Improving Data Consistency for Mobile File Access Using Isolation-Only Transactions

Qi Lu

May 1996

CMU-CS-96-131

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Thesis Committee:
Mahadev Satyanarayanan, Chair
Jeannette Wing
David Garlan
Eliot Moss, University of Massachusetts

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, Intel Corporation, and AT&T.

Keywords: Mobile computing, distributed file system, disconnected operation, optimistic replication, transactions, conflict detection and resolution, application-specific resolution, IOT.

For Yin

Abstract

Disconnected operation based on optimistic replication has been demonstrated as an effective technique enabling mobile computers to access shared data in distributed file systems. To guard against inconsistencies resulted from partitioned data sharing, past research has focused on detecting and resolving write/write conflicts. However, experience shows that undetected read/write conflicts pose a subtle but serious threat to data integrity in mobile file access. Solving this problem is critical for the future success of mobile computing.

This dissertation shows that *isolation-only transaction (IOT)*, an upward compatible transaction mechanism for the Unix File System, is a viable solution to this problem. The central idea of the IOT model is imposing serializability-based *isolation* requirements on partitioned transaction executions. Transactions executed on a disconnected client stay in a tentative state until the client regains connection to relevant servers. They are committed to the servers as soon as they pass consistency validation. Invalidated transactions are automatically or manually resolved to ensure global consistency. Powerful resolution mechanisms such as automatic transaction re-execution and application specific resolver invocation can transparently resolve conflicts for many common Unix applications. In addition, a concise conflict representation scheme enables application semantics to be smoothly integrated for only conflict resolution and consistency validation. The practical usability of IOT is further enhanced by a flexible interactive interface, full compatibility with existing Unix applications, and the ability to retain overall file system scalability, security and transparency.

A working IOT implementation in the Coda file system has been developed and used in experiments in software development and document processing applications. Quantitative evaluation based on controlled experiments and trace-driven simulations establish that the IOT model is scalable and incurs modest performance and resource overhead.

The main contributions of this thesis research are the following: the design of an isolation-only transaction model specialized for improving mobile file consistency while preserving upward compatibility with existing Unix applications; the development of a working IOT implementation in the Coda file system; experimentation and evaluation demonstrating the feasibility and practicality of the IOT model.

Acknowledgments

First and foremost, I would like to thank my advisor Satya, without whom this thesis would not have been possible. He taught me design, implementation, and evaluation skills that are critical in completing this dissertation. His patience, encouragement and support made my transition from the previous research area of software engineering to distributed and mobile computing systems much easier. His unparalleled expertise and insight in experimental systems research have been a constant source of guidance throughout the thesis research process. I also would like to thank other members of my thesis committee, Eliot Moss, Jeannette Wing and David Garlan for their valuable help that significantly improves both the technical content and the presentation of this dissertation.

My stay at the School of Computer Science of CMU would not have been as enjoyable and rewarding without the wonderful people who made it a great place to do research in computer science. I am deeply indebted to my first advisor, late Professor Nico Habermann, for his invaluable help in my development as a graduate student and a researcher both technically and personally. I would like to thank Sharon Burks for being so helpful when I needed the most. Present and past Coda project members including Maria Ebling, Bob Baron, Jay Kistler, Hiroshi Inamura, Hank Mashburn, Lily Mummert, Brian Noble, Morgan Price, Josh Raiff, David Steer, and Tetsuro Muranaga offered me generous support. Jay helped formulating the central theme of my thesis; Puneet was always available for help; Tetsuro used my software in his mobile CSCW research to provide me valuable usage experience. LeAnn Neal Reilly carefully proof read the entire dissertation. I also would like to thank my officemates, past and present, for creating an intellectually simulating environment.

My parents taught me the value of hard work and instilled in me the desire to be successful. Without their love and sacrifice, I never could have come this far. Thanks also to my daughter, Diana, for reminding me that there is so much more to life than computer science. Finally, my wife, Yin, deserves greater thanks than I can possibly give. Over the past seven years, she has been an immeasurable source of strength, love, comfort, and support. Yin, this thesis is dedicated to you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the growing prevalence of portable computers, providing mobile computers with convenient access to shared data in distributed file systems (DFSs) becomes an important problem. Although disconnected operation has been shown to be a viable technique for supporting mobile access in distributed Unix file systems (UFS) [27, 62], it induces the risk of data inconsistency due to partitioned data sharing. This dissertation advocates a new abstraction called Isolation-Only Transaction (IOT) as an effective means to improve consistency for mobile file access [40, 41]. It demonstrates the viability of this approach through the design, implementation and evaluation of IOT in the Coda file system.

This chapter begins with a brief background of DFSs and disconnected operation. It then introduces the data inconsistency problems caused by partitioned sharing during disconnected operation. It concludes with the thesis statement and an outline of its substantiation in the rest of the document.

1.1 Mobile File Access

Distributed File Systems DFSs such as NFS [58], AFS [44, 24, 59], NetWare [50] and LanManager are becoming an integral part of the basic computing infrastructure in organizations with a large number of networked personal computers or workstations. Their increasing popularity can be attributed to the following reasons. First, DFSs allow information to be shared among a large number of physically dispersed users. Second, DFSs can provide location transparency by hiding the physical distribution of the underlying system so that the users can conveniently access shared data using their logical identities from anywhere within the system. In addition, the administrative burden on users is lessened because the users can concentrate on their own work while tasks such as backup and software maintenance are handled by trained personnel.

Most of the DFSs in widespread use are organized according to the *client/server* architecture. File data are managed by a nucleus of dedicated *server machines* operating at secure locations by a central authority. A large number of *client machines* from different locations can access the data through a standard interface such as the UFS API (Application Programming Interface). A key technique for enabling good performance and scalability is *client caching*, where copies of shared data are stored on the client machine so that they can be accessed without frequent communication with the servers. At large scale, data availability becomes a serious concern because frequent failures such as a network partition or a server crash denies users access to needed data. The main technique for improving data availability is *replication*, where multiple copies of the same data are maintained at different sites so that the data can accessed even though a portion of the network or the servers are down.

The Impact of Mobile Computers Mobile computers are one of the fastest-growing segments of the computer industry. Many portable computers are powerful enough to operate as a client of DFSs and benefit from the capability of transparently accessing shared data. Unfortunately, mobile access to shared data is constrained in important ways. First, mobile client machines are often resource poor relative to their stationary counterparts. Second, portable computers are less secure and more prone to loss and destruction. Most importantly, mobile elements of DFSs must operate under a wider range of network conditions. Because mobile connectivity is often absent, intermittent, slow or expensive, the ability to operate while disconnected provides a crucial worst-case fall-back position that the users can rely on.

1.2 Disconnected Operation

The Coda file system pioneered the use of disconnection operation as a basic technique for mobile file access [61]. The essence of disconnected operation is to enable a disconnected client to act as a temporary server and continue to service file access requests using the cache copies of the requested data. The effect of disconnection is hidden from the applications running on the mobile client machine as long as the data they access are cached locally. In principle, the cache copy of an object can be viewed as its second class replica and disconnected operation as a special form of optimistic replication [26], where the client replica can be accessed without restriction while partitioned from the corresponding servers. The design rationale of disconnected operation is based on optimistic replication. It pursues maximum data availability by relaxing the traditional partitioned replication control necessary for maintaining one-copy equivalence [9]. Practical experience with disconnected operation in Coda has demonstrated that it is a viable and effective technique for supporting mobile file access in a Unix environment [62].

1.3 Partitioned Sharing and Data Inconsistency

The superior data availability brought about by disconnected operation is not without cost. Arbitrary partitioned sharing allowed during disconnected operation can cause two kinds of conflicts and possibly leave data in an inconsistent state. Past research has focused on *write/write conflicts* where the same object is updated on both a disconnected client and the corresponding servers. Fortunately, empirical evidence has shown that partitioned write sharing is very rare in practice [26] and write/write conflicts can be efficiently detected and often transparently resolved [53, 30, 31, 29, 56, 32, 68]. The more subtle threat comes from *read/write conflicts* where the same object is read in one partition and updated in another. Partitioned read/write sharing can cause data inconsistencies in various ways as demonstrated by the following examples.

> *Example 1.* Mary is a business executive and she works on a report for an upcoming shareholders' meeting using a disconnected laptop during a weekend trip. Before disconnecting, she cached a spreadsheet with the most recent budget figures and she writes her report based on the numbers in that spreadsheet. While she is away, new budget data become available and the server copy of the spreadsheet is updated. The read/write conflict on the spreadsheet could leave the report in an inconsistent state if Mary fails to recognize the stale data in her report while other presentations by her colleagues at the meeting all cite the up-to-date figures.

It is true that even if Mary's laptop remains fully connected to the servers, the same inconsistency could still occur due to lack of coordination among the users. However, file access using disconnected operation substantially enlarges the window of vulnerability. Furthermore, detecting stale data upon reconnection is no small chore for Mary because her report may use data from many other files containing data such as sales figures and revenue projections. All of them could have changed on the servers during her absence. The consistency of the report would be much better protected if Mary could be notified upon reconnection about the server updates on any of the data she used in writing the report.

Based on an actual instance of using disconnected operation in the Coda file system, the following scenario illustrates that partitioned read/write sharing can also incur data inconsistency during software development activities.

Example 2. Joe is a programmer for a research software project and he decides to release a set of new libraries containing bug fixes. He executes a dedicated software administration script which installs the relevant archive files into a public area before posting an electronic bboard message announcing the new release. Unfortunately, Joe's client lost connectivity to the servers during the process and only part of the installed archive files make it to the servers while the rest stay in the client's cache waiting to be propagated upon reconnection. Because of the support of disconnected operation, the effect of disconnection is hidden from applications. Therefore, the script has no idea that part of the installation is only performed locally on the client and not visible from the servers. It will continue to send a message to the bboard to announce the installation. Let us suppose that the message is successfully posted to the bboard via a different network communication route and another user sees the post. The user then builds a new utility tool linking those installed libraries without knowing that some of them are still the old version. There are read/write conflicts here because some of the library files are updated in one partition and read in another. Such read/write conflicts could cause serious problems for the newly built utility tool because it is linked with an incompatible set of libraries.

Note that this example is different from the previous one in that the inconsistency *could not* have happened had Joe's client remained connected to the servers because that would have guaranteed that by the time the bboard announcement was posted all the installed new libraries were already visible on the servers. Moreover, the problem could get much worse if the utility tool is used to mutate other objects, thereby causing cascaded inconsistencies. The latter are often insidious and difficult to track down.

1.4 The Thesis

Motivation and Objective The above scenarios of read/write conflicts are not far-fetched. They can really be experienced during disconnected operation. Partitioned read/write sharing will become an increasingly important issue, particularly in large scale DFSs where information sharing among collaborating users is common. Undetected read/write conflicts pose a serious threat to data integrity and may impede the usability of disconnected operation.

The main objective of this research is to develop a practically usable mechanism that can guard against inconsistencies resulting from partitioned read/write sharing. Our central focus is improving consistency support for disconnected operation while maintaining upward compatibility with existing Unix applications.

Thesis Statement Our solution is an explicit transaction extension to UFS which provides not only the necessary context information for analyzing partitioned read/write dependencies but also ample opportunities for automatic conflict resolution. Our thesis statement is:

> *As an explicit extension to UFS, the isolation-only transaction mechanism automatically detects read/write conflicts via optimistic enforcement of serializability-based isolation requirements. Equipped with flexible conflict resolution facilities, IOTs can be effectively used to improve data consistency for mobile file access in distributed Unix file systems.*

Thesis Validation This thesis has been validated through the actual development of an IOT extension to the Coda file system. A working implementation has been produced, followed by extensive experiments in the application domains of document processing and software development. A thorough quantitative evaluation and an initial usability assessment offer strong evidence in support of the thesis statement.

1.5 Organization of this Dissertation

The rest of this document is organized as follows. Chapter 2 presents a close examination of the fundamental cause of data inconsistencies resulting from partitioned sharing. Chapter 3 highlights the key design objectives and constraints, followed by a detailed description of the IOT model.

Chapters 4 to 8 cover the detailed design and implementation issues of the development of an IOT extension to Coda. Chapter 4 describes how the IOT consistency model is enforced through an incremental transaction propagation scheme. Chapters 5 and 6 concentrate on the key issues of conflict representation and conflict resolution. A library programming interface as well as an interactive shell interface for using the IOT service are presented in Chapter 7. The remaining important implementation issues are discussed in Chapter 8.

Chapter 9 evaluates the design and implementation; the focus of the evaluation is the amount of system resources consumed in supporting the IOT service on a mobile client. The evaluation is based on controlled experiments as well as trace-driven simulation and analysis. Chapter 10 discusses related work and Chapter 11 concludes with a summary of the main contributions and a discussion of future work.

Chapter 2

Partitions, Conflicts and Inconsistency

The purpose of this chapter is to discuss the fundamental cause of data inconsistencies incurred by partitioned read/write conflicts. The two practical examples presented in the first chapter are just different manifestations of the same problem, the *discrepancy* between the standard UFS semantics and the weakened semantic guarantees induced by optimistic replication. The gap between what is expected by the applications and what is actually provided by the system in the presence of partitioned sharing opens a window of vulnerability. A clear understanding of the conceptual depth and scope of the problem is essential to developing a practical solution because it establishes a foundation upon which important design decisions can be analyzed and reasoned about.

To present an in-depth analysis of the causal relation between optimistic replication and data inconsistency, this chapter first examines how standard UFS semantics is weakened by optimistic replication. It then revisits the basic notions of data consistency and consistency maintenance. Finally, it identifies the conditions under which partitioned sharing will result in data inconsistency.

2.1 The Impact of Optimistic Replication on UFS Semantics

In this section, we first review the standard UFS semantics. We then describe how optimistic replication is normally performed in distributed Unix file systems. Finally, we explain how standard UFS semantics is weakened in the presence of partitioned sharing. Because there are many abbreviations frequently used in this chapter and throughout the dissertation, a list of important abbreviations together with a brief description and the corresponding page number for further reference are displayed in Table 2.1.

Abbreviation	Description	Page
ACID	Atomicity, Consistency, Isolation and Durability	28
API	Application Programming Interface	1
ASR	Application-Specific Resolver(Resolution)	32
CCS	Cache Coherence Status	60
CML	Client Mutation Log	46
DFS	Distributed File System	1
DRR	Dual Replication Representation	79
GC	Global Certification	31
G1SR	Global One-Copy Serializability	30
IFT	Inferred Transaction	23
IOT	Isolation-Only Transaction	25
OCC	Optimistic Concurrency Control	26
RVM	Recoverable Virtual Memory	133
SG	Serialization Graph	51
TML	Transaction Mutation Log	46
UFS	Unix File System	1
WFG	Wait-For Graph	49
1SR	One-Copy Serializability	23
1UE	One-Copy Unix Equivalence	11
2PC	Two-Phase Commitment	47
2PL	Two-Phase Locking	44

Table 2.1: A Table of Abbreviations

The Standard UFS Semantics The UFS API provides applications with a hierarchically organized file storage service via a set of system calls such as `read`, `write`, `create` and `unlink`. The UFS semantic model for reading and writing file system objects is based on the traditional *shared-memory model*. It guarantees that any read operation on an object will see the result of the most recent update on that object, and the result of any write operation must be immediately visible to all subsequent read operations.

By definition, the UFS semantics depends heavily on the notion of *ordering* among file access operations. Such an order is quite obvious among operations performed on a single machine, but often not so clear in a distributed environment where concurrent file access operations can be issued from different hosts simultaneously. A widely accepted formal definition of ordering among events in a distributed system is the *Lamport time* [34], which builds up the *happened before* relation by modeling a distributed system with two intuitive notions of *process* and *message communication*. A process `P` (e.g., a Unix process) consists of a sequence of events. If `P` executes a file access operation $\mathtt{op}_1$ before another operation $\mathtt{op}_2$, then $\mathtt{op}_1$ is said to have happened before $\mathtt{op}_2$. If event `a` is the sending of a messages `msg` and event `b` is the receiving of `msg`, then `a` happened before `b`. The relation is transitive, i.e., if `a` happened before `b` and `b` happened before `c` then `a` happened before `c`. The following example illustrates this definition.

> *Example 3.* Suppose that two users Joe and Mary use their own workstations to work cooperatively on a group of objects stored in a distributed file system. Joe fixes a bug in a source file `work.c` and sends Mary an email message after he finishes editing it. Mary then compiles a new version of `work.o` from the updated `work.c` for further testing. Even though performed on two different hosts, the editing of `work.c` happened before the compiling of `work.o` because the email communication between the two users establishes the distributed order.

