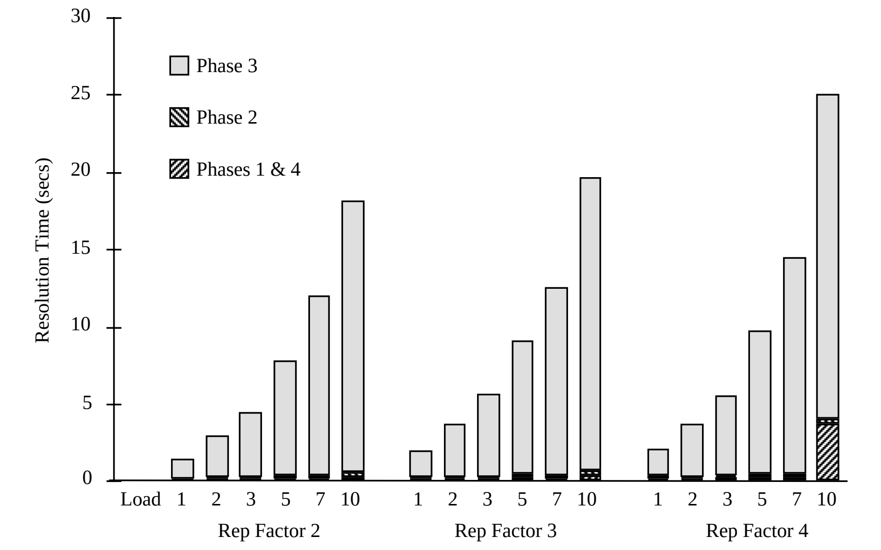
Tables 7.6 and 7.7 present the means and standard deviations of work and resolution times observed in three trials of each experiment. They also indicate the contributions of individual phases to total resolution time. The tables indicate that resolution time increases primarily with load, and secondarily with the replication factor.

The primary conclusion to be drawn from this data is that a log-based strategy for directory resolution is quite efficient, taking no more than 10% of the work time in all our experiments. This holds even up to a load of 10 at a replication of 4, corresponding to over 1000 updates being performed on each of 4 replicas of a directory.

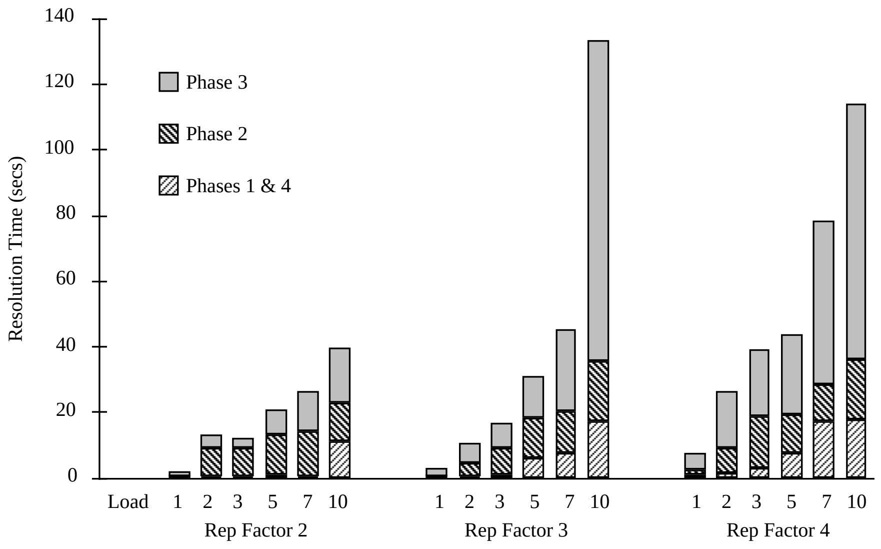
Graphs in Figure 7.4 show the contribution of each phase towards the total resolution time. Phases 1 and 4 contribute very little to the overall resolution time. Since these phases merely do locking and unlocking, the time for them should be independent of load. But, as a sanity check in our current implementation, the coordinator collects the replicas to verify equality before unlocking in Phase 4. This accounts for the dependence of this phase on load and replication factor in our experiments.

Phase 2 consists of extraction and shipping of logs by subordinates. The time for this phase is dependent on the total lengths of the logs, which is only related to the total amount of work. This is apparent in Table 7.6 where the time for phase 2 increases with load but is invariant with degree of replication. The times for Phase 2 in Table 7.7 are significantly higher than in Table 7.6. This is a consequence of our parallel RPC [51] implementation. A large log fetch from one site and zero-length log fetches from the others is much more efficient than a number of smaller, equal-sized log fetches from each site.

Phase 3 is typically the dominant contributor to the total time for resolution. This is not surprising, since the bulk of work for resolution occurs here. This includes the shipping of merged logs, computation of compensating operations, and application of these operations.



(a) Resolution Time After Work at One Replica



(b) Resolution Time After Work at All Replicas

The data for these graphs is taken from Tables 7.6 and 7.7. The graphs show the time spent in each phase of the directory resolution protocol. Most of the time is spent in the third phase. When work is done only at one of the replicas, the time spent in phase 1 and 4 is significant only at a workload of 10 units and a replication factor of 4. However when partitioned work is performed on all replicas, the time spent in these two phases is significant even at a replication factor of 2. The times spent in the second phase is independent of the replication factor.

Figure 7.4: Performance of Resolution

| Rep Factor | Load | Work Time (seconds) | Resolution Time (seconds) Total | Phase 1+4 | Phase 2 | Phase 3 | Res Time / Work Time |
|---|---|---|---|---|---|---|---|
| 2 | 1 | 27.9 (0.4) | 1.5 (0.0) | 0.1 (0.0) | 0.1 (0.0) | 1.3 (0.0) | 5.4% |
| | 2 | 69.7 (6.3) | 2.9 (0.1) | 0.2 (0.0) | 0.1 (0.0) | 2.7 (0.1) | 4.2% |
| | 3 | 111.6 (6.9) | 4.5 (0.0) | 0.2 (0.0) | 0.1 (0.0) | 4.2 (0.0) | 4.0% |
| | 5 | 188.0 (1.0) | 7.8 (0.1) | 0.2 (0.0) | 0.2 (0.0) | 7.4 (0.1) | 4.1% |
| | 7 | 352.1 (7.5) | 12.0 (0.1) | 0.2 (0.0) | 0.2 (0.0) | 11.6 (0.1) | 3.4% |
| | 10 | 563.4 (3.9) | 18.2 (0.4) | 0.3 (0.0) | 0.3 (0.0) | 17.6 (0.4) | 3.2% |
| 3 | 1 | 28.5 (2.1) | 1.9 (0.0) | 0.2 (0.0) | 0.1 (0.0) | 1.7 (0.0) | 6.7% |
| | 2 | 79.5 (2.5) | 3.7 (0.1) | 0.2 (0.1) | 0.1 (0.0) | 3.4 (0.1) | 4.7% |
| | 3 | 118.1 (10.7) | 5.8 (0.3) | 0.2 (0.1) | 0.1 (0.0) | 5.4 (0.2) | 4.6% |
| | 5 | 224.8 (10.2) | 9.0 (0.3) | 0.3 (0.1) | 0.2 (0.0) | 8.6 (0.3) | 4.0% |
| | 7 | 337.1 (9.8) | 12.7 (0.3) | 0.2 (0.1) | 0.2 (0.0) | 12.2 (0.2) | 3.8% |
| | 10 | 475.2 (7.2) | 19.7 (0.3) | 0.4 (0.2) | 0.3 (0.0) | 19.0 (0.3) | 4.1% |
| 4 | 1 | 27.9 (0.2) | 2.0 (0.4) | 0.2 (0.0) | 0.2 (0.2) | 1.7 (0.1) | 7.2% |
| | 2 | 78.6 (7.0) | 3.7 (0.0) | 0.2 (0.0) | 0.1 (0.0) | 3.4 (0.0) | 4.7% |
| | 3 | 109.5 (3.2) | 5.6 (0.4) | 0.3 (0.2) | 0.1 (0.0) | 5.2 (0.1) | 5.1% |
| | 5 | 278.8 (2.9) | 9.7 (0.1) | 0.3 (0.1) | 0.2 (0.0) | 9.3 (0.2) | 3.5% |
| | 7 | 341.7 (9.0) | 14.5 (0.6) | 0.3 (0.0) | 0.2 (0.0) | 14.0 (0.6) | 4.2% |
| | 10 | 525.2 (5.7) | 25.0 (4.5) | 3.7 (4.1) | 0.3 (0.0) | 21.1 (0.8) | 4.8% |

This data was obtained using a Decstation 3100 with 16MB of memory as client, and Decstation 5000/200s with 32MB of memory as servers communicating over an Ethernet. The numbers presented here are mean values from three trials of each experiment. Numbers in parentheses are standard deviations.