UFS shared-memory semantics originated from the early days when Unix file systems were primarily operated on time-sharing machines. It has the advantage of being simple, intuitive and convenient for programming applications. In modern Unix systems, however, file systems are often distributed over a number of hosts for better concurrency and resource sharing. Distribution makes the UFS semantics more difficult to maintain, particularly when caching and replication are involved to improve performance and availability. To retain semantic compatibility, cache coherence protocols and replica control strategies are often the focal point of distributed file system designs.

Optimistic Replication in UFS Replication is a technique that has proven to be effective for improving data availability. When used in distributed Unix file systems, each logical object is maintained by multiple physical copies called *replicas* at different sites. When some of the sites are not accessible, an object may still be retrievable from other replicas, depending on the replica control policy. The traditional pessimistic replica control strategies such as *read-one-write-all* and its variations strictly maintain the one-copy equivalence to applications by limiting partitioned replica accesses. Optimistic replication goes much further by allowing arbitrary partitioned replica accesses, thus providing much higher data availability in the presence of partition failures.

In optimistically replicated distributed file systems, a commonly used replica control strategy is the *read-any-write-all-available* policy that allows applications to perform arbitrary access operations on an object as long as one of its replicas is accessible. A read operation always returns the most up-to-date version among all accessible replicas. Similarly, a write operation always reaches all the accessible replicas. When partitions are healed, mutations performed at different partitions of the system will be propagated to those replicas that have not received the new update. When the same object is updated in more than one partition, the conflicting partitioned updates may not be automatically resolvable.[1] In that case, the file system has to mark the object as inaccessible because the one-copy image for that object can no longer be maintained. The resolution of the different replicas often requires user intervention.

Weak UFS Semantics The superior availability of optimistic replication comes at the cost of weakened *currency guarantees* promised by the standard UFS semantics [61]. In the presence of partition failures, a read operation on an object only retrieves the most recent among the accessible replicas. There could very well be a more up-to-date replica in another partition. Similarly, a write operation can only propagate its result to the accessible replicas. A subsequent read operation on the same object in a different partition will not be able to see the new value. Furthermore, any partitioned update/update conflict would force the file system to break the one-copy image to applications and request user assistance to reconcile the conflict.

For brevity of discussion, we use the terms *weak read* and *weak write* to refer to a partitioned read and partitioned write operation. We also use *weak UFS semantics* to stand for the UFS semantics with optimistic replication where update/update conflicts are automatically detected. A detailed description of the actual weak UFS semantics implemented in the Coda file system can be found in [61].

Relaxation of UFS Semantics Many distributed UFSs relax the standard UFS semantics for various performance benefits. For example, AFS adopts a *session semantics* [35] for cross-client file sharing where updates to a file `foo` are not visible on the servers until `foo` is closed and other clients maintaining an open `foo` will not see the updates until `foo` is re-opened. As another example, NFS employs a 30-second delay in its cache write-back policy to improve performance. Updates are not immediately visible on the servers until 30 seconds later. The semantic relaxations in AFS and NFS are *bounded relaxation* because their semantic differences with standard UFS are limited and applications can still obtain the standard UFS semantics by using calls within the UFS API. For example, an AFS client can alway retrieve the most recent data on the servers using the `open` call. A client in AFS or NFS can use the `close` call to flush updates back to the servers immediately.

The essence of optimistic replication is using controlled semantic relaxation to achieve higher data availability. Most of the UFS semantics is not affected except for the weakened currency guarantees and the possibilities of divergent replicas of the same object. Unlike AFS and NFS, the semantic relaxation of optimistic replication (or the weak UFS semantics) is *unbounded relaxation* because it is impossible for the applications to avoid reading a stale file

[1]Theoretically, the file system can use one of the partitioned updates to overwrite the other if there is a well-defined order among the partitioned updates. In practice, however, it is too costly to keep track of such ordering information for all partitioned mutation operations because this would require maintaining a complete history of all the file access operations and message communication performed on every host of the system.

by just using calls within the UFS API. The currency degradation of file access operations depends on the dynamic system connectivity. Thus, the window of vulnerability for semantic deviation can persist as long as disconnection continues.

One-Copy UFS Equivalence Under weak UFS, applications executed in the presence of partition failures can produce different results than they would under standard UFS semantics. A concept that best characterizes such behavioral difference is *one-copy UFS equivalence (1UE)*. A distributed file system is 1UE if for every set of computations the final file system state generated by any partitioned execution is identical to executing the same computations on a single Unix host. Note that 1UE itself is not a semantic model. It is a concept that describes a special characteristics of the semantic model of a distributed file system. For example, Sprite [48] is one of the few distributed file systems that are 1UE because of its faithful distributed implementation of the standard UFS semantics.

Weak UFS is not 1UE and it can be demonstrated by the following two simple examples. Let us consider Example 3 once again. Suppose that during Joe and Mary's cooperation there is a network failure that causes their workstations to fall into two different partitions of an optimistically replicated file system. Note that even though Joe and Mary's machines are not able to communicate within the file system to share files, it is still possible for Joe to send email to Mary via a different network communication route. Obviously, Mary's compilation result `work.o` does not include Joe's bug fix because the replica of `work.c` that she accesses has not received Joe's update yet. If both users are not aware of the disconnection, the resulting state of `work.o` is certainly not what they have expected because repeating the same scenario on a non-replicated file system would guarantee that the new `work.o` contains the bug fix. Alternatively, suppose that there is a lack of coordination between Joe and Mary, and they both edit `work.c` on the replicas within their corresponding partitions. What they end up with is two diverged versions of `work.c` that require them to manually reconcile the difference, something that never happens in a non-replicated file system.

These two examples are particular instances of two basic classes of non-1UE behaviors called *stale read* and *diverging writes*. A stale read consists of a pair of read and write operations on the same object `obj`, denoted `Read(obj)` and `Write(obj)`. Although `Write(obj)` happened before `Read(obj)` and there are no other write operations on `obj` in between, `Read(obj)` retrieves the content of `obj` that existed before `Write(obj)` updated it. The case of diverging writes consists of a pair of partitioned write operations that update two different replicas of the same object with no well-defined order between them. Any computation that creates non-1UE results must contain at least one instance of stale read or diverging writes. Otherwise, any read operation in that computation would have read the same value under standard UFS semantics and they would have created 1UE results instead.

2.2 Consistency Maintenance

The Application-Specific Nature of Consistency In the context of a Unix file system, the notion of consistency is attached to a group of objects, denoted as `OBJS` = { o_1, o_2, ..., o_n }. They are considered to be consistent or in a consistent state when their data contents satisfy a set of conditions that are necessary for them to serve their intended purposes and for the applications accessing them to function properly. The consistency requirements for OBJS are often expressed as a set of predicates `CONS(OBJS)` = { $P_1(o_1, ..., o_n)$, ..., $P_2(o_1, ..., o_n)$}. It is the applications that decide whether a group of objects are consistent or not. The same group of objects can be in a consistent or inconsistent state depending on which applications are using them and under what situations.

Suppose that the files `work.c` and `work.o` of Example 3 reside in a system installation area that is publicly visible to other users. The two objects are considered mutually consistent when `work.o` is the compilation result of `work.c` and every time `work.c` is updated a new version of `work.o` must be immediately re-compiled. Their consistency requirement is `CONS(work .c, work.o)`$_{public}$ = { `work.o` *is compiled from the latest version of* `work.c` *using* `cc`}. Consider a different scenario where `work.c` is in the public installation area and `work.o` is in the private workspace of Joe and compiled with a different compiler for experimentation. Joe may prefer `work.o` to remain unchanged even when `work.c` gets updated by a new release. Therefore, the new consistency requirement is `CONS(work.c, work.o)`$_{joe}$ = { `work.o` *is compiled from version* `V` *of* `work.c` *using* `cc` $\wedge$ `version(work.c)` $\geq$ `V`}.

Note that we make a clear distinction between the two concepts of conflict and inconsistency in this dissertation. The notion of conflict is syntactic, meaning that the same object is written in one partition and read or written in another. The notion of inconsistency is semantic, meaning the data contents of a group of objects do not satisfy the conditions required by the applications that use those objects. Conflicts can be detected by the file system, whereas inconsistency can only be detected by the corresponding applications.

Application Responsibility Unix file systems treat file objects as uninterpreted byte sequences. They do not have the semantic knowledge to interpret data stored in the files. As long as a file system strictly implements the standard UFS semantics, it bears no responsibility for any inconsistency which occurs in the file system. It is the applications that are responsible for not only maintaining data consistency but also detecting inconsistency and restoring consistency. After all, it is the applications that have the final say on whether a group of objects are consistent or not.

In Unix file systems, inconsistencies among objects often exist. For example, the public release of a software system often consists of a collection of source code, object code and executable files. A common consistency requirement is that the object and executable files must be built from the corresponding source files. Such a consistency requirement is often briefly violated during the process of system upgrade when only some of the object and executable files are re-built from the new source files. In addition, human errors such as forgetting to compile a library file could also cause inconsistency in the release. The first kind of inconsistency only exists for a short duration while the second kind can persist for a long period of time. But sooner or later the inconsistencies will be discovered by the users and applications when they lead to unexpected results, and consistency will be restored afterwards.

File System Responsibility When the implementation of a file system is not strictly faithful to the standard UFS semantics, maintaining data consistency becomes a shared responsibility between the file system and applications. When a relaxed semantic model such as the weak UFS semantics is employed, there is a window of vulnerability for a certain application, denoted `APP`, to produce a different computation result than it expects. Suppose that the final system state after executing `APP` from the initial state $\texttt{S}_{init}$ is $\texttt{S}_{std-UFS}\texttt{(APP, S}_{init}\texttt{)}$ and $\texttt{S}_{weak-UFS}\texttt{(APP, S}_{init}\texttt{)}$ under the standard UFS semantics and weak UFS semantics respectively. Also suppose that the set of objects involved is $\texttt{OBJS}_{APP}$ and their consistency requirement is $\texttt{CONS(OBJS}_{APP}\texttt{)}$. If $\texttt{S}_{std-UFS}\texttt{(APP, S}_{init}\texttt{)}$ satisfies $\texttt{CONS(OBJS}_{APP}\texttt{)}$ while $\texttt{S}_{weak-UFS}\texttt{(APP, S}_{init}\texttt{)}$ does not, then the inconsistency is entirely the file system's responsibility. In other words, the file system is responsible for any inconsistency due to its inability to keep the semantic promises in the presence of partitioned conflicts. It is important to note that application errors and user mistakes still account for many inconsistencies. When $\texttt{S}_{std-UFS}\texttt{(APP, S}_{init}\texttt{)}$ fails to satisfy $\texttt{CONS(OBJS}_{APP}\texttt{)}$, it is `APP` and its users, not the file system, that are responsible for the inconsistency. In other words, the file system should not be held accountable for inconsistencies caused entirely by application errors and user mistakes.

2.3 From Partitioned Sharing to Inconsistency

Conflicts Cause Non-1UE Effects Weak UFS semantics itself does not lead to inconsistency. The necessary condition for an application to yield a different computation result under weak UFS than under standard UFS is that there must be conflicting partitioned accesses to different replicas of the same object. We define the term *read/write conflict* to mean a pair of read and write operations that access the same object in two different partitions. Similarly, a *write/write* conflict means that the same object is updated in two different partitions. Conflicts are the necessary condition for non-1UE effects because any instance of a stale read or diverging writes must involve a read/write or write/write conflict.

However, conflicts are not a sufficient condition for non-1UE effects. A read/write conflict can often produce the same result as under the standard UFS semantics. For example, suppose that Joe first prints out a copy of `work.c` from his desktop workstation and later edits it on his laptop, and there is a partition failure causing the two operations to be performed on two partitioned replicas of `work.c`. But the net result would be the same even if there were no partition failures and the standard UFS semantics are guaranteed. Notice that a write/write conflict always produces a non-1UE effect because one-copy equivalence can no longer be maintained.

Non-1UE Effects Cause Inconsistency Non-1UE effects can cause objects to fall into an inconsistent state even though the involved applications and users all operate correctly. A pair of diverging writes immediately renders an object unusable and a stale read operation can lead to inconsistent behaviors in many different ways. As described in Example 3, reading a stale `work.c` can leave the compilation result of `work.o` in an inconsistent state because it does not reflect the latest bug fix as its user expects. Mary might have used it to build new system executables and happily notified other users that the bug has been fixed, only to find out later that that is not the case. A stale read can also inappropriately expose an inconsistent set of objects to the users, as exemplified by the following actual Coda experience.

> *Example 4.* A user runs a script to deposit the new release of a set of files into a public installation area and to announce the new release on an electronic bulletin board. Due to a partition failure, only some of the new files get to the public area. Other users who try to use the newly released files after seeing the announcement will end up getting a mutually inconsistent set of the corresponding files because only some of them are new.

Just as not all conflicts lead to non-1UE effects, not all non-1UE behaviors result in inconsistency. As mentioned in previous discussions, sometimes the users may prefer reading stale files to keep their own private workspace intact so that their private system development activities will not be affected by frequent updates in the public area. In addition, users can often tolerate reading stale files when the relevant new updates do not have any impact on their activities. For example, a user would not mind using an old version of `emacs` when its executable file is updated for a new release that enhances features he/she does not care about. Finally, a user may even willingly accept a stale read if the alternative is no data access at all.

Summary There are two important steps in the causal link from partitioned sharing to inconsistency: conflicts cause non-1UE effects, non-1UE behaviors cause inconsistency. Both are necessary conditions and not sufficient conditions. When weak UFS semantics is employed, the file system and applications have shared responsibility for consistency maintenance. However, the file system should only be responsible for those inconsistencies that result from non-1UE behaviors. This is because other inconsistencies are caused by mistakes made by users and applications, for which they are responsible.

Chapter 3

Design Rationale

This chapter outlines our fundamental design rationale and puts forth the IOT computation model. The first section highlights key design objectives and identifies main design constraints imposed by the dominant features of the underlying Coda file system and its target usage environment. The second section discusses a number of high level design decisions that shape the overall IOT structure. Finally, a detailed operational description of the IOT model is presented in the third section.

3.1 Design Objectives and Constraints

Data inconsistency resulting from partitioned sharing is a challenging problem that is of major practical significance. To develop a new file system mechanism capable of addressing such a complicated problem, we commence the description of our research by clarifying the main objectives we seek to achieve and determining the basic system context under which this endeavor is carried out. We choose to discuss design objectives and constraints before presenting the IOT model because this discussion is necessary to clarify key rationale and justify important design features of the IOT model.

3.1.1 Design Objectives

Our ultimate goal is to develop an IOT model and a working implementation that can be actually used by Unix users and applications to effectively address the data inconsistency problems in mobile file access. Practicality is the overriding goal, and it translates into the following specific design objectives.

Improved Consistency Support Providing improved consistency support for mobile file access is what originally motivated this thesis research. The IOT mechanism must provide help to Unix users and application programmers to obtain better consistency protection in the presence of partitioned file accesses, particularly on mobile computers using disconnected operation. Specifically, we want the IOT model to be able to recognize instances of data inconsistency that would otherwise be undetected in the current state of practice. Moreover, it must aid Unix users and programmers in managing the often-difficult task of restoring data consistency.

Upward Unix Compatibility A true test of practicality of any new system facility such as IOT is whether it can accommodate the large body of existing Unix applications. Maintaining upward Unix compatibility is a major priority. We must ensure that the behavior of Unix applications will remain unchanged as long as IOT is not involved. In addition, existing Unix applications with little or no change should be able to take advantage of the IOT service.

Limited Resource Consumption One of the unique concerns in supporting mobile file access is limiting resource cost on a portable computer because it often has less capacity than a stationary one. The IOT mechanism must be very sensitive to its resource consumption because excessive consumption may lead to denial of other valuable services to users and applications.

Ease of Use We pursue two complementary goals of conceptual simplicity and access flexibility. The IOT functionality must be presented to Unix users and application programmers with a simple abstraction that is easily understood and fully compatible with the traditional Unix application paradigm. The actual mechanisms through which the IOT service is accessed must be flexible and easy to use.

Reasonable Overall Performance Good performance is an intrinsic part of system usability. The IOT mechanism will inevitably incur a certain amount of performance overhead. But, it is imperative that we seek good overall system performance because excessive performance degradation could seriously undermine the usability of the IOT mechanism.

Summary The pursuit of practicality for a new transaction mechanism under the complicated setting of an optimistically replicated distributed file system is an arduous journey. As subsequent design and implementation trade-off analysis will demonstrate, the specific objectives of improved consistency, Unix compatibility, low resource cost, ease of use and good performance often create competing demands on various system components at both design and implementation levels. Our approach is to balance these concerns and make the necessary compromises that best serve the ultimate purpose of practical usability.

3.1.2 Design Context

Since the IOT mechanism is an extension to an existing distributed file system, its structural and functional properties are fundamentally dependent upon the underlying system infrastructure. In this section, we discuss the dominant Coda features that have substantial impact on IOT design and implementation. Our purpose is to identify a minimum set of specific file system characteristics under which the IOT model and its underlying principles can be applied to provide data consistency support.