Table 7.6: Resolution Time After Work at One Replica

## 7.4 Evaluation of File Resolution

Recall that a file can be resolved by servers only if its replicas have not been modified simultaneously in multiple partitions. To estimate the frequency of this event, the servers were instrumented as described in Section 7.2. The data collected from the servers over a period of 10 months is summarized in Table 7.8. The results confirm that servers could often resolve files – 99.4% of all file resolutions were performed at the servers. 48% of file resolves performed by the servers completed without propagating any file data between the VSG members, while

| Rep Factor | Load | Work Time (seconds) | Resolution Time (seconds) |  |  |  | Res Time / Work Time |
|---|---|---|---|---|---|---|---|
|  |  |  | Total | Phase 1+4 | Phase 2 | Phase 3 |  |
| 2 | 1 | 72.1 (16.6) | 1.7 (0.1) | 0.2 (0.0) | 0.1 (0.0) | 1.4 (0.1) | 2.4% |
|  | 2 | 187.1 (10.5) | 12.7 (1.5) | 0.2 (0.0) | 8.8 (1.2) | 3.8 (1.5) | 6.8% |
|  | 3 | 198.8 (3.9) | 12.0 (1.0) | 0.2 (0.1) | 8.8 (1.2) | 2.9 (0.1) | 6.0% |
|  | 5 | 517.4 (38.6) | 20.5 (2.1) | 0.7 (0.5) | 12.2 (2.0) | 7.6 (0.6) | 4.0% |
|  | 7 | 578.9 (38.1) | 26.5 (1.7) | 0.3 (0.1) | 13.6 (1.2) | 12.5 (2.9) | 4.6% |
|  | 10 | 927.9 (16.8) | 39.5 (2.5) | 11.0 (0.0) | 11.7 (2.4) | 16.9 (0.2) | 4.3% |
| 3 | 1 | 100.5 (12.0) | 2.8 (0.3) | 0.2 (0.0) | 0.1 (0.0) | 2.5 (0.3) | 2.8% |
|  | 2 | 194.0 (4.5) | 10.6 (2.2) | 0.2 (0.0) | 4.2 (3.5) | 6.2 (1.3) | 5.5% |
|  | 3 | 329.9 (10.5) | 16.3 (3.1) | 0.6 (0.6) | 8.2 (3.5) | 7.6 (0.2) | 4.9% |
|  | 5 | 569.1 (16.8) | 30.9 (6.1) | 5.7 (4.5) | 12.2 (2.0) | 13.1 (0.3) | 5.4% |
|  | 7 | 847.0 (75.2) | 45.4 (8.6) | 7.2 (1.4) | 12.9 (4.2) | 25.2 (5.8) | 5.3% |
|  | 10 | 1296.8 (9.9) | 133.3 (13.9) | 17.1 (1.2) | 18.5 (1.7) | 97.7 (13.2) | 10.3% |
| 4 | 1 | 129.2 (15.9) | 7.6 (0.3) | 0.7 (0.7) | 1.5 (2.2) | 5.4 (1.7) | 5.9% |
|  | 2 | 307.1 (32.3) | 26.1 (6.8) | 1.3 (1.0) | 7.6 (6.3) | 17.2 (7.5) | 8.5% |
|  | 3 | 463.7 (37.5) | 38.8 (17.7) | 2.8 (2.8) | 15.7 (2.5) | 20.3 (12.5) | 8.4% |
|  | 5 | 779.6 (67.7) | 43.6 (9.9) | 7.2 (8.3) | 12.2 (2.0) | 24.2 (1.3) | 5.6% |
|  | 7 | 1019.4 (14.8) | 78.4 (29.6) | 17.1 (8.3) | 11.1 (4.2) | 50.2 (21.2) | 7.7% |
|  | 10 | 1837.5 (13.3) | 114.0 (16.7) | 17.6 (18.1) | 18.2(10.0) | 78.3 (12.8) | 6.2% |

This data was obtained from experiments using the same hardware configuration as for Table 7.6. The numbers presented here are the mean values from three trials of each experiment. Numbers in parentheses are standard deviations.

Table 7.7: Resolution Time After Work at All Replicas

46% of them were invoked with at least one empty replica. An empty replica could have been created by either a resolution or a repair of its parent directory. Resolutions not involving the transmission of data are shown in Table 7.8 as resolutions of weakly-equal files, and resolutions involving at least one empty replica appear as runt force resolves.

The rest of this section answers the following two questions:

- What is the latency of file resolution?

| **File Resolutions** | **No. of Cases** |
|---|---:|
| Attempts | 20,370 |
| Successes | 20,237 |
| Weakly Equal | 9,766 |
| Runt Force | 9,338 |
| Normal | 1,133 |
| Needing an ASR | 133 |

This table shows the results of file resolutions observed at the servers over 10 months. Successful resolutions are divided into three categories: *weakly-equal*, i.e. the replicas were equal but their version-vectors were different; *runt-force*, i.e. one or more file replicas were empty; and *normal*, i.e. a subset of the replicas were stale. The last row shows the number of occasions in which a file could not be resolved by the servers.

Table 7.8: Observed File Resolutions

- What is the cost of executing an ASR at the client?

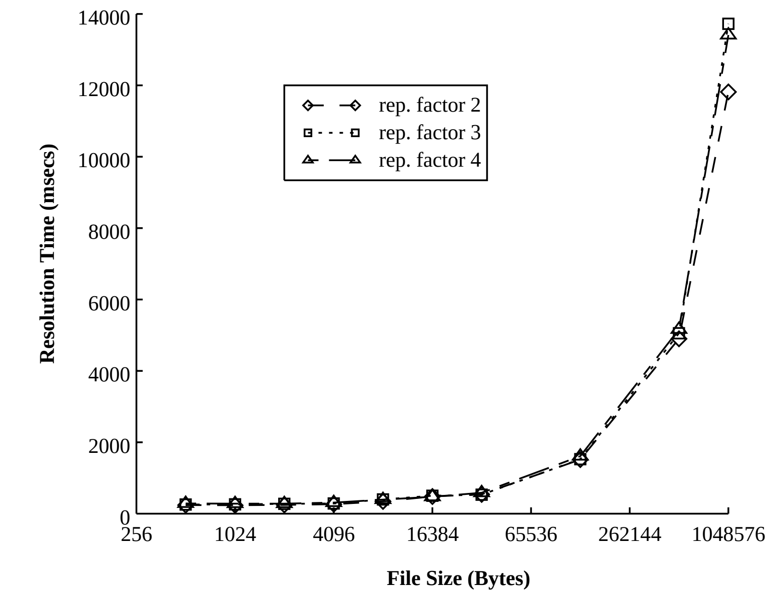
### 7.4.1 Latency of File Resolution

The latency of resolution is apparent to the first user who accesses a file after a partition during which it was modified. Like directory resolution, the resolution time for files is the elapsed time between the invocation of resolution at the coordinator and the return of control to the client.

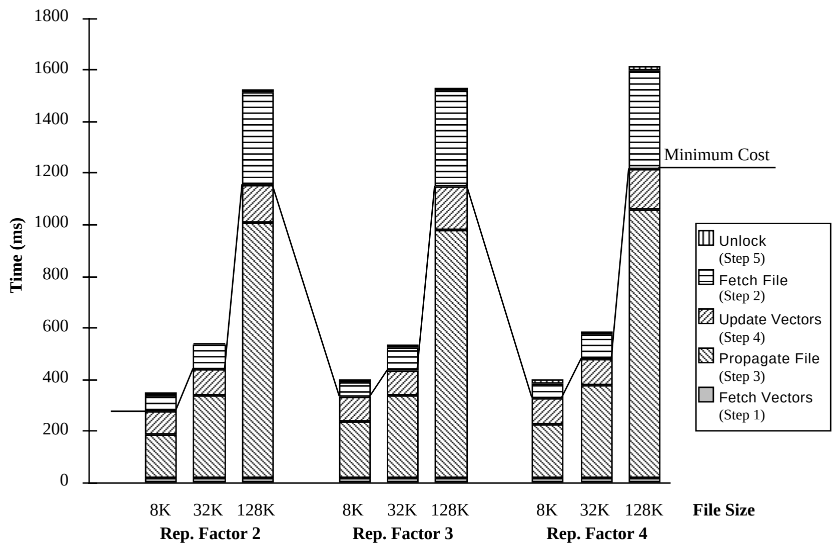
The coordinator of the file resolution protocol performs five steps: lock replicas and fetch version-vectors from subordinates, find and fetch the dominant replica, distribute that replica to all servers, distribute the list of servers that successfully completed the previous step and finally unlock the replicas. The time for resolution is dependent on the size of the file being resolved. An increase in file size lengthens the time needed to complete the second and third steps because more data needs to be transferred over the network. However, the time needed for the other steps remains unchanged. Increasing the number of file replicas lengthens the time for each step because of extra communication overheads.

#### 7.4.1.1 Methodology

A series of experiments were conducted to quantify the above effects. In each experiment, a subset of a file's replicas is modified in a partition. Then the partition is healed and a resolution for that file is triggered. The elapsed time for resolution and each of its sub-steps is measured at the coordinator server using a microsecond timer.



(a) Resolution Time for Varying File Sizes



(b) Execution Time for the Resolution Steps

The data for these graphs is taken from Table 7.9. Graph (a) shows the increase in resolution time with increasing file size and replication factor. Note that the horizontal axis has a logarithmic scale. Graph (b) compares the time spent in the five steps of resolution for three file sizes: 8 Kilobytes, 32 Kilobytes and 128 Kilobytes. The minimum-cost line shows the resolution time with optimizations, i.e. when steps two and five are removed.