Coda In a Nutshell As a descendant of the Andrew File System (AFS), the Coda file system provides a location-transparent view of a hierarchical name space to a large number of clients. Files are organized by the unit of *volumes*, each forming a partial subtree of the name space and typically containing the files of one user or project. The distinct features that set Coda apart from other distributed Unix file systems are the two complementary mechanisms it employs to achieve high availability: disconnected client operation and server replication, both relying on optimistic replica control. Disconnected operation is the basis for providing transparent mobile file access on portable computers in Coda.

Client/Server Architecture Coda employs whole-file client disk caching to achieve scalability and good performance. It uses the *callback* mechanism to maintain cache coherence. The Coda servers, collectively called *Vice*, are a group of dedicated Unix workstations placed in secure locations and running trusted software. At a Coda client, is a user level cache manager called *Venus* that transparently intercepts file access requests on Coda objects and services them using data fetched from Vice. The architectural reliance on Vice as the nucleus of the system eases the administrative burden, allowing the system to gracefully scale up to thousands of nodes. At the same time, it simplifies the system security model by only trusting the servers and not the clients.

Disconnected Operation Disconnected operation provides a client with continued file access in the presence of network failures. When the relevant servers are not accessible due to voluntary or involuntary disconnections, the role of the Venus cache manager is dramatically expanded so that it can temporarily become a self-reliant server servicing file access requests using its cache contents. Disconnected updates are performed only locally at the client cache and are logged and later reintegrated to the corresponding servers upon reconnection. Coda views caching as a special form of replication. The cache copy of an object is regarded as a second class replica while its server copy is a first class replica [26]. Conceptually, the disconnected operation mechanism is a form of optimistic second class replication.

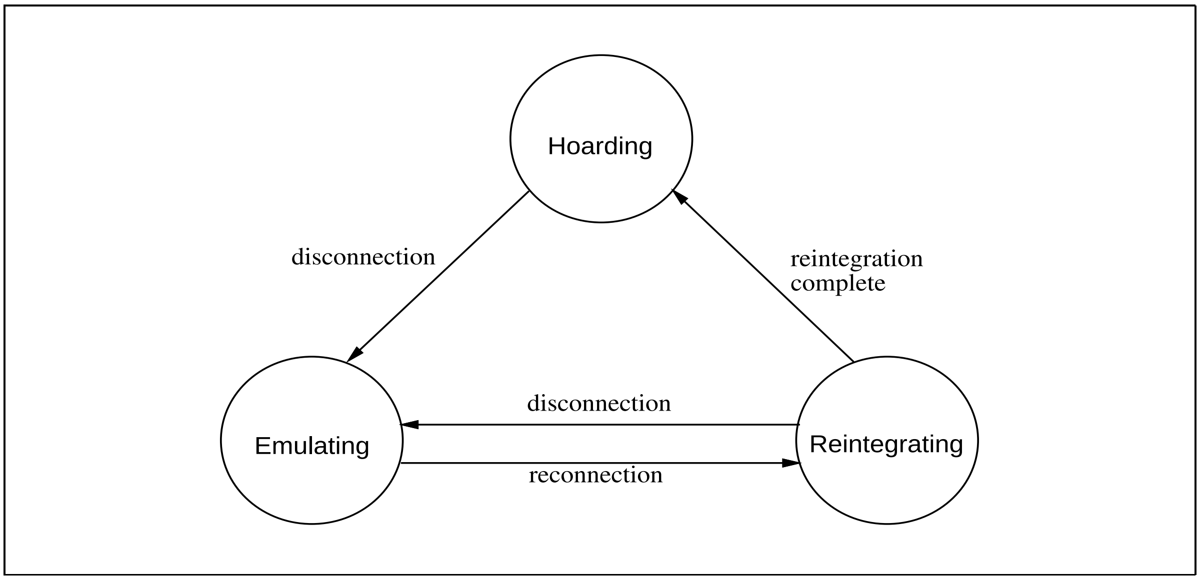


Figure 3.1: Venus States and Their Transitions

Venus operates in three different modes to cope with unpredictably changing system connectivities, as shown in Figure 3.1. When fully connected, Venus is in the *hoarding* state and functions as a normal cache manager performing client disk caching and maintaining cache coherence. When disconnected, it enters the *emulating* state and acts as a stand-alone server servicing file access requests using cache copies on its local disk. For applications running on a disconnected client, disconnection is transparent as long as the objects they access are locally cached. When the disconnected client regains lost connections, Venus transits into the *reintegrating* state to detect partitioned update/update conflict and propagate disconnected mutations to the corresponding servers. When the client cache state is fully synchronized with the server state, Venus goes back to the hoarding state.

Server Replication As a complementary mechanism to disconnected operation, server replication is also employed by Coda to further enhance data availability. The same logical data objects are replicated on multiple servers, allowing a client to service cache misses as long as one of the server replicas is accessible. Replication control among first class replicas uses the optimistic *read-any-write-all-available* strategy, allowing arbitrary partitioned sharing among different partitions of the system. Partitioned update/update conflicts are automatically detected using the version vector mechanism [53], and effective mechanisms are provided to transparently resolve conflicts [30, 31, 29, 56, 32, 68]. Although optimistic server replication helps to improve data availability, it does not have a direct role in supporting mobile file access.

Target Usage Environment Coda was designed to provide a general-purpose filing service for a network of workstations in an academic and research environment. Each workstation is operated by its primary user and the main target application domains are document processing, software development, and office automation. The likelihood of write-sharing among different users across clients is very low. As indicated by a recent study [49], the dominant usage form of disconnected operation is that users voluntarily disconnect their portable computers and continue their work at home or on trips.

Summary We have identified the above set of Coda features as the IOT design context because they have profound impact on how the IOT mechanism should operate. Although the actual implementation of IOT on Coda is subject to more restrictions from detailed Coda internal operations, the IOT model design is generic with respect to those main constraints. In other words, any distributed file system that supports disconnected operation, uses a highly scalable client/server architecture, provides good performance with client disk caching and assumes a similar usage environment should be able to accommodate an IOT extension for improved consistency support.

3.2 High Level Design Decisions

This section discusses a number of high level design decisions that essentially shape the overall IOT structure, paving the way for next section's presentation of the IOT computation model.

3.2.1 Focusing on Disconnected Operation

We limit the scope of this research by focusing on disconnected operation and excluding server replication from consideration. Our IOT model assumes that there is no server replication, and its design is optimized to provide consistency support only for disconnected operation. The main reasons that prompted this decision are the following. First, disconnected operation is the enabling technique for providing mobile file access in distributed file systems and improving its consistency support has a more significant practical impact. Second, server replication greatly complicates the task of consistency maintenance for partitioned file access operations. Third, a number of data inconsistency problems caused by optimistic server replication have already been addressed by a recent Ph.D. dissertation [29].

3.2.2 Optimistic Approach

There are two basic alternatives to safeguarding data consistency for disconnected operation. The pessimistic approach is to prevent potential inconsistency from happening by restricting partitioned file access operations using techniques such as quorum consensus and token passing [3, 18, 43, 23]. However, pessimism is fundamentally incompatible with the optimistic design principle embodied in disconnected operation. The IOT model inherits Coda's optimistic design philosophy by putting no limit on partitioned file accesses and by validating disconnected computations according to certain consistency criteria upon reconnection.

Similar to the traditional optimistic computation models, our approach assumes the low likelihood of having inconsistent disconnected computation results and takes advantage of this assumption to provide high data availability to applications executed on an isolated client. All disconnected computation results are considered tentative until the optimistic premise can be verified. The difference between our approach and the traditional optimistic models lies in the visibility of the tentative results. During disconnected operation, they are exposed not only to subsequent computations on the same client but also to external observers such as a human user, while the traditional models conceal them until the optimistic assumption can be confirmed. Therefore, it is imperative that we provide mechanisms that can be effectively employed not only to verify the optimistic premise of disconnected operation but also to compensate for the external side effects that are based on inconsistent, tentative computation results.

3.2.3 Inconsistency Detection

Because its main purpose is detecting data inconsistencies caused by partitioned file accesses, the IOT model effectively serves as the consistency criterion for deciding what kind of partitioned file accesses are admissible for disconnected operation.

Alternatives Based on the causal link from conflicts to non-1UE effects to inconsistencies discussed in the previous chapter, there is a spectrum of strategies for modeling the consistency criterion of disconnected operation. The main design trade-off is between the two competing goals of accuracy and efficiency. The accuracy of a consistency model measures how likely a computation rejected by the model causes actual data inconsistency. The efficiency of a model refers to the amount of computation needed for verifying the consistency of disconnected computations.

One end of the spectrum is to model the consistency criterion of disconnected operation using the application-specific definition of consistency itself, as has been recently explored in the Bayou System [68]. This approach has the advantage of perfect accuracy. However, it requires application semantic knowledge for all partitioned file access operations which is impractical for a general-purpose file system.

The other end of the spectrum is to approximate inconsistencies with partitioned conflicts. This model can be efficiently validated by recording and comparing object version stamps. However, its practical usability suffers from two main drawbacks. First, this approximation is gross because many instances of read/write conflicts are acceptable to Unix users and applications. Second, the notion of conflicts are defined for an individual object. Validating consistency at the granularity of an object instead of a unit of computation makes it difficult for the users to comprehend the nature and scope of the relevant inconsistency.

In between the two extremes, is another strategy which approximates inconsistencies with non-1UE effects, a much closer approximation than conflicts. However, detecting non-1UE effects requires the file system to maintain ordering information among all partitioned file access operations. Thus, it is impractical because it needs every host to record a complete history of file access operations and message communications.

Consistency Model Selection There are three main concerns in selecting a consistency model for validating disconnected computations. First, the model must accommodate common consistent behaviors of disconnected computations for typical applications in our target application domains under normal circumstances. In other words, any violation of the model must be very likely to result in inconsistent behaviors. Second, the automatic validation of the model must be computationally efficient. Third, the model should be able to screen out non-1UE behaviors as much as possible. This is very important in supporting our goal of Unix compatibility because the ability to detect non-1UE effects means that those Unix applications depending on strict UFS semantics can make use of the IOT facility to detect inconsistency without altering their semantic behaviors.

We decided to employ a serializability-based consistency model for validating disconnected operation. It requires certain serialization properties to be satisfied among partitioned file access operations. Although serializability theory has historically been used in database models to ensure the *isolation* property among concurrent transactions, it also captures the consistency requirement for a large variety of Unix applications such as `make`. Flexibility of the model is attained because there are a variety of serialization requirements to choose from. As will be shown in subsequent discussions, a strong serialization requirement called certification can be employed to detect the most common non-1UE behaviors. Finally, the feasibility of the model is assured because there are known efficient methods of validating serialization requirements.

Application Participation Any practical consistency model performs validation based on the syntax instead of the semantics of the relevant partitioned file access operations. Syntactic consistency validation is always capable of producing false negative results, i.e., some syntactically inadmissible disconnected computations may very well be semantically correct. A large gap between what is syntactically inadmissible and what is semantically inconsistent could seriously undermine the model's practical usability. The strategy we adopt to mitigate this problem is to provide support for *controlled application participation* in consistency validation. The key idea is to selectively apply application semantic knowledge to *re-validate* the consistency of certain disconnected computations after their syntactic validation has failed. This strategy combines the strength of high efficiency of a syntactically aggressive model and high accuracy from using semantic information.

3.2.4 Consistency Restoration

Effectively restoring data consistency after inconsistencies have been detected is an integral part of safeguarding the integrity of disconnected operation. For compatibility reasons, we inherit the terms *conflict* and *conflict resolution* from past research literature and use them in this dissertation with broader meanings. A conflict not only stands for a read/write or write/write conflict on an individual object but also refers to an invalidated transaction. Similarly, conflict resolution means not only the reconciliation among different replicas of an object but also the general process of restoring consistency for an invalidated transaction. Our resolution strategy embodies the following three characteristics.

Forward Progress The traditional way of restoring data consistency is to *rollback* the system state to a recorded previous state that is known to be consistent. For disconnected operation, however, throwing away the results of disconnected computations every time inconsistency is detected will seriously impede its usage. Our design pursues the *forward-progress* strategy so that the consistency restoration process's main mission is to preserve the work done while disconnected via adjustment and re-computation.

Transparency We also provide support for automatic execution of resolution actions that can restore data consistency and make the resolution process as invisible to the users as possible. The ability to transparently resolve conflicts is vital to IOT's practical usability.

Application-Specific Paradigm By the very nature of consistency, application semantics have an inherent role in conflict resolution, particularly in situations where compensating actions are necessary. Past success in using application-specific resolvers to resolve write/write conflicts [56, 32, 68] indicate that we need to allow pre-programmed application-specific actions to be automatically invoked in a systematic and controlled way to more effectively restore data consistency.

3.2.5 Starting Point: Inferred Transaction Model

James Kistler introduced an elegant consistency model called the *inferred transaction (IFT) model* for validating disconnected computations in Coda [26]. The key idea of the IFT model is to let the file system implicitly infer transactions rather than having them explicitly specified by applications or users. Most Unix file system calls constitute their own independent transaction and the IFT type and its corresponding system call are displayed in Table 3.1. The consistency criterion for validating disconnected inferred transactions is the widely used *one-copy serializability (1SR)*. With a number of carefully designed optimizations intended to enlarge the set of admissible disconnected computations, the resulting IFT model permits the same set of disconnected computations as weak UFS.

`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`
`mkobject (directory, name, object(file | directory | symlink), user)`
    `creat | mkdir | open | symlink`
`rmobject (directory, name, object(file | directory | symlink), user)`
    `rename | rmdir | unlink`

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 3.1: Inferred Transaction Types and UFS System Call Mapping

Although IFT is a much cleaner consistency model than weak UFS, it has two major limitations. First, the boundary of an inferred transaction is too small to fit the natural boundary of applications where the 1SR requirement should be applied. The 1SR guarantee among individual file access operations performed by a group of applications does not assure 1SR for these applications as a whole. Second, the IFT model admits all instances of stale read as legal partitioned computations. For Unix applications relying on strict UFS semantics for correctness, the IFT model is not strong (or restrictive) enough to assure their consistency in the presence of disconnections. The following two examples illustrate both limitations.

> *Example 5.* Consider the two non-1SR transactions `T1` and `T2` shown in Figure 3.2. When the four involved operations are treated as individual transactions, they can be serialized in the order of {`read A, read B, write A, write B`}. If both `T1` and `T2` require one-copy serializability to work correctly, the IFT model is not general enough to address their consistency needs.

Partition 1	Partition 2
`T1:  read A` `     write B`	`T2:  read B` `     write A`

This execution is not 1SR because `T1` sees the value of `A` before `T2` wrote it and `T2` sees the value of `B` before `T1` wrote it.

Figure 3.2: An Example of Non-Serializable Partitioned Transactions

> *Example 6.* In the Coda project, it is quite common for a user to edit a source file, say `cfs.c`, on a client C_{edit} and then compile it on another client $C_{compile}$. Suppose that there is a network failure causing $C_{compile}$ to fall into disconnected operation mode. Thus the compilation result of `cfs.o` does not include the update as the user expects. Because the two disconnected transactions {`read cfs.c`} and {`store cfs.o`} performed on $C_{compile}$ can be serialized before the connected transaction {`store cfs.c`} performed on C_{edit}, the IFT model would happily reintegrate the inconsistent `cfs.o` to the servers.

It is the very attempt to improve the IFT model that originally motivated this thesis. We started out this research trying to extend the scope of the IFT model so that the transaction boundary can fit the natural boundary of applications, and to strengthen the IFT semantics so that it can address the consistency needs of more Unix applications. The IOT model is the result of this effort.

3.3 Isolation-Only Transaction Model

The essence of the IOT model is to optionally execute applications as individual isolation-only transactions and impose serializability-based requirements for transaction[1] executions under various system connectivity. By demanding partitioned transaction executions to satisfy global serializability requirements, we establish a consistency model for automatically detecting partitioned read/write conflicts when partitions are healed. In addition, the IOT model provides flexible mechanisms so that when conflicts are detected, they can be effectively resolved automatically or manually.

Because of our overriding goal is practical usability, we focus our description of the IOT model on the operational aspects of the model. In addition, we try to explain relevant mechanisms from users' perspective as much as possible.

3.3.1 What Is IOT?

An Optional File System Facility IOT is an optional file system facility in the Unix File System that extends the UFS API with two new calls `begin_iot` and `end_iot`. Users and applications can use the two calls to explicitly bracket the execution of applications. Since IOT usage is optional, the semantic behavior of applications that do not use IOT are guaranteed to remain unchanged.