Figure 7.5: File Resolution Time

| Rep. Factor | File Size | Resolution Time (milliseconds) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Fetch VV | Fetch File | Distribute File | Update VV | Unlock Objects | Total Time | Growth (msecs/KB) |
| 2 | 0.5K | 14.6 (0.6) | 42.6 (4.7) | 88.3 (6.9) | 86.8 (5.8) | 10.0 (0.5) | 242.5 (12.5) | - |
| | 1K | 14.5 (0.4) | 43.4 (6.5) | 88.4 (7.1) | 82.9 (1.6) | 9.9 (0.4) | 239.3 (6.8) | - |
| | 2K | 14.6 (0.3) | 45.4 (6.2) | 89.9 (7.2) | 87.2 (4.8) | 10.0 (0.4) | 247.3 (14.3) | 8.0 |
| | 4K | 14.3 (0.1) | 52.0 (7.3) | 104.6 (3.1) | 89.2 (3.0) | 10.1 (0.4) | 270.6 (7.0) | 11.7 |
| | 8K | 14.9 (0.3) | 63.4 (3.4) | 170.8 (7.0) | 89.9 (2.3) | 10.0 (0.6) | 349.3 (5.3) | 19.7 |
| | 16K | 14.6 (0.3) | 66.6 (4.3) | 301.6 (6.0) | 93.4 (5.9) | 10.7 (0.9) | 487.1 (8.0) | 17.2 |
| | 32K | 14.5 (0.4) | 92.0 (9.5) | 324.7 (21.5) | 100.0 (3.2) | 10.5 (0.5) | 542.0 (20.6) | 3.4 |
| | 128K | 14.9 (0.5) | 360.2 (16.0) | 993.2 (70.4) | 142.9 (5.0) | 10.1 (0.4) | 1521.5 (63.6) | 10.2 |
| | 512K | 14.5 (0.3) | 979.3 (14.7) | 3721.3 (629.0) | 178.0 (8.1) | 10.6 (0.7) | 4903.9 (640.0) | 8.8 |
| | 1Meg | 13.9 (0.4) | 3300.1 (121.0) | 8239.0 (998.0) | 257.9 (9.1) | 10.7 (0.4) | 11821.8 (1064.7) | 13.5 |
| 3 | 0.5K | 16.8 (0.6) | 47.9 (11.5) | 93.4 (11.0) | 90.5 (2.5) | 11.3 (0.2) | 260.1 (20.1) | - |
| | 1K | 16.6 (0.4) | 40.9 (5.1) | 98.2 (8.5) | 97.0 (12.2) | 11.7 (0.6) | 264.7 (7.8) | 9.2 |
| | 2K | 17.0 (1.1) | 50.4 (10.4) | 102.2 (2.3) | 95.3 (6.2) | 11.8 (0.4) | 277.0 (15.1) | 12.3 |
| | 4K | 16.5 (0.2) | 54.6 (6.2) | 119.0 (4.3) | 88.2 (5.5) | 11.5 (0.2) | 290.2 (7.3) | 6.6 |
| | 8K | 16.8 (0.2) | 59.4 (10.3) | 216.6 (36.1) | 95.9 (2.9) | 11.7 (0.5) | 400.7 (33.2) | 27.6 |
| | 16K | 16.8 (0.2) | 74.9 (7.8) | 301.3 (43.9) | 97.2 (8.6) | 12.7 (0.9) | 506.6 (44.5) | 13.2 |
| | 32K | 16.6 (1.0) | 92.4 (3.1) | 317.3 (47.7) | 96.6 (4.7) | 12.1 (0.7) | 535.3 (51.4) | 1.8 |
| | 128K | 16.9 (0.5) | 372.5 (10.9) | 961.7 (26.6) | 166.4 (32.0) | 11.5 (0.2) | 1529.2 (40.6) | 10.4 |
| | 512K | 16.3 (0.6) | 962.7 (38.5) | 3883.0 (333.0) | 180.7 (8.3) | 11.7 (0.3) | 5054.8 (347.4) | 9.2 |
| | 1Meg | 15.7 (0.2) | 3365.9 (109.5) | 10087.7 (348.0) | 255.2 (6.5) | 11.9 (0.6) | 13724.4 (447.7) | 16.9 |
| 4 | 0.5K | 19.1 (0.9) | 48.5 (9.2) | 106.9 (10.8) | 98.0 (10.6) | 13.3 (0.2) | 286.1 (19.9) | - |
| | 1K | 18.9 (0.2) | 49.4 (4.8) | 108.9 (8.2) | 93.0 (8.2) | 13.1 (0.2) | 283.5 (5.2) | - |
| | 2K | 18.4 (0.5) | 44.6 (4.5) | 114.2 (7.9) | 93.7 (5.5) | 13.2 (0.1) | 284.4 (11.9) | 0.9 |
| | 4K | 18.9 (0.2) | 53.2 (3.9) | 133.8 (3.3) | 94.1 (3.4) | 13.3 (0.2) | 313.6 (3.8) | 14.6 |
| | 8K | 18.7 (0.0) | 59.5 (4.5) | 203.6 (5.3) | 102.0 (5.8) | 13.2 (0.4) | 397.3 (8.2) | 20.9 |
| | 16K | 19.2 (0.7) | 75.9 (11.2) | 263.6 (33.7) | 100.6 (5.2) | 13.3 (0.2) | 473.0 (36.7) | 9.5 |
| | 32K | 19.0 (0.2) | 95.4 (5.0) | 357.8 (48.7) | 100.8 (3.7) | 13.5 (0.4) | 587.5 (50.4) | 7.2 |
| | 128K | 18.8 (0.2) | 385.1 (18.0) | 1036.2 (77.1) | 158.1 (6.0) | 13.2 (0.2) | 1611.8 (74.0) | 10.7 |
| | 512K | 18.5 (1.4) | 947.4 (7.3) | 4005.7 (453.6) | 179.4 (2.7) | 13.5 (0.8) | 5165.2 (456.7) | 9.3 |
| | 1Meg | 17.9 (0.6) | 3324.5 (15.4) | 9782.5 (531.4) | 270.4 (2.8) | 13.1 (0.2) | 13409.2 (526.4) | 16.1 |

This table shows the time spent in each step of resolution for files of different sizes and replication factor. The last column shows the increase in resolution time for each additional KB of data in the file being resolved. The experiments were conducted using Decstation 5000/200s with 32MB of memory as servers communicating over an Ethernet. The time values are in milliseconds and show the mean value from five trials of each experiment. Numbers in parentheses are standard deviations.

Table 7.9: File Resolution Time

To study the effect of increasing file size, a set of ten experiments was conducted. In these experiments, the size of the file being resolved was changed from half a Kilobyte to one Megabyte. To study the effects of replication, each set of experiments was conducted for three replication factors: two, three and four.

#### 7.4.1.2 Results

A study of Unix files on the Internet [24] showed that the average file size is 22 Kilobytes. The same study also reported that the mode and median of the distribution of file sizes is between 1 and 2 Kilobytes. Therefore, from the results in Table 7.9, we can conclude that the resolution time for most files is not noticeable to users. In fact, the resolution time for a file smaller than 32 Kilobytes is less than half a second. As shown in Figure 7.5 (a), the resolution time increases with file size growing to more than 10 seconds for files greater than 1 Megabyte. The growth in resolution time with increasing replication factors is sub-linear – in fact it is less than 20% with the addition of each replica. This is explained by the fact that the coordinator uses multi-RPC [51] to communicate with the group of servers in a file's VSG. Multi-RPC allows a host to communicate with multiple hosts at a cost lower than the total cost to communicate with the same set of hosts individually.

The increase in resolution time for each additional Kilobyte of file data is shown in the last column of Table 7.9. In most cases the increase is less than 20 milliseconds/Kilobyte. This increase is also apparent from the graph in Figure 7.5 (a). Note that the axis showing the file size in this graph has a logarithmic scale. A linear regression analysis of the data in Table 7.9 yielded a very good fit. Using the file size as the independent variable yielded an $R^2$ values of 0.99, 0.98 and 0.99 for replication factor 2, 3 and 4 respectively. In all three cases the regression coefficient for the file size was 0.01, meaning the increase in resolution time for each additional byte in the file is 0.01 milliseconds or 10 milliseconds/Kilobyte.

The graph in Figure 7.5 (a) shows the increase in resolution time with size and replication factor. Note that the horizontal axis, showing the file size, has a logarithmic scale.

Figure 7.5 (b) shows the contribution of each resolution sub-step towards the total resolution time for three file sizes: 8KB, 32KB and 128KB. As expected, the coordinator spends most of its time in steps two and three during which it fetches and distributes the dominant replica. Steps one and five contribute very little to the total resolution time since they merely do locking and unlocking. The results in Table 7.9 confirm that the time taken by these two steps is independent of the size of the file being resolved. However, contrary to our expectation the execution time for step four is significant and not independent of the file size. This anomaly occurs due to an artifact of our implementation. The coordinator fetches the dominant replica's contents into a temporary file on the local disk. This file is deleted only after its contents have been distributed to all servers. The time to delete a file increases with its size because more disk blocks need to be reclaimed. Since the temporary file is deleted in step four, the execution time of this step also increases with file size.

The performance of file resolution can be improved with two optimizations. The unlock step can be eliminated altogether by having the subordinates unlock all objects at the end of the fourth step. The fetch-file step can be eliminated by choosing the server with the dominant replica as the coordinator of the protocol. The improvement in performance with these optimizations is shown in Figure 7.5 (b).

### 7.4.2 Overhead of the ASR Mechanism

An ASR is invoked transparently when a user requests service for a file with diverging replicas. Since the user's request is suspended while the ASR is executing, the ASR's execution time is apparent to the user as a higher latency in the servicing of her request.

Two factors contribute towards the increase in latency for servicing requests – the time needed to find and invoke the ASR and the time needed to execute the ASR. While the former contributor is independent of the application and can be measured by experimentation the latter contributor is application dependent and can only be estimated by empirical measurements. Due to our limited experience with ASRs, this section only presents measurements of the former contributor. In fact, this contributor is a measure of the minimum latency seen by a user making a system call that triggers an ASR.

The time needed to find an ASR depends on the location of the `ResolveFile` with respect to the file being resolved. Since the ASR-`starter` traverses the file's ancestors until it finds the `ResolveFile`, a longer distance between these files lengthens the time needed to find the ASR. Furthermore, a longer `ResolveFile` increases the time to start an ASR since the `starter` must do more work to parse the file and find the matching rule. The time needed to start an ASR includes the time to `fork` processes for each command listed in the resolution rule. Obviously, a bigger command-list also increases the execution time of the ASR.

#### 7.4.2.1 Methodology

A series of experiments were conducted to measure the above effects. In each experiment an ASR was triggered by `stat`-ing a file with diverging replicas. The execution time of the ASR was measured by Venus – it is the elapsed time between Venus requesting the ASR and the `starter` informing Venus that the ASR execution has completed. The end-to-end latency of the `stat` system call was also recorded since this is the latency apparent to the user. In both cases, a microsecond timer was used to get an accurate measurement of the elapsed time.

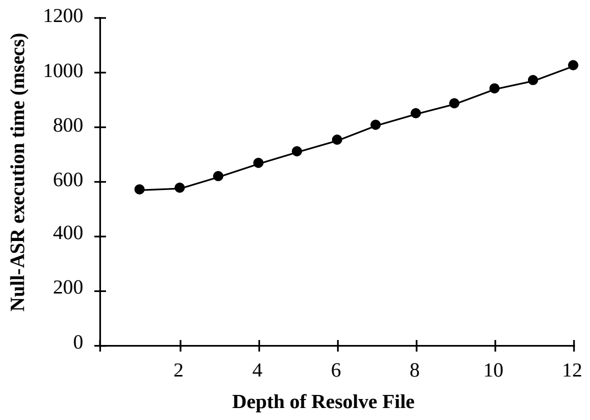
To study the effect of changing the *depth* of the `ResolveFile`, i.e. the distance between the `ResolveFile` and the file being resolved, the above experiment was conducted with depth levels from 1 to 12. A depth-level of $n$ implies the ASR-`starter` had to lookup $n$ ancestral directories to find the `ResolveFile`.

In order to minimize the effect of the other parameters that lengthen the ASR invocation time, the experiments were conducted with a `ResolveFile` containing only one resolution-rule

Resolution rule used for the experiments:

```
*:
    /bin/echo -n
```

| Depth of `ResolveFile` | ASR Exec. Time | | End-to-End Latency | |
|---|---|---|---|---|
| 1 | 571.5 | (4.9) | 625.7 | (5.5) |
| 2 | 576.6 | (4.5) | 631.3 | (4.6) |
| 3 | 620.8 | (5.2) | 675.5 | (5.2) |
| 4 | 665.0 | (3.2) | 720.6 | (4.2) |
| 5 | 708.5 | (5.2) | 765.6 | (5.6) |
| 6 | 754.1 | (13.9) | 808.0 | (14.0) |
| 7 | 806.5 | (10.9) | 863.9 | (11.9) |
| 8 | 847.1 | (4.3) | 904.3 | (4.4) |
| 9 | 888.9 | (4.0) | 945.6 | (4.5) |
| 10 | 940.7 | (16.7) | 997.9 | (16.5) |
| 11 | 970.2 | (5.3) | 1028.1 | (7.2) |
| 12 | 1026.4 | (6.8) | 1084.6 | (6.5) |



The table shows the execution time of the ASR and the elapsed time for the `stat` system call while the graph shows only the former. The file being resolved was replicated at two servers. The experiments were performed on a Decstation 5000/200 with 32 Megabytes of memory. The time values are in milliseconds and show the mean value from nine trials of each experiment. Numbers in parentheses are standard deviations.

Figure 7.6: Execution Time for a Null ASR

with one command. In order to factor out the application-dependent overhead, i.e. the time needed to resolve the contents of the file's replicas, the Unix program `echo` was used as the null ASR. Instead of using an empty command-list we chose to use a null ASR in order to include the cost of `fork`ing a process in the measurements. Note that at least one call to `fork` is necessary during the execution of any ASR. Therefore, our measurements provide an accurate lower bound on the minimum latency for an ASR invocation.

#### 7.4.2.2 Results

The results of our experiments, shown in Figure 7.6, confirm that the framework for executing an ASR has a small overhead. In most cases, it takes between one-half and one second for the ASR to be invoked and its results returned to Venus. The system call overhead to process the `stat` call is about 55 milliseconds. Obviously, these measurements exclude the time needed to perform the resolution.

The graph in Figure 7.6 shows the increase in overhead for the ASR-invocation framework as the depth of the `ResolveFile` is changed. A linear regression analysis on the data yielded a very good fit. Using the depth of the directory as the independent variable yielded an $R^2$ value of 1.0. The regression coefficient for the depth was 44.80, meaning the overhead for looking up an ancestral directory for a `ResolveFile` increases the latency of the ASR by 44.80 milliseconds.

The minimum overhead for the framework for invoking an ASR is 571.5 milliseconds. In this experiment the depth of the `ResolveFile` is 1, meaning the `ResolveFile` and the file being resolved are in the same directory. More detailed measurements of this experiment configuration show that there is a 12.5 millisecond delay between the time Venus requests an ASR and the `starter` receives the request. The `starter` `fork`s an `executor` which causes a delay of 50 milliseconds. The `executor` takes 496 milliseconds to perform its work – 180 milliseconds to parse the `ResolveFile` and 316 milliseconds to lookup the parent directory for the `ResolveFile`, lock the volume and `fork` the command from the resolution rule. Finally there is an additional 12.5 millisecond delay in the `Result_Of_ASR` RPC from the `starter` to Venus.

Although the minimum latency for executing an ASR may seem high, it is much lower than the time needed to manually invoke an ASR or perform the resolution. Furthermore, the convenience of automating this task far outweighs the pain of manual resolution. Finally, as discussed in Chapter 5, a significant fraction of this cost is due to the need to provide flexibility, security and robustness for the ASR mechanism.

## 7.5 Evaluation of Manual Resolution

Previous measurements from AFS [25] showed that conflicting updates to files and directories are rare in a Unix environment. This pattern is exhibited by users of the Coda file system also – the high success rate of automated resolution supports this claim. However conflicts do occur in Coda and the affected objects must be manually repaired. A repair facility that is easy to use and automates most of the task is necessary to make the system usable. In the absence of an effective repair facility, a user may stop using the system during a failure to avoid repairing objects when the failure ends. In fact, this behavior was common in the initial stages of the project when the system had no means of resolving directories and the repair facility did not assist users in repairing objects.

This section presents empirical results showing the effectiveness of Coda's repair facility. It addresses three fundamental questions:

- How often is a file or directory repair needed and how often does it succeed?
- How effective is the repair facility in assisting a directory repair?
- What strategies do users employ to repair files?

To answer these questions, repair statistics were collected for a period of one year. During this period, the repair facility was used by at least sixteen users. Since the users were completely unaware of the fact that their repair session was being recorded, the results in this section are realistic.

### 7.5.1 Methodology

The file and directory repair tools were instrumented to collect a statistics record for each repair session. Each record contained the name of the object being repaired and the result returned by each server for the `ViceRepair` RPC. The object's name was used to classify the repair according to the type of object being repaired while the error code was used to classify it as a successful or failed repair. In addition to these two fields, the record for a file repair also recorded the user's choice for performing the repair. Recall from Section 6.2.3 that a user has four options to repair a file: remove it, use a new file, use one of the file's replicas or explode the file into a directory. The user's choice along with the name of the repaired file provides enough information to deduce the preferred repair strategy for each file type.

Recall that the directory repair tool assists the user in creating the repair-file. However, due to security and correctness considerations the tool cannot generate a repair-file in certain situations. For example, if a directory's replicas have conflicting updates to their access-control lists, the user must decide the corrective actions. Anecdotal evidence from our environment indicates that most users dislike editing the repair-file and would prefer the repair tool to generate the repair-file under all circumstances.

To measure the effectiveness of the directory repair tool and to understand the nature of directory repairs, two additional pieces of information were recorded in the statistics record: first, the frequency with which users repaired a directory without changing the repair-file generated automatically by the tool; and second, the contents of the repair-file used to perform each repair. The first statistic provides an exact count of the instances in which the tool's assistance was adequate. If a user modified the repair-file, he/she was asked to grade the difficulty of making the modifications.

The statistics record for each repair session was stored in a separate file that was named uniquely using the identity of the user initiating the corresponding repair session. Thus, statistics corresponding to all repairs initiated by a user are easily identifiable. To simplify the collection of data, the statistics files were collected in a shared directory in a Coda volume. All users had access rights to add files to this directory but not to delete or modify any of the files. A disadvantage of this approach is that some records may have been lost if the client initiating the repair could not access the directory storing the statistics files. However, the loss of data was minimized by replicating this directory across the same set of servers that store most user's volumes.

### 7.5.2 Results

The summary of repair statistics collected from July 1993 to July 1994 are shown in Table 7.10. This section discusses answers to each of the questions posed above.

**Frequency and success of repairs** As shown in Table 7.10 (c), repairs are infrequent and succeed more than 80% of the time. On the average a user repairs one file and two directories every month. Even though the Coda community consists of over 30 users, only 16 of these users needed to repair a file or directory during the one year period. Objects that belong to the remaining users were either always resolved automatically or never updated during a network partition. As expected, the number of directory conflicts was much higher than the number of file conflicts. In the one year period, 320 directories and 157 files were repaired. 83% of directory repair requests and 87% of file repair requests completed successfully. A user attempting a repair with expired or no authentication tokens was the leading cause of repair failures. Semantic errors in the repair-file caused some failures during directory-repairs. These errors were introduced by users modifying the repair-file generated by the tool.

**File repair strategies** The names of repaired files were used to classify them into five groups: editor backup and checkpoint files, object files produced by compilers, files containing source code, files produced by $\LaTeX$ and Scribe, and `.history` files written by `csh`. Files that could not be classified into any of these groups were placed in the unrecognizable file category. The number of files in each category is shown in Table 7.10 (a). Note that almost 50% of the files fall into the unrecognizable category. During the first six months of instrumentation, the name of the file being repaired was not recorded. Therefore, all files repaired in that period of time are classified into the unrecognizable file category.

| **Kind of File** | Repair Strategy: Remove All Replicas | Repair Strategy: Use one of the Replicas | Repair Strategy: Use a new File | Total |
|---|---|---|---|---|
| Backup&Checkpoint | 6 | 2 | 0 | 8 |
| Object | 20 | 14 | 0 | 34 |
| Source | 3 | 17 | 1 | 21 |
| Word-processing | 7 | 9 | 0 | 16 |
| `.history` | 1 | 0 | 0 | 1 |
| Unrecognizable | 17 | 40 | 20 | 77 |
| **Total** | 54 | 82 | 21 | 157 |

(a) File Repairs

| | | **No. of Cases** |
|---|---|---|
| **Repair file contains:** | | |
| No operations | | 83 |
| Renames | | 43 |
| Access-control list ops. | | 31 |
| Set ownership ops. | | 15 |
| **Repair-file creation:** | | |
| Automatic (by tool) | | 254 |
| Edited/created by user | | 66 |
| **Difficulty in editing:** | | |
| Easy | Level 0 | 22 |
| ↓ | Level 1 | 15 |
| | Level 2 | 12 |
| | Level 3 | 9 |
| | Level 4 | 4 |
| Hard | Level 5 | 4 |

(b) Directory Repairs

| | **Files** | **Directories** |
|---|---|---|
| Attempts | 157 | 320 |
| Successes | 136 | 265 |
| Failures | 21 | 55 |
| No. of Users | 14 | 16 |

(c) Summary of All Repairs

This data was collected over a period of one year, from July 1993 to July 1994. Table (a) shows the number of times each file repair strategy was used for the various file types. The totals for each strategy and file type are also shown. Table (b) shows statistics for directory repairs. It lists the frequency with which some operations occured in repair-files. It also shows the number of times the repair-file generated by the tool was insufficient to perform the repair. The users' rating of the difficulty in creating/modifying the repair-file is also shown. Table (c) provides a summary of all repairs performed.