Transaction Entity A transaction is the execution of an application whose file access operations are guaranteed a set of properties specially designed for safeguarding data consistency in the presence of disconnection. The notion of a transaction in this document is used to refer to two different kinds of system entities depending on the context. Statically, a transaction is just an application program whose execution makes use of the two IOT calls. Dynamically, a transaction is a sequence of file access operations bracketed by `begin_iot` and `end_iot`. The dynamic entity of a transaction is the sequence of file access operations issued by the execution of its static entity.

[1] In the rest of this dissertation, we will use the term transaction to refer to an isolation-only transaction when there is no ambiguity in the context.

Process Hierarchy A transaction `T` is started when the file system receives a new `begin_iot` call and the process that issued the call, denoted P_T, is called the *master process* of `T`. Any file access operations issued by P_T or its descendent processes are considered members of the transaction. `T` is terminated when an `end_iot` call is received or the master process P_T exits, whichever comes first. This hierarchical process structure is intended to emulate the process structure of Unix application executions and ease the burden of programming transactional applications. For example, if we want to execute the `make` application as a transaction, simply putting the `begin_iot` and `end_iot` calls at the beginning and end will cause all the file access operations performed by `make` to be included in the scope of one single transaction.

Flat Structure The IOT mechanism does not support nested transactions for the following two main reasons. First, including the nesting capability in the IOT model will greatly complicate its design and implementation on practical distributed file systems. Second, the benefits of nested transactions such as increased concurrency and finer granularity of recovery control are not of main concern in this research. Because of the flat structure, a `begin_iot` call within the scope of an ongoing transaction serves no practical purpose and is treated as a no-op; equally, an `end_iot` call outside the scope of any transaction is also ignored.

In order to avoid premature transaction termination due to inadvertent transaction nesting, the transaction system also needs to maintain a counter for each ongoing transaction to record the current depth of internal `begin_iot` calls issued within the scope of the transaction. The counter is incremented for each internal `begin_iot` call, and decremented for each internal `end_iot` call. An ongoing transaction is terminated as soon as its counter becomes negative.

3.3.2 Execution Model

The design of the IOT execution model was inspired by Kung and Robinson's Optimistic Concurrency Control (OCC) model [33]. The key idea is to use the client cache as a private workspace for transaction execution while the servers maintain the public space that reflects the results of all the committed transactions. All file access operations issued by a transaction execution are performed locally. The results of transactional mutations are held within the client cache until the transaction can be validated and committed to the public space on the servers. No partial result of transaction execution is visible on the servers and from any other clients. The biggest advantage of this OCC-style execution model is that the scope of potential inconsistency is limited to the boundary of a client's local cache.

As shown in Figure 3.3, the execution of a transaction can go through a number of different states. When a transaction `T` is first initiated by a `begin_iot` call on a client C_T, it starts out in the *running* state and stays there as long as the execution is still going on. When `T` is terminated by the corresponding `end_iot` call or when its master process exits, it must transit

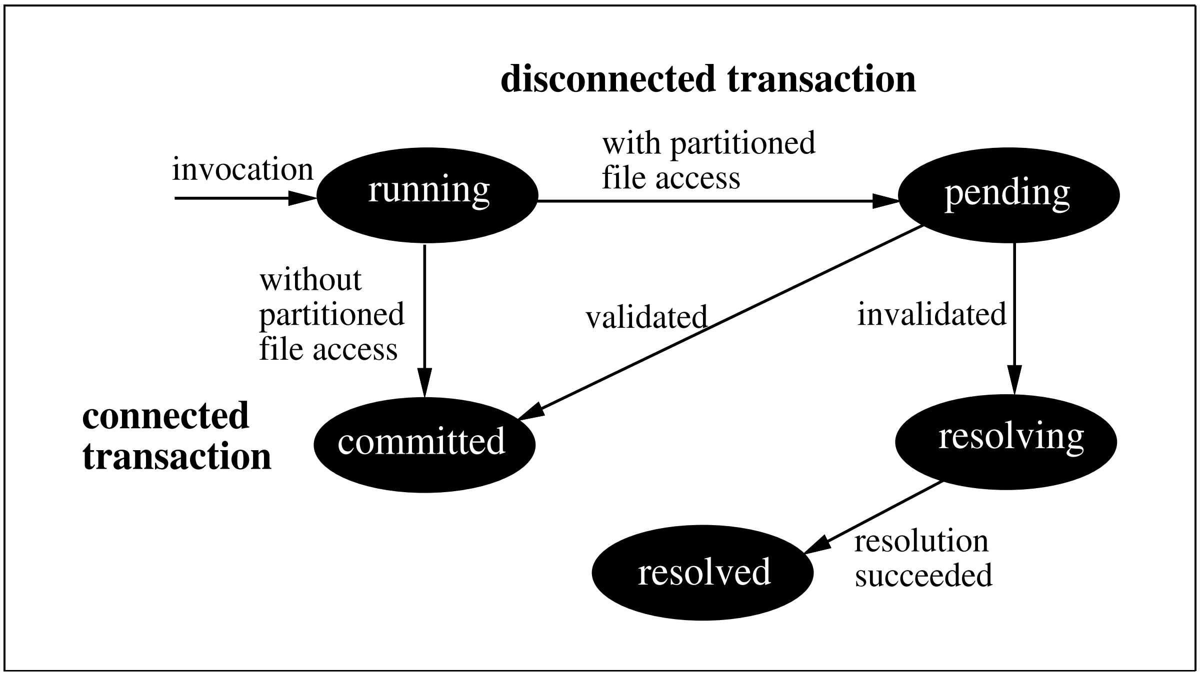


Figure 3.3: IOT States and Their Transitions

into a different state depending on whether it is *connected* or *disconnected*. T is a disconnected transaction if its execution has ever accessed one object for which C_T did not have server connection at the time of access.[2] In this case, T will transit into the *pending* state so that its validity can be verified when lost connections are regained. Otherwise, T is a connected transaction and will immediately write its result to the servers and go into the *committed* state.

The result of a pending transaction is confined within the client cache but still visible to subsequent processes on the same client. There can be multiple pending transactions executed on the same client, creating possible dependencies among themselves. If a transaction T *reads from* another pending transaction T', i.e., the execution of T reads the content of an object that is written by T', then the computation result of T logically depends on that of T'. In this situation, T must stay in the pending state as long as T' does, even though client C_T may maintain server connections for all the objects that T accessed. Therefore, the definition of a disconnected transaction needs to be extended to include the case of reading from pending transactions. Accordingly, the definition of a connected transaction needs to exclude those that read from pending transactions.

[2]The IOT model assumes a volume structure in the file system name space and the client/server connectivity is maintained at the volume level. The same client can be connected for one volume while disconnected for another.

A pending transaction `T` is validated when the following two conditions are satisfied. First, client C_T currently maintains server connections for all the objects accessed by `T`. Second, all the transactions that `T` read from are already committed or resolved. The validation of `T` checks whether the local result of `T` is consistent with the current global state maintained on the servers according to the consistency criteria required by the IOT model to be discussed shortly. If such validation succeeds, `T`'s result is immediately written to the servers and `T` goes into the *committed* state. Otherwise, `T` is invalidated (or `T` is an invalidated transaction) and it transits into the *resolving* state where its local result will be automatically or manually resolved against the current server state. When the resolution is completed and the new result is written to the servers, `T` enters the *resolved* state. Conceptually, there is no difference between the resolved and committed state; both mean that the computation (or re-computation) result of a transaction has been committed. From practical usability point of view, it is important to provide users with the information about whether a transaction is directly committed or has been successfully resolved. Therefore, we use two different states in order to reflect the different paths a terminated transaction went through.

3.3.3 Why Isolation Only?

The name "IOT" stems from the fact that we retain only the *isolation* guarantee of the traditional transaction's ACID (*A*tomicity, *C*onsistency, *I*solation and *D*urability) properties [14, 72, 20]. The design of the IOT model in many ways can be regarded as an exercise in minimalism. The IOT model embodies only those capabilities deemed vital to providing consistency support for disconnected operation. The consistency property in ACID requires any transaction to be a correct program that transforms one consistent system state into another, which is generally assumed in most distributed systems. Although failure atomicity and durability are valuable in distributed computing systems, they are not fully supported in the IOT model. Instead, restricted versions of both guarantees are provided. The reasons for this are explained below.

Regional Atomicity The IOT model does not guarantee that either all or none of a transaction is executed during system crashes for two main reasons. First, the system resource cost needed for supporting failure atomicity can be very expensive. In order to roll back the partial result of a transaction after a crash, the transaction system must record sufficient information to recover the system to a recorded consistent state. Such space cost could stretch the capacity limit on resource-poor portable computers because transactions running Unix applications such as `make` can last a long time and access a large number of big objects. Second, the all-or-nothing property is not compatible with many existing Unix applications that have developed their own approaches to crash recovery. For example, `emacs` employs *auto-save files* to guard against severe data loss caused by crashes. Moreover, there are situations where atomicity is simply undesirable. For example, suppose a user runs `make` as a transaction to compile many object files and build a large system. The machine crashes just before the task is about to complete. The user would much prefer to keep the compilation results and resume the task after the machine comes back, rather than having the system automatically throw away the partial results and re-start the whole process all over again.

As pointed out by previous research, the atomicity boundary and the concurrency control boundary need not be the same for many common applications [42]. The IOT model takes a regional approach in utilizing failure atomicity for fault tolerance. Specifically, a number of small segments of transaction execution such as validating a pending transaction and committing its local result to the servers are performed as atomic units.

Conditional Durability The traditional durability property requires that once a transaction completes successfully, its result must be able to survive all system failures. This also implies that once the result of a transaction is made visible to external observers, it must remain a permanent part of the system state until modified by later transactions. In the IOT model, the result of a pending transaction is visible to not only subsequent transactions on the same client but also external observers such as human users. It is also subject to change upon future validation and resolution, during which system crashes can happen. Therefore, the durability of a transaction result can only be guaranteed when the transaction is successfully committed or resolved.

3.3.4 Consistency Model

In contrast to the weakened guarantees of atomicity and durability, the IOT model strengthens the isolation guarantee to ensure that any interleaved and/or partitioned executions of a set of transactions are equivalent to a serial execution of the same set of transactions. We use the term *consistency model* here to stand for the actual mechanisms for achieving isolation of transaction execution. This is because data consistency can be ensured by the isolation guarantee as long as each individual transaction transforms one consistent system state into another.

3.3.4.1 Connected Transactions

Connected transactions are executed under a connected system environment. All file access operations are provided with the standard UFS semantic guarantees. Thus, a natural selection of the IOT consistency guarantee for connected transactions would be the traditional one-copy serializability (1SR). This means that when a connected transaction is committed, the IOT mechanism guarantees that it is one-copy serializable with all previously committed transactions[3].

3.3.4.2 Disconnected Transactions

Selecting the consistency requirement for disconnected transactions is much more complicated than for connected transactions. First, disconnected transactions represent tentative computation results that are local to a disconnected client. The IOT consistency model must ensure that the visible client state presents a locally consistent view of those disconnected transactions. Second, the validity of disconnected transactions must be verified before they can be committed to the servers. Thus, the IOT consistency model serves as the criterion for validating disconnected transactions and in effect decides what kind of disconnected computations are admissible.

Local Serializability (LSR) The IOT consistency model requires that any pending transaction on a disconnected client must be locally serializable with other pending or ongoing transactions executed on the same client. LSR ensures that local interleaved transaction executions are always equivalent to a serial execution, which is necessary for the pending transactions to present a locally consistent view about their results.

Global One-Copy Serializability (G1SR) The IOT consistency model adopts two serializability-based criteria to define admissible disconnected computations. The first one is called *global one-copy serializability (G1SR)*: if a disconnected transaction `T`'s result is copied to the servers as is, `T` must be 1SR with all previously committed transactions. Essentially, the G1SR consistency criterion ensures that if the results of disconnected transactions are simply propagated to the servers, the effect would be equivalent to some serial execution of all involved transactions in a non-replicated environment.

G1SR was first introduced as an optimistic transaction model by Davidson [7, 8, 9]. As a natural adaptation of the traditional 1SR model to an optimistic replication environment, G1SR is useful for a wide variety of applications where isolation of execution suffices to ensure correctness. It is also suitable in situations where the durations of disconnections are short. This is because under such conditions, distributed computations often appear to be concurrent computations, for which 1SR can best address the consistency needs.

However, G1SR alone is not adequate for our purpose of supporting mobile file access because it admits many instances of non-1UE behaviors. Some Unix applications rely on the currency guarantees of the standard UFS semantics for their correctness. The failure to catch

[3]When used in the discussion of different serializability guarantees, *committed* transactions also include *resolved* transactions because they are conceptually equivalent in terms of their impact on the global server state.

non-1UE effects would allow some incorrect disconnected computations to go undetected. G1SR's inability to detect non-1UE behaviors makes it particularly inadequate for addressing the consistency needs of long-lasting voluntary disconnected operation sessions, the dominant usage form of Coda laptop clients as shown in the example below.

> *Example 7.* A Coda programmer, Joe, caches relevant files on his laptop for a weekend trip. While disconnected, he does some hacking and executes `make` as a transaction T_{Joe} to build a new version of `cfs`, a Coda utility program. But one of the linked libraries `libutil.a` is updated by the system administrator on the servers using a connected transaction T_{Admin} during Joe's absence. Suppose that there are no other relevant file accesses. When Joe comes back from the trip and re-connects his laptop to the servers, T_{Joe} will be admitted under G1SR because it can be serialized before T_{Admin}. However, Joe would like his new `cfs` to be compatible with the latest libraries or at least be notified about the changes to the files that his work depends on.

Global Certification (GC) As a remedy to the limitation of G1SR, the IOT consistency model adopts a stronger consistency criterion for validating disconnected transactions called *global certification (GC)*. GC requires that if a disconnected transaction `T`'s result is copied to the servers as is, `T` must be not only *serializable with* but also *serializable after* all the previously committed transactions. GC can address the above problem by invalidating transaction T_{Joe} because it can not be serialized after T_{Admin}. The IOT model provides both the G1SR and GC consistency criteria for disconnected transactions and allows transaction programmers and users to specify the choice according to their consistency needs.

Intuitively, the GC criterion assures that the data accessed by a disconnected transaction are unchanged on the servers during the disconnection. It is a very strong consistency requirement and any disconnected transaction satisfying GC represents a disconnected computation whose result would be the same had the client not been disconnected at all. In fact, GC is so restrictive that it can detect any instances of stale read where the read operation is performed on a disconnected client. GC is much more suitable than G1SR for voluntary disconnected operation because the users are fully aware of the disconnection and more concerned about whether their disconnected computations are still compatible with the up-to-date server state rather than whether the disconnected computations can be serialized with committed transactions. Furthermore, it is conceptually simple and easier for Unix users and application programmers to grasp.

3.3.4.3 Conflict Resolution Options

The fundamental difference between 1SR for connected transactions and G1SR/GC for disconnected transactions is that the former is strictly enforced at transaction execution time while the latter can only be validated for pending transactions when communication with the relevant servers is restored. Hence, as an integral part of consistency maintenance for disconnected transactions, the IOT consistency model must specify what the system will do if validation fails. Our design provides the following four conflict resolution options to assist transaction users and programmers to restore consistency.

Automatic Re-execution When a disconnected transaction fails validation, one way to restore data consistency is to *re-execute* the transaction automatically by accessing up-to-date data from the servers. Consider Example 7 once again. If Joe wants to make sure that the `cfs` executable built on the disconnected laptop is compatible with the latest system release, he can choose GC as the consistency criterion and automatic re-execution as the resolution option for T_{Joe}. When T_{Joe} is invalidated at re-connection time, the IOT system will automatically rerun *make* to build an up-to-date version of `cfs` by linking in the new `libutil.a`.

The main advantage of this option is that it can be performed automatically without putting additional demand on transaction programmers and users. It is particularly useful for a class of Unix applications that primarily act as translators, reading input data from a set of files and generating output into a different set of files, such as `make`, `cc`, and `latex`. The ability to restore consistency automatically and application independently provides much-desired transparency in the resolution process, making automatic re-execution the workhorse of conflict resolution in our target application domains.

Automatic Abort The second conflict resolution option is to automatically *abort* an invalidated transaction by throwing away its local computation result. It is similar to the *rollback* mechanism of the traditional transaction recovery. Like the first option, it is also automatic and application independent. But it is only applicable in a few situations where getting rid of the local results is more appropriate than making forward progress. For example, suppose that a transaction runs every midnight traversing a file system subtree, computing statistics such as height and average branch factor of the subtree, and appending a new record to a log file accumulating the statistics. If such a transaction is invalidated, it means that the statistics it computed were based on a stale version of the subtree cached on the client. Automatic abort is an appropriate resolution choice for this transaction because it prevents an incorrect record from being appended to the log file.

Application-Specific Resolver (ASR) Our third option is to automatically invoke a user-supplied *application-specific resolver (ASR)* [29, 32, 68]. At the start of execution, a transaction can specify a user program that will be automatically invoked for conflict resolution if this transaction is later invalidated. ASR is a unique feature of the IOT model and is the key mechanism that allows integration of application semantic knowledge into the basic transaction operations.

First, the ASR mechanism enables automatic compensation of disconnected transactions containing external side effects. Consider a schedule maintenance program used by a business executive to automatically check her schedule files and send out fax messages to arrange business activities. Suppose that she takes her portable computer on a trip. While disconnected, she runs the program as a transaction to enter a few new appointments and sends out fax messages to notify relevant people about an upcoming meeting. However, her secretary adds several conflicting activities into the server copies of the schedule files during her absence. When the executive reconnects her portable computer to the servers, the transaction system can automatically execute a resolver program that checks for conflicting appointments and sends out new fax messages if necessary.

Second, the ASR mechanism allows application semantics to be utilized to resolve conflicts more effectively. Let us revisit Example 7 one more time. Suppose Joe ran a large `make` transaction to build a new kernel on the laptop. However, one of the linked libraries was updated during the disconnection. Using the automatic re-execution option can certainly resolve the conflict, but it would throw away all the local compilation results and rerun the long transaction all over again. Instead, we can program a resolver that takes advantage of the semantic knowledge of `make` to avoid unnecessary recompilations. In this case, the resolver can simply keep most of the local compilation results and re-link them with the new library.

Third, the ASR option enables application-specific consistency validation. By default, consistency validation is performed solely on the basis of syntactic information of file access operations. But this is a conservative approach. An invalidated transaction does not necessarily mean that the data involved are actually inconsistent. The ASR mechanism provides the resolver an opportunity to refine the check for data inconsistency using application semantics. Consider the following example where machine reservations for two computer clusters A and B are stored in files `reserveA` and `reserveB`. A simple consistency requirement for both files is that the same user can not reserve machines at both clusters. To reserve a slot for user Joe in cluster A, a transaction T_{Joe} is executed to add a new record in `reserveA` after making sure that `reserveB` does not contain a conflicting record. Suppose that T_{Joe} is a disconnected transaction due to a partition failure. Meanwhile, file `reserveB` is updated on the servers to add a reservation record for user Mary. Even though transaction T_{Joe} will be invalidated, its local result is consistent with the new server state. The unnecessary resolution for T_{Joe} can be avoided by using a resolver that is capable of recognizing the consistency of `reserveA` and `reserveB` based on application semantics and retaining the transaction's local result.

In summary, the application-specific conflict resolution paradigm is a centerpiece of the IOT model that integrates different aspects of the model into a coherent design. First, our need for practical usability and Unix compatibility demands transaction validation to be computationally efficient and capable of screening out non-1UE behaviors. Such requirements entail the selection of a restrictive consistency criterion such as GC. However, ASR supplements GC by integrating application semantics in consistency validation. As a result, consistency validation is well supported by two complementary mechanisms: efficient GC validation for the common cases and application-specific validation for the subtle cases. Second, disconnected operation departs from the traditional optimistic computation models by exposing tentative results to external viewers. ASR addresses the vulnerability of potentially unrecoverable side effects by serving as a generic dispatcher for automatic compensating actions.

Manual Repair As a last resort, a transaction can choose to be manually resolved when invalidated. We use the term *repair* to refer to the manual conflict resolution process. It also serves as the fallback option when other options fail. If an abnormal condition or an erratic resolver results in failed automatic resolution of an invalidated transaction, the users will be requested to manually repair the transaction. The IOT mechanism provides a repair tool to assist the users to inspect the local and global states of the relevant objects and to create the resolution result.

Transaction Consistency Specification Before execution starts, a transaction must specify which criterion (G1SR or GC) to use for consistency validation when disconnected and which resolution option to use for conflict resolution when invalidated. If the resolution option is ASR, the user must also provide information about the resolver. We call such information about the selection of consistency criterion and resolution option the *transaction consistency specification*. It can be supplied to the transaction system through the arguments of both `begin_iot` and `end_iot` calls.

3.3.4.4 Implementation Restriction

We decided to implement only the GC consistency validation in this research because the implementation complexity of G1SR far outweighs its practical value for the following reasons. First, in the implementation framework proposed by Davidson [9], validating G1SR for disconnected transactions requires the construction of a global graph data structure, which is likely to hurt system availability and scalability. Furthermore, it requires the servers to maintain a complete history of committed transactions from the beginning of a disconnection. Given the fact that a portable client can be disconnected for a long period of time, the server space cost for recording transaction histories can be prohibitive.

Second, although G1SR is a much more general consistency framework than GC and less susceptible to false-negative validation, the combination of GC and application-specific revalidation capability can adequately offset the loss of G1SR's generality. Our experience suggests that GC is better suited for the consistency needs of using disconnected operation for mobile file accesses. Moreover, validating GC for disconnected transactions can be implemented much more efficiently with minimum server resource cost, which is very important for preserving overall system scalability.

Finally, although an implementation of G1SR is outside the scope of this dissertation, the current IOT design and implementation are fully compatible with future support for it.

3.3.5 Handling Non-Transactional Operations

Because IOT is an optional file system facility intended to be used for selected applications, there will be plenty of non-transactional file access operations, i.e., operations that are not within the scope of any IOTs. The IOT model needs to provide a semantic specification for non-transactional file access operations that satisfies the following two criteria. First, the original IFT semantics must be preserved for those non-transactional operations that do not interfere with any IOTs (i.e., do not share objects with any IOTs). Second, the semantics of those non-transactional operations that interfere with other IOTs must fit smoothly with the IOT consistency model.

We adopt a uniform semantic specification that regards each individual non-transactional file access operation as a special IOT containing only one operation and using manual repair as its conflict resolution option. Obviously, the behavior of any non-transactional operations under this semantic model is guaranteed to be the same as that under the IFT model, as long as they do not interfere with IOTs. This is very important to our goal of maintaining upward Unix compatibility. The behavior of other non-transactional operations is much the same, except that they have to obey the concurrency control requirements when interacting with IOTs. For example, a non-transactional operation trying to update an object `obj` must wait until the ongoing IOT that has already been accessing `obj` finishes.

3.3.6 Model Optimization

In an effort to improve availability (i.e., to enlarge the set of admissible partitioned file access operations), Kistler proposed a number of innovative optimizations for the IFT model [26]. The IOT model inherits many of those optimizations for compatibility reasons and extends them to a larger transaction granularity.

Basic Operation	Readset	Writeset	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. This table is taken from Kistler's dissertation[26].

Table 3.2: Transaction Specification for File Access Operations

Employing Sub-Object Granularity Intuitively, Unix programmers and users regard file access operations to operate generally on the granularity of an individual object, i.e., an entire file, directory or symbolic link. But such a coarse granularity may cause many unnecessary conflicts. For example, partitioned `chmod` and `chown` on the same object would appear to be in update/update conflict, but the conflict is false because the two operations touch different parts of the object.

One way to avoid such false conflicts is to adopt a finer granularity of transaction specification so that different sub-parts of an object are treated as independent logical entities. With each object, we associate the following set of attributes: `fid` (the internal identifier of the object), `owner`, `mtime`[4], `mode`, `linkcount`, `length` and `rights`[5]. In addition, each object will also have a `data` field. For a regular file and symbolic link, `data` is just a `length` long byte sequence. For a directory, the `data` field is regarded as a set of <`name`, `fid`> pairs where any name can occur only once.

Because the intuitive view held by Unix users and programmers is that the <`name`, `fid`> bindings are basically independent of each other, we model the directory content as a fixed-size array indexed by all the possible legal names (there are 256^{256} of them). If a `name` is bound to a `fid` in a directory, the array element `data[name]` contains that `fid`. For any other unbound `name`, `data[name]` contains a special value `nil`. This directory model would permit a partitioned pair of intuitively independent operations such as "`mkdir joe/foo`" and "`mkdir joe/bar`".

Eliminating Non-Critical Side Effects The standard UFS semantics require certain time attributes such as *access time* and *modification time* to be maintained as a side effect of some file system calls. Maintaining the strict read/write semantics for such time attributes would severely limit the amount of legal partitioned activity admissible by the model. Because of the low information content and limited importance of those attributes, we eliminate attributes such as access time from transaction specification. The `mtime` attribute is retained but it can only be updated by the `utimes` system call. Such minor semantic redefinition significantly improves the usability of the model and the Coda experience indicates that the semantic changes have very little impact in practice [26].

Exploiting Type-Specific Semantics The `linkcount` attribute represents the number of directory bindings that refer to an object. The `length` attribute for a directory stands for the cumulative length of all the names that are currently bound in the directory content. Both of them are updated as side effects of operations such as `link` and `mkdir`. We cannot afford to

[4]The modification time stamp of the object.
[5]The access control list for directory object.

maintain strict read/write semantics for these two attributes because it would forbid the above pair of operations since they both need to update the two attributes. The solution to this problem is to exploit the *counter semantics* of these attributes. Because the results of operations such as `link` and `mkdir` are propagated by replaying the same operations in another partition, the value of the counters, i.e., `linkcount` and `length`, are incremented and/or decremented accordingly instead of being *copied*. Therefore, by expanding the transaction specification with explicit `increment-set` and `decrement-set` and employing counter semantics instead of read/write semantics on the two sets, serializability can be maintained while still allowing the above pair of operations.

The resulting transaction specification for all the basic file access operations after the above three optimizations is listed in Table 3.2. For an individual file access operation `op`, we use the notation `R(op)`, `W(op)`, `I(op)` and `D(op)` to denote its readset, writeset, increment-set and decrement-set. Similarly, for an isolation-only transaction `T`, the notations `R(T)`, `W(T)`, `I(T)`, `D(T)` refer to the same sets for `T`. If the set of file access operations performed by `T` is $\mathtt{OPS(T)} = \{\mathtt{op}_1, \mathtt{op}_2, ..., \mathtt{op}_n\}$, then the readset of `T` is just the union of the readsets of all the involved operations, i.e., $\mathtt{R(T)} = \mathtt{R(op_1)} \cup \mathtt{R(op_2)} \cup .... \cup \mathtt{R(op}_n\mathtt{)}$. The definitions for `W(T)`, `I(T)` and `D(T)` are similar.

Adopting Weak Consistency For Read-only Transactions The IOT consistency model makes a special case for read-only transactions, or *queries*. Read-only transactions containing multiple objects can affect the overall serializability for involved transactions in a subtle way, as explained in Figure 3.4. Previous research indicates that in many cases, it is preferable to allow queries to proceed rather than to restrict them merely to satisfy strict 1SR in all situations [17]. The IOT model allows read-only transactions to be *weakly consistent*, meaning that they each see only a consistent state, i.e., the result of a 1SR execution of update transactions, but they may not be 1SR with respect to each other.

In the context of a distributed file system, adopting weak consistency for queries is quite an acceptable design tradeoff. It is a small and controlled departure from the strict correctness criterion. One-copy serializability is still enforced for all update transactions. On the other hand, the availability gain is quite significant because queries are a large fraction of inferred transactions in a typical file system environment [26].

3.3.7 Closing Remarks

Now that most of the features of the IOT computation model have been presented, we offer further observations on several key aspects of the model. First, we identify the key characteristics of file system state when IOTs are used. Second, we examine the capability of the IOT model in solving the data inconsistency problems caused by partitioned read/write conflicts.

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

This execution is not 1SR because `T2` requires an ordering where `T1` precedes `T3`, and `T4` requires one where `T3` precedes `T1`.

Figure 3.4: Read-Only Transactions Violating One-Copy Serializability

Third, we analyze the relationship among different semantic models used in designing the IOT consistency model.

Asymmetric System State In a distributed file system with disconnected operation but without server replication, the entire system is divided into two kinds of partitions with distinct consistency properties. The first kind of partition consists of a group of inter-connected servers and clients and is called a *first class partition*. The servers in a first class partition maintain the home repository for a portion of the file system name space. The other kind of partition consists of only one disconnected client and is called a *second class partition*. System failures such as disconnection and node crash will break up a first class partition into smaller first class partitions and/or second class partitions. Recovery from those failures will merge smaller partitions into larger ones.

The IOT model regards the portion of the system state maintained at any first class partition to be of high quality and always in a consistent state. Any second class partition containing disconnected computation results are considered of lesser quality. All disconnected computations are regarded as tentative, being only locally consistent within themselves. Their validity with respect to the state maintained on the corresponding servers is suspect. This asymmetric consistency view of system state is largely independent of the IOT consistency model. Instead, it is the combined result of the nature of disconnected operation and the OCC-based IOT execution model, both of which make a strong distinction between the roles of a client and a server.

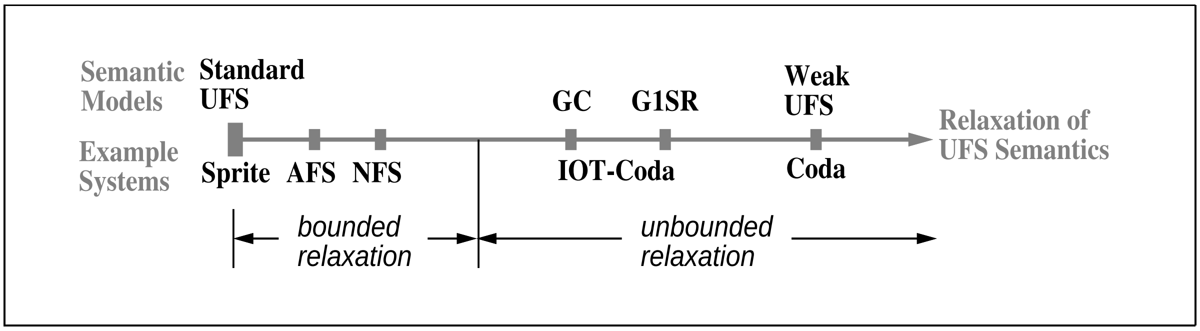
We use the term *global state* to refer to the server state of the portion of the name space maintained at a first class partition, and *local state* to refer to the portion of the name space that is visible from a second class partition. Both the global and local states satisfy their own serializability-based consistency requirements. At any given moment, a global state is the result of one-copy serializable execution of a set of transactions. A local state is the combination of a previously 1SR consistent state (i.e., the global state that a disconnected client inherited at the start of disconnection) and the result of a serializable execution of local disconnected transactions.

IOT Model Capability The IOT consistency model itself constitutes a general purpose computation model offering a flexible set of serializability-based isolation guarantees, whether it is enforced pessimistically under a connected environment or optimistically for disconnected operation. It is adequate to ensure the consistency of a large variety of Unix applications under common circumstances. We analyze the capability of the IOT consistency model with GC as the consistency validation criterion from two different angles.

First, we compare the IOT consistency model against the general purpose 1SR consistency model. The capability of IOT and 1SR can be assessed by comparing the set of transaction execution histories admissible by either model, denoted as H_{IOT} and H_{1SR} respectively. Obviously, based on pure syntactic recognition power, H_{IOT} is only a subset of H_{1SR} because some one-copy serializable histories are not recognizable by the IOT consistency model for two main reasons. First, the GC consistency criterion will reject many partitioned transaction histories that are 1SR. Second, the OCC-based execution model also renders some legal histories in H_{1SR} impossible to realize because partial transaction execution results are not visible to transactions executing on other clients. However, the ASR resolution option allows application semantics to enlarge the set of legal histories for the IOT model, making it possible for the IOT consistency model to admit even more histories than H_{1SR}.

Second, we evaluate the capability of the IOT consistency model from the viewpoint of maintaining compatibility with the standard UFS semantics, i.e., the ability to detect inconsistencies for applications relying on standard UFS semantics instead of general serializability to operate correctly. Thanks to the restrictiveness of the GC criterion, the model is capable of detecting all instances of stale read when the read operation is performed on a disconnected client. Generally speaking, the IOT model is capable of detecting half of all the possible non-1UE behaviors caused by disconnected operation where the stale read is performed on a disconnected client. It is not capable of detecting inconsistencies resulting from stale reads that are performed on connected clients because the asymmetric consistency maintenance model always allows connected computations to commit immediately. Overall, the IOT consistency model meets our goal of safeguarding the integrity of mobile file access using disconnected operation.

Relationship Among Semantic Models At center of the IOT consistency model design is the selection of an alternative semantic model that can be used to bridge the semantic gap between standard UFS and weak UFS. Such a model serves as the criteria for validating disconnected computation results by requiring certain properties to be satisfied by partitioned file access operations. The explicit transaction extension of IOT provides the underlying file system with information about file access groups bracketed by transaction boundaries. This provides the opportunity to impose a variety of serializability-based requirements on partitioned transaction executions. Both the GC and G1SR semantic models used by IOT are capable of addressing the consistency needs of mobile file access under various system and usage environments. Enforcing the GC or G1SR requirements when partitions are healed enables the system to detect situations where standard UFS semantics is violated and data becomes inconsistent.