Table 7.10: Empirical Data Collected for Repairs

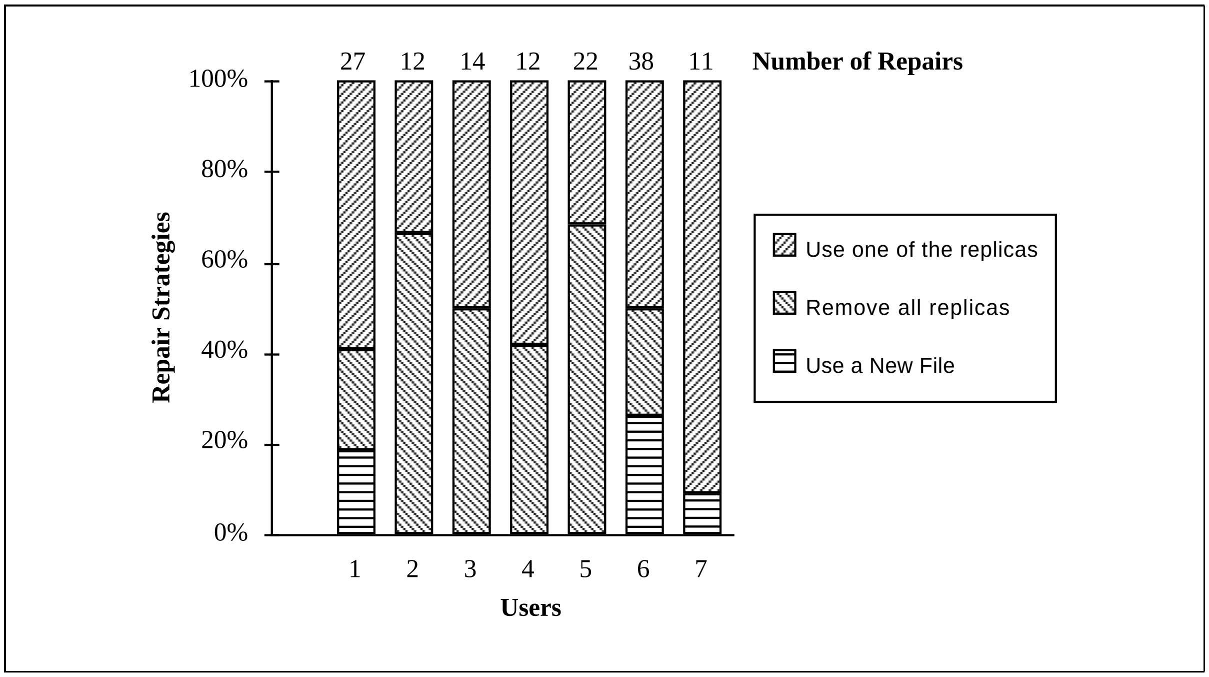
In our environment, object files suffered the highest number of conflicts followed by source code files. Object files exhibit the highest degree of "write-sharing" because users often use multiple clients as compile engines. Only one `.history` file had a conflicting update. This is an expected result because the `.history` file is stored in a user's home directory and most Coda users have their home directory in the Coda file system at only one client, their laptop computer.

Users used three of the four options to repair files. The most popular file repair option, chosen 52% of the time, was using one of the file's replicas to replace all its copies. The options to delete all replicas of a file and to use a new file were chosen 34% and 14% of the time, respectively. Users never chose to explode a file into a replicated directory, probably because they did not understand its semantics. A user will be more inclined to choose this option if the file being repaired is truly shared with other users and the actions needed to repair the conflicting updates are not clear.

There were few surprises in the preferred repair strategy for each file type. Deleting all replicas was the preferred strategy for object files while using one of the replicas was the preferred strategy for source files. This was expected since object files can be recreated more easily than source files. The repair policy for files in the word-processing category was evenly divided between these two strategies. This dichotomy is due to the inclusion of two kinds of files in this category – user text files that are not reproducible and files generated by the word-processing system that can be regenerated easily. The preferred repair action for backup and checkpoint files was to delete their replicas since they are useful only if the master copy of the file is deleted.

Does a user prefer to use one file repair strategy over the others? To answer this question we examined each user's choice for repair strategy. Figure 7.7 shows the frequency with which each repair strategy was chosen by seven users who repaired at least ten files. Only one of the users, User 7, exhibited a strong preference for using one of the replicas for the repair. User 1 and User 4 used this strategy 60% of the time while User 2 and User 5 deleted all replicas of the file 70% of the time. We cannot conclude that users prefer a single repair strategy for two reasons: first, the number of data samples is small; second, none of the users who performed a significant number of repairs, chose to use a single strategy for all repairs.

**Effectiveness of the directory repair tool** Our measurements show that the tool was very effective in automating directory repairs. As shown in Table 7.10 (b), the automatically generated repair-file was successful in repairing 79% of the conflicts. Contrary to our expectations based on anecdotal evidence, the measurements show that users did not find the task of editing the repair-file to be very difficult. On a scale from 0 to 5, they rated the difficulty at a level of 2 or less 74% of the time. It is possible that this anomaly is caused by users being over cautious while rating the difficulty.



The graph shows file repair strategies chosen by 7 users. Each of these users performed at least 10 file repairs. The number of repairs performed by each user was scaled to 100 to simplify comparing the user's preferences. Numbers at the top of the figure show the exact count of files repaired by each user.

Figure 7.7: File Repair Strategies

Analysis of the repair-files shows that out of 66 repair-files modified by users, 31 contained commands to modify an access-control list and 15 contained commands to change directory ownership. Since these commands are not generated automatically by the tool's `CompareDirReplicas` command, the user was forced to edit the repair-file in these instances. In the remaining twenty instances the user voluntarily edited the repair-file.

On 83 occasions the repair-file contained no operations for the servers, i.e. the directory being repaired had identical replicas. This situation arises if the directory replica with conflicting updates is not accessible during the repair or the directory being repaired participated in a partitioned rename. Consider an object being renamed between two directories during a partition. When the partition ends, resolution marks both directories in conflict. If the rename was the only conflicting update, then repairing one directory automatically makes the replicas of both directories identical. Therefore, the repair of the other directory will not require any operation.

## 7.6 Summary of Chapter

The effectiveness of the resolution techniques developed in this thesis can be estimated by answering the following questions:

- How often do the automated facilities succeed in practice?
- What is the latency of automated resolution?
- What are the overheads for providing these facilities?
- How effective is the repair facility?

These questions are answered by using empirical and quantitative measurements from the Coda system.

Empirical measurements show that automatic resolution works very well in practice. It completes its task successfully 99% of the time. Quantitative measurements show that the automatic resolution process has excellent performance and is rarely noticeable in normal operation. The time and space overheads for making the system capable of automatic resolution are unnoticeable. The repair facilities, although used rarely in practice, are effective.

# Chapter 8

# Related Work

This chapter describes work related to Coda's resolution mechanisms. It is divided into four sections. The first three sections describe work done in the context of other file systems for automatic and manual resolution of directories and files. The last section describes resolution techniques developed in the context of optimistically replicated databases.

## 8.1 Automatic Directory Resolution

Automatic resolution of partitioned updates to directories has been explored by two file systems besides Coda: Locus [53] and Ficus [20]. Chronologically, Locus predates Coda by many years. On the other hand Ficus, a descendant of Locus, was developed contemporaneously with Coda. This section compares the Locus/Ficus approach for automatic directory resolution to Coda's log-based approach.

### 8.1.1 Locus

The Locus system developed at UCLA was the first system to recognize the potential for using optimistic replication in Unix file systems. In fact Coda's replica control policy is influenced by Locus' strategy. Locus file system objects were grouped and replicated at multiple hosts; and updates were allowed in any partition as long as at least one replica was accessible. Thus the availability offered by Locus and Coda is comparable.

A fundamental difference between Coda and Locus arises from their structural organization. Unlike Coda that uses a client-server model, Locus used a peer-to-peer model. In other words Locus did not make any distinction between hosts that stored objects and hosts that were used to access these objects. Abandoning the use of a client-server architecture seriously limits the system's scalability [45]. As a result Coda is much more scalable than Locus.

Another difference between Coda and Locus lies in the mechanism they use for concurrency control. Locus used a pessimistic primary replica strategy that strictly serialized all updates within a partition. Coda on the other hand uses an optimistic strategy for this purpose. If updates are made concurrently, then only one of the updates is allowed to complete successfully.

The Locus system recognized the possibility of using Unix directory semantics to automatically resolve partitioned updates to directories [43]. It used a version-vector method to detect diverging replicas of a directory and proposed reconciling them by inferring the partitioned updates from the state of each replica. It recognized the difficulty in disambiguating partitioned removes from creates but did not provide a viable solution. In fact, the biggest shortcoming of this work is that none of the algorithms proposed for automatic resolution were implemented successfully.

### 8.1.2 Ficus

The Ficus distributed file system, a descendant of Locus, is built on top of NFS [44]. It inherits Locus' optimistic replica control strategy and its peer-to-peer structure. As a result, it preserves Locus' strength and weakness – its high availability and limited scalability.

As in Coda, Ficus objects are grouped into volumes that form the unit of replication. However the two system use different strategies to keep the replicas consistent. In Coda updates are propagated to the replicas in parallel by the client; and resolution is used occasionally to merge updates only after a failure. Ficus on the other hand uses resolution frequently, even in the absence of failures, to distribute an object's updates to its storage sites. For performance reasons, an update to a Ficus object is made to a single closest replica of the object. Then the peer hosts that store other replicas of this object use resolution to pull the update asynchronously at their convenience. Therefore Ficus hosts must perform resolution periodically to prevent the replicas from diverging. This is in contrast to the Coda approach where resolution is needed only to handle the exception case, i.e. when updates are made during a failure, and thus can be performed lazily.