The concept of bounded and unbounded relaxation of standard UFS is previously discussed on page 10.

Figure 3.5: Relationship Among Semantic Models

Figure 3.5 depicts a spectrum of semantic relaxations of standard UFS. Above the gray arrowed line are various semantic models representing different degrees of relaxation. The further to the right, the bigger the semantic gap. Note that it is quite possible to relax standard UFS even further than weak UFS (e.g., optimistic replication without detection of update/update conflicts). To formally discuss the relationship among these semantic models, we define a *stronger than* relation between two semantic models S_1 and S_2. S_1 is stronger than S_2 (or S_2 is *weaker than* S_1) if any admissible computation in S_1 is also admissible in S_2. Obviously, both GC and G1SR are stronger than weak UFS because they put additional constraints on partitioned file access operations. GC is stronger than G1SR because any transaction that satisfies GC also satisfies G1SR. However, both GC and G1SR are weaker than standard UFS because any violation of GC or G1SR involves at least one stale read (or diverging write), which also violates standard UFS. In addition, GC and G1SR are only an approximation of standard UFS because there are situations such as stale reads performed on a connected client that are admissible by GC and G1SR but are not admissible by standard UFS.

Below the gray arrowed line in Figure 3.5 are examples of actual distributed file systems that implement the corresponding semantic models above the line. For example, standard UFS and weak UFS are realized by Sprite and Coda respectively; GC and G1SR are supported by IOT-Coda (Coda with IOT extension); both AFS and NFS represent different instances of bounded relaxation of standard UFS. Note that the semantic gap between standard UFS and GC (or G1SR) is unbounded because applications are not capable of obtaining the standard UFS semantics by using the IOT-extended UFS API.

Chapter 4

Detailed Design: Consistency Enforcement

This is the first of the five consecutive chapters devoted to the detailed design and implementation of an IOT extension to the Coda file system. The central theme of chapters 4, 5 and 6 is how to realize the IOT model in Coda, which consists of two main parts. The first part is enforcing transaction isolation within a partition, i.e., 1SR for connected transactions within a first class partition and LSR for disconnected transactions within a second class partition. The second part is ensuring global transaction isolation when propagating transactions from a reconnected client to the corresponding servers.

The first section of this chapter discusses the concurrency control issues in guaranteeing 1SR for connected transactions. In addition to discussing enforcing LSR for disconnected transactions, the second section also describes how to maintain a local transaction history on a disconnected client in preparation for the future consistency validation and transaction commitment/resolution. Finally, the third section presents an incremental transaction propagation scheme that propagates (i.e., validates, commits or resolves) transactions one at a time on a reconnected client.

In the rest of this document, all the discussed design and implementation issues are newly introduced in the Coda file system to support the IOT extension. Some of the mechanisms necessary for supporting IOT take advantage of existing facilities in the original Coda file system. They will be explicitly pointed out to make a distinction between what has originally been done in Coda and what is the new extension to support IOT.

4.1 Concurrency Control for Connected Transactions

This section concentrates on realizing the first part of the IOT consistency model, guaranteeing 1SR for connected transactions. In a connected environment, the weak UFS semantics implemented in the Coda file system delivers the same semantic guarantees as the standard UFS. Therefore, enforcing 1SR for connected transactions as required by the IOT consistency model becomes an issue of concurrency control. We first analyze the design alternatives and then present the detailed design of our choice, the OCC model.

4.1.1 Design Alternatives

A wide range of concurrency control techniques have been developed for enforcing 1SR among concurrent transactions. However, not all of them are legitimate design candidates because the OCC-based IOT execution model described in Chapter 3 precludes the possibility of transactions executed on different clients reading each other's partial results.

Lock Based Approach The most commonly used concurrency control method is using locks to coordinate accesses on shared objects among concurrent transactions. As the representative of this approach, *two phase locking* (2PL) and its variations can be found in plenty of commercial database systems. Although it is possible to employ distributed locking to ensure 1SR for connected transaction execution in Coda, there are many disadvantages. Among other things, managing distributed locks in a large scale system with unpredictable disconnections is very difficult and costly. In addition, standard locking protocols such as 2PL are not suitable for IOT implementation because they are known to be susceptible to poor performance for long running transactions [4, 2] and Unix applications such as `make` often access a large number of objects and last a long time.

Time Stamp Based Method Another major class of concurrency control algorithms associates each transaction with a unique time stamp, and attaches time stamp information to every object accessed by ongoing transactions. Serializability is achieved by enforcing time stamp order for all conflicting file access operations issued by concurrent transactions [5]. Its main disadvantage is that the time stamp information attached to objects must be immediately updated on the servers for every file access operation to globally synchronize concurrent transactions. This will greatly increase the client/server communication traffic and negate much of the performance benefit of client caching. In addition, maintaining time stamp information[1] on each object also incurs significant server space cost.

[1] The content of such information depends on the specific concurrency control algorithm and typically includes the time stamps of the latest transactions that have read and written the object.

Optimistic Concurrency Control Our choice of the concurrency control method for connected transaction execution is the OCC model [33]. It is a natural fit because the IOT execution model is based on OCC. In OCC, transaction execution is performed within a private workspace. At the end of the execution, a transaction is validated against the public space to see if its private execution result can be serialized with previously committed transactions. If so, the transaction's private result is committed to the public space. Otherwise, it is thrown away and the transaction is automatically re-executed.

There are several advantages for using OCC as the IOT concurrency control algorithm in a connected environment. First, previous studies indicate that OCC offers strong performance in systems such as Coda where the likelihood of data contention is very low [71, 6, 75]. Second, it can be implemented highly efficiently in Coda by simply recording and comparing the object version information already maintained [61], thus incurring almost no extra resource cost. Third, it fits well with Coda's highly scalable architecture because the essence of OCC in Coda is trading client transaction re-execution for the global communication needed for synchronizing concurrent accesses on shared objects. In a large scale distributed file system, client computation time is a much less critical resource than global communication bandwidth.

4.1.2 Realizing OCC in Coda

Although conceptually simple, OCC has not been used in practical systems for concurrency control so far. The actual realization of OCC in Coda needs to address a number of important issues.

4.1.2.1 OCC Validation

The IOT execution model uses the client disk cache as the private workspace for transaction execution. The public space reflecting all the committed transactions is maintained on the corresponding servers. All file access operations issued by a connected transaction are performed locally. The main task of OCC validation is to check whether any of the data items that are accessed by the transaction have been updated on the servers during the execution. Because OCC validation is logically identical to the GC validation for disconnected transactions, it shares the same basic mechanisms used by the GC validation, which will be discussed in Section 4.3.4.

4.1.2.2 Committing Transaction Result

As required by OCC, the execution result of a connected transaction must be held within the client cache until it is successfully validated and then committed to the corresponding servers. The specific mechanisms used in performing transaction commitment are based on the client mutation logging and reintegration mechanisms of the existing Coda file system. The next two paragraphs describe these mechanisms. We then explain how they are extended to support committing the result of a connected transaction to the servers.

Client Mutation Log (CML) During disconnected operation, Venus maintains a separate client mutation log (CML) in persistent storage for each volume for which it has cached objects. A CML record is constructed and appended to the appropriate log whenever a mutation operation is performed locally. Each CML record contains sufficient information about a locally performed mutation operation so that it can later be replayed on the corresponding server. The only exception is the `store` operation whose CML record does not include the actual data content of the operation. Instead, it is supplied by the cache copy of the corresponding file.

Client Mutation Reintegration Logged client mutations are propagated to the corresponding servers on a volume by volume basis through Coda's reintegration mechanism. In the first phase, Venus packs the CML records of a volume into a buffer and transmits it to the corresponding server. During the second phase, the server locks the relevant volume, checks that none of the received mutation operations are in conflict with involved server replicas, and replays all the mutation operations in order. For a `store` operation on object `obj`, the server needs to back-fetch the cache copy of `obj` from the client because the data content is not included in the CML record. In the final phase, the server sends back the reintegration outcome and Venus accordingly updates the cache status of all involved objects. Note that the entire reintegration process is atomic, either all or none of the mutations in the CML are reintegrated.

Transaction Mutation Log (TML) We now describe how to extend the existing volume based CML to log mutations for connected transaction execution. Conceptually, all the mutation operations performed by a connected transaction `T` are recorded in its transaction mutation log denoted `TML(T)`. The physical organization of `TML(T)` utilizes the underlying CML data structures. To deal with the possibility of concurrent transactions mutating objects in the same volume, a new field is added to the basic CML record format to store the identifier of the transaction that issued the mutation operation. Every time a mutation operation is locally performed by transaction `T`, a new record containing `tid(T)`[2] is created and appended to the corresponding CML. As shown in Figure 4.1, the records of `TML(T)` may be scattered around different CMLs because `T` can update objects in multiple volumes.

Transaction Reintegration Logged mutations in `TML(T)` of a connected transaction `T` are committed to the servers by utilizing a generalized reintegration mechanism called the *basic*

[2]We use the notation `tid(T)` to refer the transaction-id of `T`.

reintegration process. Instead of taking the entire CML, the basic reintegration process takes as an input argument an already-packed mutation log composed of records from a subsequence of a CML and performs the second and third phases of the original reintegration mechanism. The transaction reintegration process commits the result of `T` by extracting its records from the relevant CMLs and applying the basic reintegration process for every volume `T` has updated. This is illustrated by the pseudo code in Figure 4.2.

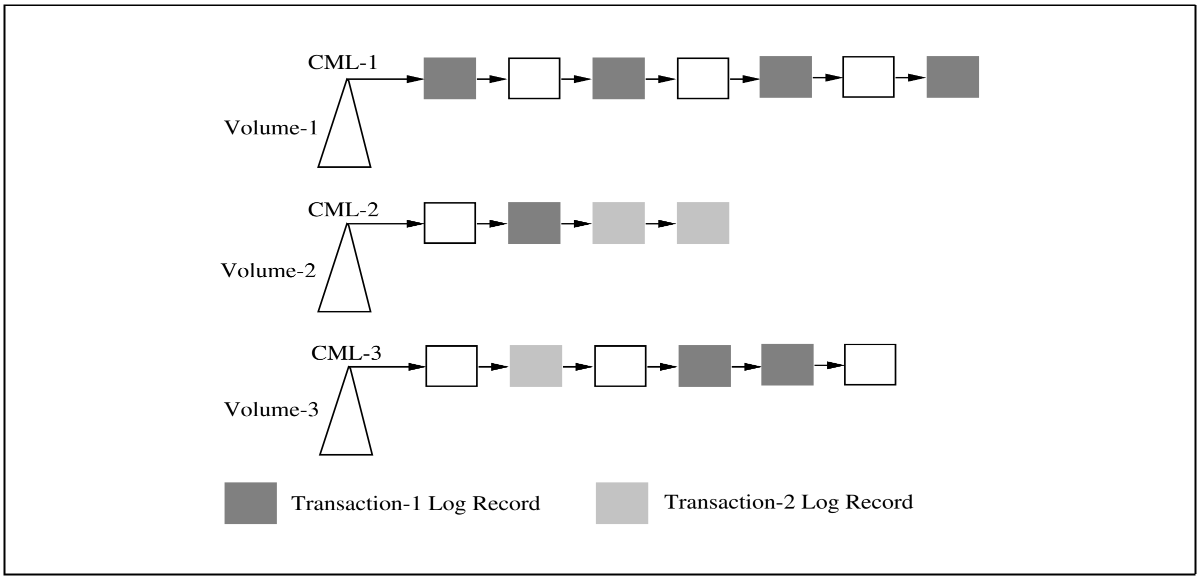


Figure 4.1: Transaction Mutation Log Organization

During the transaction reintegration process, failure atomicity is only guaranteed at the granularity of single-volume reintegration. To ensure full atomicity for transaction commitment as required by the IOT model, a distributed protocol such as *two phase commitment (2PC)* is needed [19, 5, 20]. The current IOT implementation has not included full failure atomicity for transaction commitment mainly because it is not necessary in practice most of the time. Volumes contain logically related objects and it is unlikely that an individual transaction will update objects in multiple volumes. In our limited experience of executing common software development and document processing tasks as transactions, we have yet to find one that mutates objects in more than one volume. Because 2PC is a mature technique in transaction processing, it will be straightforward to extend the current transaction commitment mechanism to include full 2PC support in a refined IOT implementation in the future.

```
Transaction-Reintegration(T) {
    foreach (volume V updated by transaction T) {
        extract TML(T) records from the CML of V;
        pack the records into mlog;
        apply the basic reintegration process on mlog;
        if (reintegration failed) break;
    }
}
```

Figure 4.2: The Transaction Reintegration Process

4.1.2.3 Automatic Abortion and Re-execution

When a transaction `T` fails validation, OCC requires its local result to be aborted and the transaction to be automatically re-executed. Aborting the local transaction result can be accomplished by throwing away all the `TML(T)` records and invalidating the cache copies of all the mutated objects so that they can be re-fetched from the servers during transaction re-execution. Since automatic transaction re-execution is also employed by the IOT model as one of the basic conflict resolution options, we defer detailed discussions until Chapter 6.

One problem caused by the automatic OCC re-execution is the visibility of repeated transaction side effects. Many Unix applications include external side effects such as displaying messages on terminals and they will re-appear during the OCC re-execution and confuse users. As Pausch has shown [54], including external effects in the transactional model is a very difficult problem with no easy solutions. The current IOT implementation alerts the user by printing out a warning message "... is being re-executed because of data contention" immediately before the OCC re-execution takes place.

4.1.2.4 Local Concurrency Control

The OCC validation only takes care of concurrency control among transactions executed on different clients. We need a second level concurrency control mechanism because the client cache is not a true private workspace for transaction execution. Concurrent connected transactions executed on the same client can access shared objects in the same client cache. They must be synchronized so that the same physical client cache can serve as a logical private workspace for each of them.

We decided to employ the strict two phase locking protocol for local concurrency control. The main reason for selecting strict 2PL is its simplicity and a proven record of being able to deliver reasonable performance when data sharing among concurrent transactions is infrequent [2]. Because Coda clients are typically operated by a single user, the likelihood of executing concurrent transactions involving heavy read/write sharing is low. Under such an environment, implementation simplicity makes 2PL the most suitable choice as the local concurrency control algorithm.

4.1.2.5 Deadlock and Livelock

Deadlock Detection In local concurrency control, it is possible for a group of transactions executing on the same client to deadlock because of the 2PL protocol. The traditional techniques for deadlock prevention are not applicable because we have no control over the order of file access operations issued by a transaction. We therefore employ the standard deadlock detection technique of maintaining a *wait-for graph*, denoted `WFG`. Every running transaction `T` corresponds to a node in `WFG` denoted w_T. When a transaction `T` needs to wait for another transaction `T'` due to 2PL, an edge from $w_{T'}$ to w_T is inserted into `WFG`. Deadlock detection is performed by a periodic daemon checking for cycles in `WFG`. If `WFG` becomes cyclic, it means that the corresponding transactions on the cycles are deadlocked. When that happens, the transaction system will print out messages notifying the users that this group of transactions are deadlocked and some of them must be killed in order to make progress. Note that the user must explicitly kill transactions. The transaction system does not do this automatically.

Livelock Prevention On the other hand, cross-client OCC concurrency control can suffer from the *livelock* problem. When there is heavy data contention, it is possible for a transaction to follow an endless cycle of "execution → invalidation → abortion → re-execution". In other words, a transaction can suffer from *starvation* and never be able to commit its result because there are endless conflicting transactions executing on other client machines. Because the likelihood of livelock is extremely low in our target environment, our approach is to detect this situation and notify the users. We do this by attaching a re-execution counter to each ongoing transaction. When the counter exceeds a certain threshold, we force termination of the transaction and inform the users that there is too much data contention and that this transaction should be tried at a later time.

4.1.2.6 Summary

A significant portion of the IOT consistency model, the guarantee of 1SR for connected transactions, is realized using a combination of inter-client OCC and intra-client 2PL. The key insight is the recognition that OCC fits well with Coda's client/server architecture and target usage environment to deliver high performance and scalability, and that the client disk cache can serve as the transaction processing workspace as required by OCC. The implementation of OCC in Coda relies on extending Coda's existing client mutation logging and reintegration mechanisms for performing local transaction execution and committing transaction results to the servers.

4.2 Maintaining the Local State of A Disconnected Client

The transaction system running on a disconnected client has two major responsibilities. First, it needs to fulfil the IOT consistency model's local consistency requirement of enforcing LSR for disconnected transactions executed on the same client. Second, it must maintain a local history of disconnected transactions and manage disconnected mutations in preparation for future transaction propagation (i.e., validation followed by commitment/resolution). The latter is critical to realizing the central focus of the IOT consistency model, guaranteeing global transaction isolation when propagating disconnected transactions from a reconnected client to the servers. This section focuses on four key issues:

- maintaining local consistency (1SR for disconnected transactions)
- recording disconnected transaction history information
- managing disconnected mutations to support transaction propagation
- cancelling redundant disconnected transactions to reduce resource cost

4.2.1 Maintaining Local Consistency

The IOT consistency model ensures the consistency of the local state of a disconnected client by requiring disconnected transactions to satisfy local serializability. At any moment during a disconnected operation session, the result of all the disconnected transactions must be equivalent to a serial execution of the same set of transactions. Thus, the issue of local consistency maintenance becomes a matter of concurrency control among disconnected transactions. Since a local concurrency control mechanism using 2PL is already employed for connected transaction execution, we reuse the same mechanism for disconnected transaction execution based on the same design rationale. Similarly, deadlock among concurrent disconnected transactions is detected by maintaining the `WFG` and periodically checking for cycles.

4.2.2 Recording Transaction History

A critical step in transaction propagation is consistency validation, which requires the transaction system running on a disconnected client to record history information about disconnected transaction executions. We use the following two persistent data structures.

Transaction Table A client-wide table is used to keep track of all *live* transactions, i.e., transactions that are not yet committed or resolved. Conceptually, each table entry stores the internal representation of a transaction, recording the transaction readset and writeset among other things. Each element of the readset or writeset corresponds to a Coda object that is read or written by the transaction respectively. It contains the necessary information needed by transaction validation such as the internal identifier of the object (`fid`) and a description about the sub-parts of the object that are actually accessed by the transaction. Because the IOT model uses weak consistency for read-only transactions, the table entries for read-only transactions are immediately discarded after their execution is completed.

Note that we do not record the transaction increment-set and decrement-set because their main purpose in transaction specification is to propagate updates on the counter attributes and they do not affect the outcome of transaction validation based on readset and writeset. In addition, transactions are committed to the servers via the reintegration mechanism which replays disconnected mutations on the servers and includes the effect of incrementing/decrementing counter attributes.

Serialization Graph An important part of local transaction history is the inter-dependency among live transactions. It is recorded by a data structure called the *serialization graph*, denoted `SG`, where each node corresponds to a transaction (either IOT or IFT). For transactions `T` and `T'`, we use n_T and $n_{T'}$ to denote their corresponding nodes in `SG`. There is an edge from node n_T to $n_{T'}$, denoted $n_T \rightarrow n_{T'}$, if and only if transactions `T` and `T'` performed a pair of conflicting operations and `T` did its operation before `T'`. This means that transaction `T` must be serialized before transaction `T'`. Because of the 2PL local concurrency control, `SG` is guaranteed to be acyclic and defines a partial order among all disconnected transactions. A total order can be obtained by a topological sort on the nodes of `SG`.

To avoid the performance overhead of checking every file access operation, we adopt the following strategy for `SG` maintenance. A new node is created and inserted into `SG` each time a transaction starts execution. Some `SG` edges are constructed as a by-product of the 2PL concurrency control. For example, when a transaction `T` needs to wait for another transaction `T'`, we can safely add an edge $n_{T'} \rightarrow n_T$ to `SG`. Other edges for the node n_T are created at the conclusion of `T`'s execution. All we have to do is to check all the pending transactions. If a pending transaction `T'` performed an operation that conflicts with `T`, then an edge $n_{T'} \rightarrow n_T$ is inserted into `SG`. The node n_T is removed from `SG` and discarded as soon as transaction `T` is committed or resolved.

4.2.3 Managing Disconnected Mutations

During transaction propagation, a successful consistency validation means the result of a disconnected transaction can be committed to the servers as is. This requires the transaction system running a disconnected client to carefully manage disconnected mutations so that the transaction result can be written to the servers using the transaction reintegration infrastructure discussed earlier. Two key data structures needed for this task are the transaction mutation log and the shadow cache file.

4.2.3.1 Transaction Mutation Log

As in connected transaction execution, all the mutations performed by a disconnected transaction `T` are recorded in its transaction mutation log `TML(T)`. In addition, transaction mutation logs are organized using the same underlying CML data structures and maintained in a similar manner. Every time a disconnected mutation operation is performed locally, a log record for that operation is created and appended to the CML of the corresponding volume. The transaction-id field of the new record stores the identifier of the transaction (whether it is an IOT or an IFT) that issued the mutation operation. However, there is a major difference between mutation logging for connected transactions and disconnected transactions. For a connected transaction, all its logged mutations are prevented from being overwritten by other transactions because of the local 2PL. But, the same *cannot* be said for disconnected transactions. This difference requires the use of shadow cache files for reasons explained below.

4.2.3.2 Shadow Cache File

The Need to Supplement TML The main reason for supplementing the TML is the transaction system's need to represent the complete mutation results of individual disconnected transactions so that they can be independently committed to the servers. For a `store` operation performed on a file object `obj` by transaction `T`, its data content is not recorded in `TML(T)` but instead provided by the cache copy of `obj` denoted `cache(obj)`. Thus, it is the combination of `TML(T)` and `cache(obj)` for every `obj` stored by `T` that represents `T`'s complete mutation effect. TML(T) is adequate when `T` is a connected transaction because the local 2PL prevents `obj` from being overwritten until `T` is committed. However, if `T` is a disconnected transaction, it is possible for `obj` to be overwritten by a later transaction `T'`. In such a situation, `cache(obj)` no longer holds the value written by `T`. Therefore, the transaction system must use an additional mechanism to record the value that `T` wrote on `obj` so that the complete mutation effect of `T` is still retained.

Compensate TML With Shadow Cache File Our strategy is to maintain shadow copies for the overwritten cache files. If a file object `obj` is updated by a pending transaction `T` and is about to be overwritten by another transaction `T'`, the transaction system will automatically create a shadow copy for `cache(obj)`. We use the notation `shd(T,obj)` to refer to the shadow cache file that contains the data content of `obj` that was written by transaction `T`. Because `T` can have more than one overwritten file, we use `shdset(T)` to denote the complete set of shadow cache files for transaction `T`. The data content of any `store` operation on a file object `obj` performed by `T` is guaranteed to be preserved by either a shadow cache file in `shdset(T)` or the current `cache(obj)`.

With the aid of `shdset(T)`, the previously introduced transaction reintegration process can be applied to commit the local result of transaction `T` with slight modifications on the Venus side as described by the pseudo code in the Figure 4.3. The temporary switch of cache file bindings is necessary for the servers to back-fetch the correct data content during reintegration for those locally overwritten `store` operations performed by `T`. An internal Venus volume locking mechanism is employed to prevent the temporary binding between `obj` and `shd(T,obj)` from being inappropriately exposed to the users and applications.

Other Uses of Shadow Cache Files The use of shadow cache files goes beyond supporting independent reintegration of individual transactions. During conflict resolution for an invalidated transaction `T`, the IOT system must preserve sufficient information about `T`'s original execution so that the resolver can investigate what has been performed by `T` locally. As will be explained in detail in Chapter 5, the IOT conflict representation provides a *snapshot original view* for all the objects accessed by `T`. For any file object `obj` accessed by `T`, this view presents the data content of `obj` when it was last read or written by `T`. Even if `obj` is only read by `T`, we still need to make a shadow copy of `cache(obj)` before it is updated to preserve the data content that `T` originally read. In general, for any file object `obj` accessed by at least one pending transaction, a shadow copy of `cache(obj)` needs to be created whenever it is about to be updated.

Shadow Cache File Maintenance Because the same shadow cache file can be shared by multiple disconnected transactions, the management of shadow cache files requires the transaction system to maintain a pool of *shadow entries*, an internal representation of shadow cache files. Each shadow entry contains two components: a pointer to the *inode* of the disk container file that physically holds the shadow content and a counter that records the number of live transactions that have accessed the shadow content.

Every time a cached file object `obj` is opened for update (write, append or truncate) while disconnected, the transaction system must check to see if any pending transaction has accessed the current contents of `obj`. If there are K pending transactions that have accessed the current `cache(obj)`, a new shadow entry for `obj` denoted `se(obj)` is created and its counter is initialized to K. The transaction system will allocate disk space, create a new container file `F`, copy `cache(obj)` into `F`, and set the file pointer of `se(obj)` to point to `F`. Finally, each of the K pending transactions must extend its `shdset` to include the newly created `se(obj)`. Whenever a pending transaction `T` is committed or resolved, the transaction system will iterate through every shadow entry in `shdset(T)` and decrement its counter. When the counter of a shadow entry reaches zero, the shadow entry is discarded after the corresponding disk container file is removed to reclaim the disk space.

```
New-Transaction-Reintegration(T) {
    foreach (volume V updated by transaction T) {
        extract TML(T) records from the CML of V;
        pack the records into mlog;
        lock volume V;
        foreach (overwritten "store obj" in mlog) {
            preserve cache(obj);
            bind obj to shd(T,obj)∈shdset(T);
        };
        apply the basic reintegration process on mlog;
        foreach (overwritten "store obj" in mlog) {
            re-bind obj to preserved cache(obj);
        };
        unlock volume V;
        if (reintegration failed) break;
    }
}
```

Figure 4.3: The New Transaction Reintegration Process

A challenging problem in shadow cache file maintenance is client disk space management. Maintaining shadow copies for large files that are repeatedly overwritten during disconnected operation could cost a substantial amount of local disk space. Our disk space allocation policy adopts two basic strategies. First, shadow cache files are maintained on the *best-effort* basis, i.e., shadow copies are created only when sufficient disk space is available for supporting normal disconnected activities. When the transaction system runs out of shadow space, the reintegration or resolution process for the affected transactions will be performed differently. The specific details are presented in the relevant discussions later in this chapter and the next two chapters. Second, we allow the users to dynamically adjust the limit on the amount of disk space that can be allocated for shadowing cache files. Fortunately, the typical shadow space usage for normal disconnected operation is small even when the disconnection lasts for a long time, as will be shown by the evaluation results in Chapter 9. The impact of the best-effort shadow space allocation policy on other aspects of the transaction system will be analyzed when appropriate in rest of the document. The implementation details of shadow space management are presented in Chapter 8.

4.2.4 Cancelling Disconnected Transactions

Maintaining disconnected transactions costs client local resources in two main areas: persistent memory space for recording transaction history and disk space for storing shadow cache files. To support long-lasting disconnected operation sessions involving a large number of disconnected transactions, it is necessary for the transaction system to reduce such client resource costs. Our strategy is to cancel those redundant transactions that no longer have any impact on the file system state. Note that transaction cancellation is only an optimization in realizing the IOT model in Coda, which does not change the overall IOT consistency model.

An important phenomenon in disconnected operation is that many file access operations cancel the effect of previous ones. For example, in the typical "edit → compile → debug" cycles during software development, a `store` operation often overwrites a previous one and a `remove` operation is likely to offset an earlier `create` operation. This is exploited by the non-IOT Venus (i.e, the Coda Venus without the IOT extension) to cancel unnecessary records from the CML during disconnected operation, and it results in significant resource savings [26]. The IOT implementation extends the same principle to a larger granularity to cancel those pending transactions that no longer have any effect on local client state. Such transaction cancellation has two important benefits. First, it frees up client resources such as persistent storage space used by the transaction internal representation and the disk space occupied by shadow cache files. Second, it reduces the amount of server/client communication traffic as well as server computation time needed for transaction validation and commitment. Evaluation results presented in Chapter 9 will show that these benefits are substantial.

Basic Mutation Cancellation Behaviors To understand how a pending transaction can be cancelled, let us first consider two kinds of basic mutation cancellation behaviors. The first kind of mutation cancellation is called *overwriting* which means that the effect of a mutation operation is *eliminated* or *made obsolete* by a later mutation operation. For example, the effect of "`store foo`" can be overwritten by a later "`store foo`" or "`remove foo`". The second kind of mutation cancellation is called *offsetting* which means that the effect of a pair of mutation operations offset each other so that their combined result produces no effect at all, i.e., a `no-op`. For example, a "`create foo`" can be offset by a later "`remove foo`" and a "`rename foo bar`" can offset a previous "`rename bar foo`". The basic principle behind mutation cancellation is that when the effect of a disconnected mutation operation op_1 is eliminated by a later operation op_2, there is no point of preserving op_1 in the mutation log and propagating its effect via reintegration. This is because after op_1 is replayed on the corresponding server during the reintegration, the subsequent replay of op_2 by the same reintegration process will immediately nullify op_1's effect.

Transaction Cancellation Criteria Intuitively, cancelling a pending transaction must preserve certain correctness conditions and we adopt the following three criteria to decide whether a pending transaction can be cancelled or not.

- The first criterion requires that a pending transaction `T` must be *obsolete* before it can be cancelled. `T` is an obsolete transaction if none of its effects are visible on the client local state due to subsequent transactions. For example, a `make` transaction compiling a file `work.c`, i.e., $\mathtt{T}_{make}$ =$\{$`R(work.c), W(work.o)`$\}$, can be made obsolete by another `make` transaction performing the same compilation. As another example, a pair of transactions $\mathtt{T}_{create} = \{$`create foo, create bar`$\}$ and $\mathtt{T}_{remove}$ $= \{$`remove foo, remove bar`$\}$ are both made obsolete by each other. In essence, this criterion ensures that the cancellation of `T` does not affect the final outcome when all the disconnected transactions are committed to the servers.

- The second criterion requires any cancellable transaction to be *covered*. It means that the removal of a transaction `T` from the local transaction history will not affect the validation outcome of other disconnected transactions. Although an obsolete transaction `T` does not leave behind any visible effects, it is still capable of influencing the validation outcome of other pending transactions. Consider the following example. A transaction $\mathtt{T}_1 = \{$`R(work.c), R(work.h), W(work.o)`$\}$ compiled an object file `work.o` that is later used by another transaction $\mathtt{T}_2 = \{$`R(work.o), W(work)`$\}$ to build an executable file `work`. A third transaction $\mathtt{T}_3 = \{$`R(work.c), R(work.h), W(work.o)`$\}$ re-compiled `work.o` after some updates are made to `work.c` and rendered $\mathtt{T}_1$ obsolete. However, cancelling transaction $\mathtt{T}_1$ will remove the indirect dependency between transaction `T2` and files `work.c` and `work.h` from the local transaction history, thus possibly affecting the validation outcome of $\mathtt{T}_2$. If `work.h` is updated on the servers, transaction $\mathtt{T}_2$ will pass the GC validation and commit to the servers even though its result indirectly depends on the old version of `work.h`. Preserving $\mathtt{T}_1$ in the local transaction history is necessary to allow the transaction validation process to follow the dependencies and appropriately invalidate $\mathtt{T}_1$, and thereafter $\mathtt{T}_2$.

- Even if a transaction `T` is both obsolete and covered, it does not necessarily mean that we can cancel it. Suppose that `T` contains just one operation `mkdir home/foo` and it is made obsolete by another transaction `T'` containing `rmdir home/foo`. Both `T` and `T'` must be cancelled together because cancelling `T` alone will cause failure in propagating `T'`. Formally, when one of `T`'s mutation operations offsets another belonging to transaction `T'`, we say that the two transactions have an *offsetting relation* between them and the two must be cancelled together. This is because cancelling either of them while leaving the other behind would result in a non-equivalent global state after the remaining transactions are validated and committed to the servers. Therefore, the third cancellation criterion for `T` requires that all the transactions that have an offsetting relation with `T` can be also cancelled together with `T`.

Note that it is quite possible that some of the cancelled transactions may have been invalidated if they had remained in the transaction history. For example, consider the following two disconnected, offsetting transactions $\mathtt{T}_1 = \{\mathtt{mkdir\ home/foo}\}$ and $\mathtt{T}_2 = \{\mathtt{rmdir\ home/foo}\}$. Suppose that a new object `home/foo` has been created on the servers during the disconnection. If $\mathtt{T}_1$ and $\mathtt{T}_2$ are not cancelled, they will be invalidated because of the update/update conflict on `home/foo`. Thus, transaction cancellation increases the chances for the complete history of disconnected transactions to pass validation and commit their results to the servers. Formally, if the local transaction history for a disconnected operation session is H_d and the set of cancelled transactions is $\sum$, then it is possible that $(H_d - \sum)$ can be validated while H_d can not.

Checking the Cancellation Criteria The automatic checking of the first cancellation criterion can be implemented using Coda's original mechanism for cancelling inferred transactions. For each non-transactional, disconnected mutation operation, Venus iterates through the entire CML of the corresponding volume in reverse chronological order to search for and remove any log record that is overwritten or offset by the new mutation. In the case of offsetting, the new mutation operation itself will also be removed from the CML [26]. Similarly, for every disconnected mutation operation op_T performed by a transaction `T`, Venus searches the corresponding CML and marks all the log records that are either overwritten or offset by op_T, including op_T itself. If all the records in `TML(T)` are marked, it means that the result of `T` is no longer visible on the client local state and Venus then marks `T` as obsolete.