There are two problems with the Ficus approach. First, it requires frequent resolution between the hosts to ensure that no updates are missed by any host. Frequent polling between hosts affects the system's scalability. More seriously, it does not prevent hosts from handing out stale data even in the absence of failures: an update that arrives at a host soon after a round of resolution remains unnoticed at other hosts until the next round of resolution; in the mean time these hosts could service user requests using the stale version of the object. Therefore unlike Coda, Ficus cannot guarantee that an update is immediately visible to all hosts in the same partition. A second problem with the Ficus method is its susceptibility to recovery storms. A host recovering from a failure must contact all its peers and pull updates it has missed. This approach introduces a high demand on the system and may result in cascading failures.

The directory resolution algorithm in Ficus uses inferential techniques rather than the log-based approach used in Coda. In other words partitioned updates in Ficus are computed by comparing the state of the directory's replicas. While Coda's resolution techniques use simple syntactic methods like comparing a list of log entries, Ficus' resolution techniques are complex and are closely tied to the semantics of the directory's contents and its updates. Not only is the latter method conceptually harder but its performance depends on the size of the directory being resolved, not the amount of partitioned activity. This inefficiency can seriously affect Ficus' scalability especially because resolution is a frequent event in that system.

Like Coda, Ficus preserves information about deleted objects in order to disambiguate creates from deletes. However the systems differ markedly in their approach to reclaiming space pertaining to these objects. Ficus uses a complex two phase distributed garbage collection algorithm whose scalability is open to question. Coda, in contrast, uses the much simpler strategy of allowing each site to unilaterally reclaim resources via log wrap-around. This provides a clearly-defined trade-off between usability and resource usage, a trade-off that is essential in any practical system.

The Ficus literature has failed to report any quantitative and empirical measurements of the directory resolution techniques it implements; nor does it make any mention of a user community. The lack of measurements of the directory resolution system makes it impossible to compare it with Coda's performance, scalability and usability.

## 8.2 Automatic File Resolution

Coda uses two orthogonal mechanisms for file resolution – version vectors and application-specific resolvers. This section summarizes related work for each of these mechanisms.

### 8.2.1 Version Vectors

The use of version vectors to detect write-write conflicts between an object's replicas was first proposed by Locus [39]. Locus' descendant system, Ficus, also uses the version vector mechanism for the same purpose.

The use of version vectors in Coda is inspired by their use in the Locus system. In fact, Coda version vectors are conceptually similar to Locus version vectors. However, they differ in structure. Unlike a Coda version vector that only stores version numbers, a Locus version vector also stores the identity of the host corresponding to each version number in the vector. Although storing this additional information requires more space, it simplifies the process of dynamically adding or removing a host from the set of replication sites for an object. This functionality would be useful in a production system where a storage site may be shutdown permanently due to hardware errors or a new site may be added to improve availability or throughput. Lack of this functionality makes Coda version vectors less flexible than Locus version vectors.

### 8.2.2 Application-specific Resolvers

Coda was the first file system to provide a framework for invoking application-specific resolvers transparently. Since the first publication of Coda's approach to application-specific resolution [26] a conceptually similar mechanism has been described for resolving diverging replicas of files in the Ficus system [42]. Coda's framework for application-specific resolution is also conceptually similar to the watchdog mechanism proposed by Bershad and Pinkerton [4] – both these mechanisms extend the semantics of the file system for specific files. This section summarizes Ficus' approach for invoking ASRs and highlights the differences between watchdogs and ASRs.

#### 8.2.2.1 Ficus

Like Coda, Ficus invokes resolvers transparently if and when the need arises. Although Ficus uses a rule-based approach for selecting a resolver, it provides much less flexibility than Coda. For example, Ficus assumes that a resolver will resolve exactly two file replicas and therefore uses a fixed format for the parameter list of the resolver. Ficus resolution rules can be specified only in two files – a system wide rule file and a user specific personal resolver file. Furthermore it is not possible to specify that a group of programs are to be executed as one logical resolver, nor can a group of files be resolved together.

There are also important differences in the execution models of resolvers in Ficus and Coda. Since Ficus uses a peer-to-peer rather than a client-server model, it is more liberal in its choice of execution site – any site with a replica of a file can run a resolver for it. If one resolver fails, others are tried in succession.

The Ficus design pays less attention to issues of security and robustness. Ficus resolvers are run on behalf of the owner of the file and not the user accessing it. Therefore, a malicious user could cause serious damage by using a misbehaved resolver on a file owned by another user. There are no specific mechanisms in Ficus to provide atomicity or isolation. Coda, in contrast, takes these issues much more seriously and provides specific mechanisms to improve safety.

#### 8.2.2.2 Watchdogs

A watchdog [4] is an extension to the file system that allows users to design and implement their own semantics for file system objects. Like an ASR, a watchdog is implemented as a user level process and its functionality is invoked transparently. However the functionality provided by these two mechanisms is very different. While an ASR is used only to resolve diverging replicas of a file, a watchdog is more general. It can be used to change the functionality of existing system calls or provide new functionality for files and directories.

Since the watchdog mechanism is not specifically intended for resolving partitioned updates, it does not incorporate many important mechanisms needed by ASRs. For example, it does not need to provide a mechanism for exposing file replicas, or pay particular attention to the issues of isolation and atomicity. Instead of using rules to select a watchdog, it relies on a system call to link a watchdog to a file or directory. Therefore it is not as flexible as Coda's rule-based approach. The execution of a watchdog is managed by a special process called the *chief* watchdog which serves the same purpose as the ASR-`starter` in Coda. Unlike an ASR that is transitory, i.e. a new process is started every time a resolution is needed, a watchdog can remain active for a longer period of time and service multiple requests.

## 8.3 Manual Resolution

Very few distributed file systems have implemented general purpose tools to help the user in resolving files or directories. The need for such tools was recognized by some file systems like Saguaro [40, 41] and Locus. But neither of these systems provided such tools to their users.

The Locus system was responsible for providing a taxonomy of conflict types for directories. It classified all directory conflicts into three distinct classes: name/name, remove/update and update/update conflicts [18]. Conflicts involving renames did not appear in this list because Locus treated a rename operation as a `link` followed by an `unlink` operation. Coda's repair tool uses this taxonomy, appropriately extended for the Coda environment, to correctly infer the conflicts for a directory with diverging replicas. Although Locus recognized that replicas needed to be preserved as evidence and exposed to the user during the process of manual resolution, it did not implement any facility that provided these features.

The literature describing Locus' descendant system, Ficus, provides little information about its manual resolution facility. However one can assume that Ficus must provide this facility since it is being used by a small group of users. The sketchy information available in the literature indicates four significant differences between the Coda and Ficus manual resolution facilities. First, Ficus makes use of a special *orphanage* directory to store diverging replicas of an object. Doing so results in some loss of contextual information about the cause of the conflict and is in contrast to Coda's strategy of preserving an object in the directory it existed in when its replicas diverged. A second difference arises in the method used to notify users about conflicts. Ficus uses electronic mail to notify the owner of an object with conflicting updates immediately after the conflict is detected. Coda on the other hand uses a lazy notification strategy – a user becomes cognizant of a conflict only when he tries to access an object with conflicting updates. The third difference is that Ficus does not provide any facility that compares directory replicas and suggests ways to repair the conflict. Finally, none of the Ficus literature indicates whether the system addresses issues related to fault-tolerance of the manual resolution sessions.

### 8.3.1 Commercial Packages

In the past two years some file-synchronization programs have become available for the Apple Macintosh and PC DOS environment. The goal of these programs is to keep the file system of two or more computers synchronized. In some sense, this functionality is the same as that provided by directory resolution in Coda. However, unlike directory resolution that is invoked transparently, these programs require the user to invoke them manually. Moreover, these programs do not pay any attention to issues such as scalability, security and performance.

In comparison to Coda's manual resolution facility, the biggest shortcoming of these programs is that they are not integrated with the file system. Therefore, they must rely solely on user level attribute information to perform resolution. The problem is that this information is insufficient to detect certain kinds of conflicts and can sometimes be deceptive. For example, different files could be mistaken to be identical if they have the same name, length and time-stamp; or a newer version of a file could appear to be older just because some program revised the older version's time stamp without changing its data contents. The Coda repair tools do not suffer from these problems because they have complete access to all file system information.

Non-commercial programs like Howard's RECONCILE facility [23], and commercial programs like MBS Technologies' FileRunner, Traveling Software's LapLink and Symantec's Norton Essentials are a few examples of file-synchronization programs. Users of these programs can select specific files or entire sub-trees that need to be monitored. The communication of information between the two computers is typically via a floppy disk.

Some of these programs, for example, FileRunner and RECONCILE, have been influenced by Coda's log-based resolution strategy. Since these programs are not integrated with the file system they cannot log updates as they occur. However they resort to a scheme that uses logging partially. When a user specifies the directories that need to be managed by the synchronizer, they build a journal reflecting the initial state of each directory. This journal and the final state of the directories is used during resolution to compute the updates made at each directory. Then the updates are resolved using their semantics.

## 8.4 Optimistic Replication in Databases

The use of optimistic replication for high availability in database systems has been explored by a number of researchers in the 1980s. In particular, three pieces of work by Davidson [9, 10], Blaustein et al [5, 6] and Garcia-Molina [14] have some relevance to Coda's resolution mechanisms and are described below.

### 8.4.1 Davidson

Davidson provided the ground breaking work on optimistic replication methods for databases [8]. Her work concerns the resolution of an optimistically replicated database that has executed transactions concurrently in two or more partitions. The database maintains a log of transactions committed in each partition and the resolution algorithm uses these logs to construct a partial order of the transactions. The data structure used to compute the partial order is called a *precedence graph* whose nodes represent transactions and whose edges represent *reads-from* or *conflicts-with* relationships between transactions. Transactions conflict if and only if the precedence graph has cycles. The conflict is resolved by undoing one of the transactions in the cycle and redoing it after all the other transactions. Davidson proved the correctness of this algorithm by showing it satisfies one-copy serializability.

The most striking similarities between Davidson's and Coda's approach are the use of logging for resolution and the use of serializability theory for proving correctness of the resolution system. The biggest difference between the two approaches is the way in which they resolve conflicting transactions. While Davidson can undo the conflicting transactions and redo them in the order determined by the precedence graph, Coda cannot undo the transactions since the computation associated with the inferred transaction is not known. Therefore Coda must resort to user assistance for resolving conflicts. The other major difference between the two approaches lies in their method for detecting conflicts. While Davidson uses a precedence graph, Coda uses a combination of value and version certification techniques along with the temporal ordering of intra-partition transactions to decide if the transaction histories are serializable.

There are also other minor differences between the two methods. For example, Davidson's method resolves database replicas pair-wise at the coordinator site. Coda, on the other hand, uses a distributed protocol to resolve all replicas in parallel. Coda's method uses more network bandwidth than Davidson's method but has a lower latency since the transaction logs are processed in parallel. Another difference between the two methods is that Davidson does not pay any attention to practical issues such as scalability, performance and security. This may be because her work was purely conceptual in nature and there is no indication in the literature that the concepts were ever implemented.

### 8.4.2 Blaustein

Blaustein et al [5] analyzed two fundamental resolution techniques: inferential and log-based. They recognized that inferential techniques are very closely tied to the semantics of the data being resolved and can be difficult to implement in the presence of conflicts. Log-based techniques, in their opinion, are not as *ad hoc* as inferential techniques, but are less frugal in using space and network bandwidth. A theoretical analysis of the use of transaction logs to merge updates from multiple hosts is provided by Blaustein and Kaufman [6].

Most of this theoretical work was done in the context of a distributed database model and does not apply directly to Unix file systems. This is because the file system and database models are in direct contrast to one another – while databases exhibit frequent updates and a high degree of write-sharing, Unix file systems show completely opposite characteristics. As shown by this dissertation, a log-based strategy works well in a practical implementation of a Unix file system.

### 8.4.3 Data-Patch

Garcia-Molina et al [14] proposed a special tool called Data-patch for repairing conflicts in replicated databases. From the literature available it appears that this design was not implemented or used. Data-patch was designed to be used by a database administrator to develop a program to automatically resolve diverging replicas of the database. The input to the tool is a database schema and the rules describing the schema; as output it generates the program that performs the resolution. At a very high level this idea is similar to the assistance provided by Coda's directory repair tool: it uses the semantic rules of the directory structure and the state of the diverging replicas to produce a list of commands whose execution results in making the replicas equal. The major shortcoming of this work is it makes impractical assumptions. For example, it assumes that partitioned transactions never delete objects and that partitions do not occur during a resolution. These are not viable assumptions in a real system.

# Chapter 9

# Conclusion

Optimistic replication techniques can mask many failures and substantially improve data availability in distributed file systems. However, the danger of conflicting updates to partitioned replicas, combined with the fear that the machinery needed to cope with conflicts might be excessively complex, has prevented designers from using optimistic replication in real systems.

This thesis puts such fears to rest by designing and implementing the mechanisms needed to cope with the consequences of optimistic replication in a real distributed file system. It uses simple yet novel techniques to perform automatic resolution and shows that these techniques can improve usability without a significant impact on the system's scalability, security or performance.

Coda is the first distributed file system with a significant user community to demonstrate the practicality of optimistic replication. It has been in daily use by 35 users for more than 3 years, thereby providing the first opportunity to report on significant usage experience with such systems. Measurements from the system show that it provides significantly higher availability than its predecessor system, AFS, without compromising performance or security. The system has maintained usability by automatically resolving partitioned updates on more than 99% of the attempts.

## 9.1 Contributions of the Dissertation

The main contribution of the thesis is validating – through the design, implementation and evaluation of a system – the fact that optimistic replication can be used effectively in distributed file systems. More specifically, the contributions are in four areas:

1. Design contributions:

   - Architectural design of a system that can cope with the shortcomings of optimistic replication.
   - Detailed design of resolution strategies:
     - A directory resolution protocol that performs its task using the log of partitioned updates.
     - A file resolution framework for invoking arbitrary application-specific resolvers that combines transparency and flexibility with robustness and security.
   - Formal specification of the Unix file system interface that permits reasoning about correctness. Use of this formalism, to specify the goals of the resolution algorithm and prove its correctness.

2. Working implementation in a real system under daily use.
   - Demonstration of the feasibility and effectiveness of log-based directory resolution.
   - Implementation of a prototype ASR framework and example resolvers.
   - Demonstration of fault-tolerance of both these resolution methods.
   - Implementation of a repair tool that is user-friendly and simplifies manual resolution.

3. Empirical measurements to validate the following claims:
   - Automated resolution techniques work well in practice.
   - Genuine conflicts are rare in the Unix environment.
   - Logging mutations at the server does not use excessive storage.
   - The repair tool is effective when needed.

4. Quantitative analysis based on controlled experimentation:
   - Use of file system traces to analyze the growth in resolution logs for long partitions.
   - Measurements showing that the overhead for resolution is minimal.
   - Evidence proving the latency of resolution is not noticeable.

## 9.2 Future Work

Each of the three parts of this thesis, namely automated resolution of directories, automated resolution of files and manual resolution, can be enhanced to improve the system's usability beyond its current level. For example, the number of conflicts reported by the automated resolution facility could be reduced by incorporating semantic knowledge that is missing in the current implementation; and, the repair facility could be enhanced to provide greater assistance to the user during the resolution process.

The resolution techniques described in the thesis were developed with specific goals. However, as the system matures and new technologies are developed some of these techniques could be reused, with slight modifications, for a different purpose. For example, the resolution facility that is currently only used to cope with write-write conflicts could be extended to cope with read-write conflicts also.

This section describes the suggested enhancements in two parts: the first part discusses extensions to the implementation that will improve the system's usability while the second part discusses enhancements needed to use the resolution mechanisms in a slightly modified environment.

### 9.2.1 Implementation Extensions

**Automatic Directory Resolution:** The implementation of directory resolution could be extended immediately in two ways: (1) to automatically resolve partitioned cross-directory renames and (2) to automatically resolve concurrent partitioned updates to directory access-control lists. The former extension needs the transitive closure mechanism described in Chapter 4. The latter extension requires a fine-grained value certification scheme rather than the coarse-grained version certification scheme currently implemented. In other words, the resolvability of a set of partitioned access-control list updates should be determined using the values of the individual elements in the list's replicas and not the version number for the entire list. This extension would require a change to the structure of the log record for this update and a modification to the certification algorithm.

Two long-term enhancements that are less critical than the extensions mentioned above are: (1) use of an aggressive resolution policy during conditions of light load and (2) use of resolution rules to implement directory specific resolution policies. The former enhancement would hide the cost of resolution from the user. The latter extension would provide users the flexibility of choosing a strategy *a priori* for coping with each conflicting update.

The current implementation enforces a single system-wide policy for overwriting records when the resolution log is full. As mentioned in Section 4.4.2.3, the optimal log overwrite strategy is different for each volume. It depends on the volume's directory structure and the sequence of partitioned updates. Analyzing the effects of the various strategies using trace-driven simulations would be an interesting experiment worth pursuing. It will provide answers to two questions: How does the number of conflicts generated by each strategy compare with the number of conflicts generated by the optimal strategy? Should the implementation be changed to accommodate volume-specific log overwrite strategies?

**Automatic File Resolution:** A handful of ASRs were implemented to test the infrastructure used to support application-specific resolution. To verify the completeness of the rule language syntax and the execution model several more ASRs should be implemented. For example, ASRs could be implemented immediately for files like `mbox`, `.newsrc`, `.history` and `.mosaic-global-history`. In the long run, resolvers for new applications should be implemented along with the application itself so that the latter can ensure that all information needed for resolution is made available to the resolver.

**Manual Resolution:** The directory repair tool can be enhanced in two ways – by improving its graphical user interface and by providing new functionality. Displaying the tree structure of the replicas, highlighting the differences between the replicas and allowing the user to produce repair commands using a click and drag mechanism would significantly improve the tool's usability. New functionality that seems to be worth exploring includes: (a) showing the effects of the repair commands at the client before propagating them to the server so that users are given a chance to undo their actions; and (b) allowing users to specify a "correct" directory replica – the tool would generate repair commands by comparing each replica with the "correct" replica, not by comparing the replicas with one another. This enhancement will allow the tool to generate repair commands without further user assistance.

### 9.2.2 Impact of New Research

Coda is being extended in several ways by other members of the research group. Two of these research efforts in particular, namely *isolation-only transactions* and *weakly-connected operation*, could benefit from some of the resolution techniques described in the thesis. The goals of these research efforts and their potential use of resolution techniques are discussed below.

**Isolation-only transactions:** As mentioned in Chapter 3, lack of transaction support in the Unix programming interface poses a problem for partitioned operation. The basic problem is that the system is unaware of the grouping of operations into computations and must assume that either all operations in a partition belong to one computation or each operation belongs to a separate computation. The former assumption hurts availability because the system, in an attempt to disallow partitioned read/write conflicts, must disallow write operations in all but one partition. The latter assumption, which is made in this thesis, can sometimes hurt correctness – the system may expose inconsistent state to operations belonging to the same computation.

Work is already underway in the Coda project to overcome this shortcoming. It extends the Coda file system with a new transaction service called isolation-only transactions [30]. The extended system guarantees serializability of computations and ensures that both read-write and write-write conflicts are detected. However, it does not guarantee other transaction properties, namely failure-atomicity and permanence. Extending the Coda system to explicitly support transactions will effect resolution in two ways.

First, the implementation of resolution will need to be extended to resolve read-write conflicts in addition to the write-write conflicts it resolves currently. The ASR mechanism could be used to provide this support. A resolver would be associated with a transaction rather than a file or directory and would be executed automatically when a conflict is detected. A variant of the resolution rule language could be used to specify resolvers for each transaction type or for any transaction accessing a group of objects. Automatic transaction re-execution will probably be the most frequent method of resolution.

Second, resolution of write-write conflicts will benefit from the knowledge of transaction boundaries. Since the computation associated with each transaction will be known, the resolution process could undo and redo the conflicting transactions in a global serial order. Thus, the system could successfully resolve a larger set of histories – histories that are 1SR but not resolvable in the current system simply because they are not commutative would be resolvable. This advantage combined with the ability to perform transaction specific resolution could further reduce the need for manual resolution.

**Weakly-connected operation:** This research effort [33] is looking at ways to exploit intermittent and low bandwidth networks in Coda. Like a disconnected client, a weakly-connected client avoids use of the network by emulating remote services locally as far as possible. However, unlike a disconnected client, it uses the low bandwidth network to service cache misses and trickle updates to the server in small chunks.

An update made at a disconnected client may conflict with updates being made concurrently at the server. However the conflict can be detected only when the client is reconnected to the server, which is often a long time after the update is made. Most disconnected updates that follow the conflicting update affect the same or related objects and are also conflicting. Therefore, to cope with the conflict, the client saves all disconnected updates on its local disk, flushes the modified objects from the cache and lets the user propagate the updates manually. Can this situation be improved?

Connectivity with the server, even if it is intermittent, provides a weakly-connected client with an opportunity to detect conflicting updates sooner than its disconnected counterpart. By resolving the conflict soon thereafter, the client can prevent wasted effort – if the conflict is resolved then updates that follow can be propagated correctly to the server.

Some of the resolution mechanisms described in this thesis can be extended to automatically resolve conflicts generated by updates at clients. In particular, the ASR framework can be extended to resolve conflicts between first and second-class replicas. The second-class replica would be preserved by the client and included as a child of the fake directory used to expose the replicas. Furthermore the ASR would be invoked transparently as soon as the update propagation fails. Obviously, ASRs that will be used on weakly-connected clients would have to be judicious in their use of network resources.

## 9.3 Closing Remarks

Data availability is a fundamental problem faced by all distributed file systems. As argued earlier in the dissertation this problem will become worse as DFSs get larger and new technologies encourage the use of mobile clients. Replication, the key to solving this problem, must strike the right balance between three potentially conflicting requirements: *availability*, *consistency* and *usability*. Pessimistic replication provides full consistency but is considered unusable because it restricts availability excessively. Optimistic replication, on the other hand, provides the desired availability but is considered unusable because it compromises consistency excessively.

The usability of optimistic replication can be improved without sacrificing availability by automating the resolution of inconsistencies. In the best case, all inconsistencies would be resolved automatically and the system's usability would be completely restored. In practice however some inconsistencies cannot be resolved by the system and the burden of their resolution falls on the user. Optimistic replication will be practical only if inconsistencies that cannot be resolved by the system occur rarely and the pain associated with manual resolution is much less than the cost of reduced availability. Furthermore, the system's availability, consistency or usability must never be compromised in the absence of failures.

This thesis has shown that optimistic replication can be used effectively in a distributed Unix file system and that the system's availability can be greatly enhanced without undue compromise of its consistency or usability. The automatic resolution techniques – a server-based mechanism that uses logging, and a client-based mechanism that uses application support – are simple yet effective in alleviating the shortcomings of optimistic replication. The efficacy of these mechanisms will pave the way for serious use of optimistic replication in distributed file systems.

# Bibliography

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A New Kernel Foundation for Unix Development. In *Proceedings of the Summer Usenix Conference* (June 1986).

[2] Alsberg, P., and Day, J. A principle for resilient sharing of distributed resources. In *Proceedings 2nd International Conference on Software Engineering* (October 1976).

[3] Bernstein, P., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] Bershad, B. N., and Pinkerton, C. B. Watchdogs - Extending the UNIX File System. *Computing Systems 1*, 2 (Spring 1988).

[5] Blaustein, B., Garcia-Molina, H., Ries, D., Chilenskas, R., and Kaufman, C. Maintaining Replicated Databases Even in the Presence of Network Partitions. In *Proceedings of the IEEE EASCON Conference* (September 1983).

[6] Blaustein, B., and Kaufman, C. Updating Replicated Data During Communications Failures. In *Proceedings of the Eleventh International Conference on Very Large Databases* (August 1985).

[7] Cannan, S., and Otten, G. *SQL – The standard handbook: based on the new SQL standard*. McGraw-Hill, 1993.

[8] Davidson, S. *An Optimistic Protocol for Partitioned Distributed Database Systems*. PhD thesis, Princeton University, October 1982.

[9] Davidson, S. Optimism and Consistency in Partitioned Distributed Database Systems. *ACM Transactions on Database Systems 9*, 3 (September 1984).

[10] Davidson, S., Garcia-Molina, H., and Skeen, D. Consistency in Partitioned Networks. *ACM Computing Surveys 17*, 3 (September 1985).

[11] Ebling, M. Evaluating and Improving the Effectiveness of Hoarding. Thesis proposal, Carnegie Mellon University School of Computer Science, April 1993.

[12] Ebling, M. R., and Satyanarayanan, M. SynRGen: An Extensible File Reference Generator. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, TN, May 1994).

[13] Eppinger, J., Mummert, L., and Spector, A., Eds. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

[14] Garcia-Molina, H., Allen, T., Blaustein, B., Chilenskas, R., and Ries, D. Data-Patch: Integrating Inconsistent Copies of a Database after a Partition. In *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems* (October 1983).

[15] Garcia-Molina, H., and Wiederhold, G. Read-Only Transactions in a Distributed Database. *ACM Transactions on Database Systems 7*, 2 (June 1982).

[16] Gifford, D. Weighted Voting for Replicated Data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles* (August 1979).

[17] Gray, J. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[18] Guy, R. A Replicated Filesystem Design for a Distributed Unix System. Master's thesis, University of California, Los Angeles, 1987.

[19] Guy, R. *Ficus: A Very Large Scale Reliable Distributed File System*. PhD thesis, University of California, Los Angeles, June 1991.

[20] Guy, R., Heidemann, J., Mak, W., Page, T., Popek, G., and Rothmeier, D. Implementation of the Ficus Replicated File System. In *Proceedings of the Summer Usenix Conference* (June 1990).

[21] Herlihy, M. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions on Computer Systems 4*, 1 (February 1986).

[22] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[23] Howard, J. H. Using reconciliation to share files between occasionally connected computers. In *Proceedings of the 4th IEEE Workshop on Workstation Operating S ystems* (Napa, CA, October 1993).

[24] Irlam, G. A Static Analysis of Unix File Systems circa 1993. `ftp://cs.dartmouth.edu/pub/file-sizes/ufs93b.tar.gz` (October 1993).

[25] Kistler, J. *Disconnected Operation in a Distributed File System*. PhD thesis, Carnegie Mellon University, Pittsburgh, May 1993.

[26] Kumar, P., and Satyanarayanan, M. Supporting Application-Specific Resolution in an Optimistically Replicated File System. In *Proceedings of the 4th IEEE Workshop on Workstation Operating S ystems* (Napa, CA, October 1993).

[27] Kung, H., and Robinson, J. On Optimistic Methods for Concurrency Control. *ACM Transaction on Database Systems 2*, 6 (1981).

[28] Lampson, B., and Sturgis, H. Crash Recovery in a Distributed Data Storage System. Tech. rep., Computer Science Laboratory, Xerox Palo Alto Research Center, 1976.

[29] Levy, E., and Silberschatz, A. Distributed File Systems: Concepts and Examples. *Computing Surveys 22*, 4 (December 1990).

[30] Lu, Q., , and Satyanarayanan, M. Isolation-only Transactions for Mobile Computing. *ACM Operating Systems Review* (April 1994).

[31] Mashburn, H. *RVM User Manual*, 1.1 ed. Carnegie Mellon University School of Computer Science, June 1992.

[32] Minoura, A., and Wiederhold, A. Resilient extended true-copy token scheme for a distributed database system. In *IEEE Transactions on Software Engineering* (May 1982).

[33] Mummert, L. Exploiting Weak Connectivity in a Distributed File System. Thesis proposal, Carnegie Mellon University School of Computer Science, December 1992.

[34] Needham, R., and Schroeder, M. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM 21*, 12 (December 1978).

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

[36] Ousterhout, J. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[37] Parker Jr., D., Popek, G., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., and Kline, C. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering SE-9*, 3 (May 1983).

[38] Popek, G., and Walker, B. *The LOCUS Distributed System Architecture*. MIT Press, 1985.

[39] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G. LOCUS: A Network Transparent, High Reliability Distributed System. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles* (December 1981).

[40] Purdin, T. *Enhancing File Availability in Distributed Systems (The Saguaro File System)*. PhD thesis, University of Arizona, August 1987.

[41] Purdin, T., Schlichting, R., and Andrews, G. A File Replication Facility for Berkeley Unix. *Software Practice and Experience 17*, 12 (December 1987).

[42] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. Resolving File Conflicts in the Ficus File System. In *USENIX Summer Conference Proceedings* (Boston, MA, June 1994).

[43] Rudisin, G. J. Architectural Issues in a Reliable Distributed File System. Master's thesis, University of California, Los Angeles, 1980.

[44] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and Implementation of the Sun Network Filesystem. In *Summer Usenix Conference Proceedings* (June 1985).

[45] Satyanarayanan, M. On the Influence of Scale in a Distributed System. In *Proceedings of the Tenth International Conference on Software Engineering* (April 1988).

[46] Satyanarayanan, M. A Survey of Distributed File Systems. In *Annual Review of Computer Science*. Annual Reviews, Inc, 1989. Also available as Tech. Rep. CMU-CS-89-116, Carnegie Mellon University School of Computer Science, February, 1989.

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

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

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

[50] Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., and Kistler, J. J. Lightweight Recoverable Virtual Memory. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, NC, December 1993).

[51] Satyanarayanan, M., and Siegel, E. Parallel Communication in a Large Distributed Environment. *IEEE Transactions on Computers 39,* 3 (March 1990).

[52] Thomas, R. A solution to the concurrency control problem for multiple copy datbases. In *IEEE Compcon* (Spring 1978).

[53] Walker, B., Popek, G., English, R., Kline, C., and Thiel, G. The LOCUS Distributed Operating System. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* (October 1983).