Validating the second cancellation criterion for a pending transaction `T` can be performed by checking all the live transactions that read from `T`. We use `read-from(T)` to denote the set {`T'` | The execution of `T'` reads an object whose value is written by `T`.}. `T` can be marked as a covered transaction if every transaction `T'`∈`read-from(T)` satisfies the condition that `R(T')` is a super-set of `R(T)`. Because the readset of a transaction is always a super-set of its writeset, this is sufficient to guarantee that the removal of `T` from the local transaction history will not affect the GC validation outcome of any other pending transactions.

The third criterion is more difficult to validate because it requires a group of transactions to satisfy a certain relationship. A key observation is that the offsetting relation induces an equivalence relation among all the disconnected transactions. Pending transactions belonging to the same partition of the equivalence relation must be cancelled together when all of them are marked as obsolete and covered. Thus, this problem can be solved by representing the offsetting relation with a simple graph and identifying fully connected components. Although detecting cancellable isolation-only transactions is a more complicated process than that of inferred transactions, the benefits of transaction cancellation far outweighs the detection cost.

Intra-Transaction Optimization Typical transactions such as `make` usually perform a lengthy computation accessing a large number of objects. Such long transactions often create temporary objects and later remove them. The same cancellation mechanism used for inferred transactions by Venus is applied within the scope of a single transaction to remove the unnecessary records from the TML.

4.3 Merging Local State with Global State

This section discusses how to realize the central part of the IOT consistency model, ensuring global transaction isolation during transaction propagation. Our discussion concentrates on the actions performed by the transaction system when a disconnected client is able to re-establish communication with relevant servers. We first outline a general framework of synchronizing the local client state with the global server state, establishing a broader perspective for the discussion of the transaction propagation process. We then describe how the results of disconnected transactions are incrementally propagated to the servers, i.e., validated and committed or resolved one at a time. Finally, details about transaction validation and transaction commitment are presented. Due to its complexity, the discussion of transaction resolution is deferred to the next two chapters.

4.3.1 Synchronizing Local and Global States

During disconnected operation, both the local state of a disconnected client and the global state of the servers evolve along their own courses, departing from the shared initial state at disconnection time. Upon re-connection, the disconnected computation results on the client need to be validated and propagated to the corresponding servers. In addition, cached objects need to reflect new server updates. Such synchronization between the local and the global states demands a change in the basic Venus operation states.

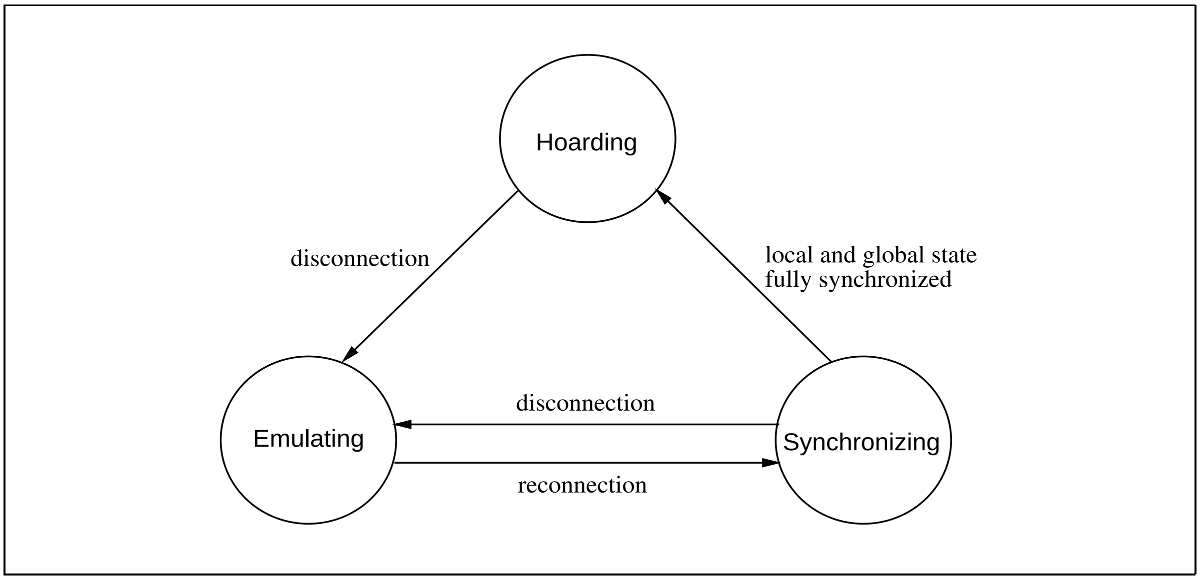


Figure 4.4: The IOT-Venus States and Their Transitions

In the original Venus, propagating disconnected mutations is carried out during the reintegrating state as illustrated in Figure 3.1. It is a transient state because reintegration can be performed in a short period of time and Venus immediately transits to the hoarding state even if the reintegration fails [26]. In the IOT-Venus (the Venus with the IOT extension), the process of propagating disconnected computation results can last arbitrarily long. Transactions may be invalidated and conflict resolutions can take an unlimited amount of time because the transaction system has no control over how long manual repair will last. In short, there can often be a sustained period during which the local state of a physically connected client remains asynchronous(i.e., unsynchronized) with the global state.

As a result, Venus may not be able to transit from the emulating state into the hoarding state by passing through a short transitional state. As shown in Figure 4.4, the original reintegrating state is replaced by a *synchronizing* state. A client is in the synchronizing state if it possesses physical connections to the relevant servers but contains transactions that are still in the pending state. In this state, new computations can be performed but will be checked to see if they depend on any currently pending transactions. If so, they will become new pending transactions and wait to be propagated to the servers. Otherwise, the new results will be committed to the servers immediately. The fundamental difference between the original Venus's reintegrating state and the new synchronizing state needed for supporting IOT is that the former is a short-lived transient state while the latter can be sustained for a long period of time.

4.3.2 From Servers to Client: Cache Validation

Cache validation is the process by which a re-connected client synchronizes itself with server state. During disconnected operation, Venus marks the *cache coherence status(CCS)* of all cached objects to be suspect. Upon re-connection, the cache validation mechanism will compare the local and global versions of every cached object with suspect CCS. If the two versions are identical, the object's CCS will be reset as valid again. Otherwise, it is marked as invalid so that a subsequent access to the object will cause Venus to re-fetch the new global version from the corresponding server. The actual cache validation process is carried out as a side effect of demand fetching or via a periodic daemon, whichever comes first. Logically, cached objects can be regarded as immediately re-synchronized with the global state upon re-connection, except for those accessed by transactions that are yet to be committed or resolved. Note that the CCS of objects accessed by live transactions is temporarily marked as valid to prevent their server copy from being fetched until the relevant transactions are committed or resolved. Note that the cache validation process described in this paragraph is already supported in the existing Coda Venus. No changes in cache validation are needed for supporting IOT.

4.3.3 From Client to Servers: An Incremental Propagation Approach

The other direction of client/server state synchronization is merging disconnected computation results with the servers. We decided to employ an incremental approach where disconnected transactions are validated, committed or resolved one at a time. Note that the incremental transaction propagation mechanisms to be discussed shortly do not exist in the original Coda system. They are specifically introduced in Coda for the purpose of supporting IOT.

Rationale The key reason behind the incremental propagation approach is the overriding concern of practical usability. For many Unix users and application programmers, the concept of a transaction service providing serializability-based isolation guarantees is new. Requiring them to deal with the potential conflicts by either programming resolvers or manually repairing invalidated transactions is even more demanding. Reducing conceptual complexity, minimizing the scope of conflicts, and cutting down the exposure of transaction operation details is of paramount importance. The incremental approach emphasizes simplicity by localizing the scope of potential inconsistency and exposing conflicts to the resolver one invalidated transaction at a time. This allows the resolver to concentrate on what the transaction has done locally and what has been changed on the global state during disconnection, without worrying about possible interference from other transactions. Although propagating disconnected transactions in groups could lead to reduced performance cost, on balance the incremental strategy better serves our key design objectives.

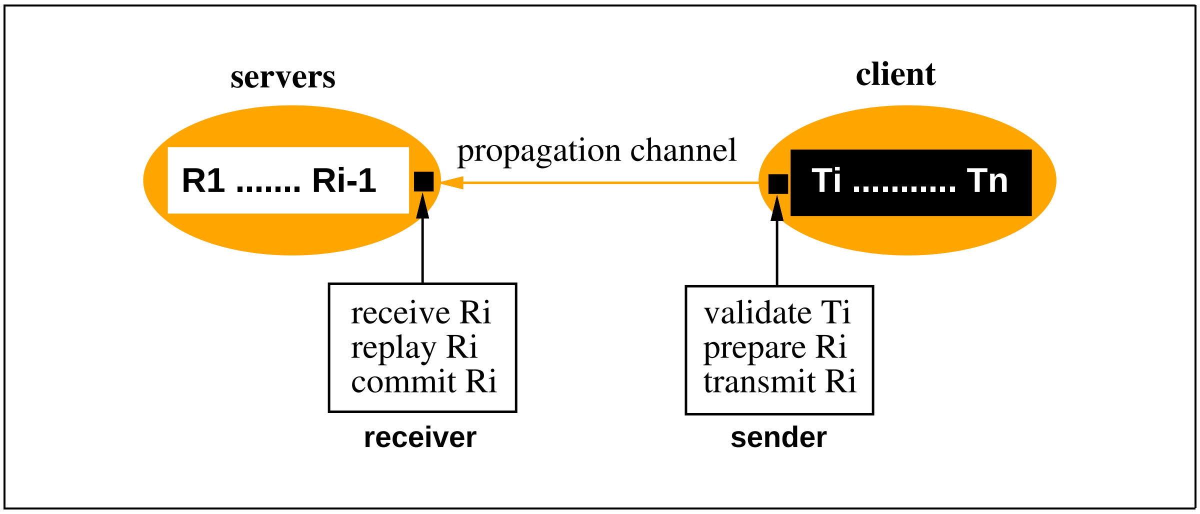


Figure 4.5: An Incremental Transaction Propagation Framework

Intuitive Description A better way to describe the incremental propagation process is to imagine that there is a logical channel through which local computation results are transmitted from the client to the servers, as show in Figure 4.5. There is a *sender* at the entrance of the channel to ensure that only consistent results are allowed to go through the channel. At the server end, a *receiver* fetches the propagated results from the channel and atomically installs them. Suppose that the local transaction history contains n transactions: $\mathtt{T}_1$, $\mathtt{T}_2$, ..., $\mathtt{T}_n$. The sender repeatedly grabs the transaction at the head of the list, say $\mathtt{T}_i$, validates its consistency and prepares $\mathtt{R}_i$, the representation of the final outcome of $\mathtt{T}_i$ that is actually transmitted through the channel. If the validation succeeds, $\mathtt{R}_i$ is just the mutation log of $\mathtt{T}_i$. Otherwise, the sender will force a resolution for $\mathtt{T}_i$ and $\mathtt{R}_i$ is the sequence of logged mutations representing the resolution outcome. When the receiver obtains $\mathtt{R}_i$, it replays the mutation operations in the log on the corresponding servers. After all the replay results are atomically committed, it notifies the sender to propagate the next transaction T_{i+1}.

Propagation Algorithm Venus dedicates a special thread called the *propagator* to carry out the mission of incremental transaction propagation. The propagator is awakened every time the client regains a lost connection to a server. The pseudo code in Figure 4.6 describes the algorithm of incrementally validating, committing or resolving disconnected transactions.

Two steps in the above algorithm need further explanation. First, the step that invokes the automatic resolver is not limited to application-specific resolvers. It includes automatic transaction re-execution and automatic abort as special cases when the resolver is the transaction itself or a `no-op` respectively. Second, the successful resolution of `T[i]` often requires adjusting the state of those transactions that read from it. For example, suppose that transaction `T[j]` (`j>i`) reads the value that `T[i]` wrote on an object `obj`, and the resolution of `T[i]` produces a new server replica of `obj` that is different from what it wrote during the original execution. Clearly transaction `T[j]` needs to be invalidated because it has accessed an object value that no longer exists in the file system. Specifically, the propagator thread needs to iterate through each object `obj` that `T[i]` updated and for which `T[i]`'s resolution created a new server replica. Any pending transaction that read the value `T[i]` wrote on `obj` is immediately invalidated.

Transaction Group One important issue that is not included in the algorithm is that the incremental approach cannot be strictly followed when some of the transactions are not equipped with their complete set of shadow cache files due to client disk space shortage. Suppose that a pending transaction `T` updated a file object `obj` during its execution. However, `obj` is overwritten by another transaction `T′` and `shdset(T)` does not include a shadow cache file for `obj`. We call `T` an *incomplete* transaction and `T′` the *overwriter* of `T`. Under such circumstances, propagating `T` alone is semantically incorrect because `obj` will reflect the result of `T′` instead of `T`.

Our solution to this problem is to group an incomplete transaction `T` with its overwriter(s) into a *result propagation unit* denoted `RPU(T)`. Because the overwriting relation among disconnected transactions is transitive, a transaction `T′` is a member of `RPU(T)` if it satisfies one of the following conditions:

1. `T′` is `T` itself.
2. `T′` overwrote `T` on a file object `obj` and `shdset(T)` does not include a shadow cache file for `obj`.
3. There exits another transaction `T"`$\in$`RPU(T)` such that `T′` overwrote `T"` on a file object `obj′` and `shdset(T")` does not include a shadow cache file for `obj′`.

```
Start:
NeedRepeat = false;
perform topological sort on the serialization graph SG;
put all live transactions in sorted order T[1..n];
foreach (i in 1..n) {
    if (T[i] has no predecessor in SG and is fully connected) {
        validate T[i];
        if (validation succeeds) {
            commit T[i];
            if (commitment succeeds) {
                transit T[i] into committed state;
                remove T[i]'s corresponding node from SG;
                NeedRepeat = true;
            } else {
                transit T[i] into resolving state;
            }
        } else {
            transit T[i] into resolving state;
        }
        if (T[i] is in resolving state and
            can be automatically resolved) {
            invoke automatic resolver;
            if (automatic resolution succeeds) {
                transit T[i] into resolved state;
                adjust state for transactions that read from T[i];
                remove T[i]'s corresponding node from SG;
                NeedRepeat = true;
            } else {
                T[i] stays in resolving state;
                request manual repair;
            }
        }
    }
}
if (NeedRepeat) goto Start;
```

Figure 4.6: An Algorithm for Incremental Transaction Propagation

The following minor adjustment needs to be incorporated into the algorithm described in Figure 4.6. When the current transaction `T[i]` is an incomplete transaction, the transaction set `RPU(T[i])` is computed and is treated as one single logical transaction. If all the member transactions of `RPU(T[i])` pass validation, their combined results will be transmitted together to the corresponding servers for commitment. Otherwise, we will conservatively invalidate all of them and request manual repair for them because it is too complicated to automatically resolve a group of transactions. Note that this is likely to be rare in practice because it requires three conditions to be satisfied together: (a) overwriting transactions, (b) transaction invalidation, and (c) client running out of shadow space.

4.3.4 Transaction Validation

The most frequent activity during the incremental propagation process is validating disconnected transactions. Because the G1SR consistency criterion is not supported in the current implementation, we focus on consistency validation using the GC criterion, which will be referred to as *transaction certification* for compatibility with past research literature. Intuitively, the certification of a transaction `T` is to check whether any object accessed by `T` has been updated on the corresponding server since it was first accessed by `T`.

Version Certification There are two basic alternatives for transaction certification. *Value certification* records the data values of objects accessed by a pending transaction and compares them against the corresponding server values. It is impractical for IOT implementation on Coda because recording transaction accessed data values will cost too much client disk space. We adopt the *version certification* strategy by recording the local version identifiers of transaction accessed objects and comparing them against the corresponding server version identifiers.

The process of version certification for a pending transaction `T` consists of the following actions. For each element in `R(T)` and `W(T)`[3], we first use its recorded `fid` to locate the corresponding cached object `obj` and its local version vector[4] denoted `lvv(obj)`. We then send `fid` and `lvv(obj)` to the corresponding server and let it compare `lvv(obj)` against the current global version vector of `obj` denoted `gvv(obj)`. If the two version vectors do not match, transaction `T` fails certification and is invalidated. Otherwise, the version comparison process is repeated until both `R(T)` and `W(T)` are exhausted. To improve performance, the `fid`'s and their corresponding local version vectors are batched and sent to the servers in groups.

[3]The notations `R(T)` and `W(T)` stand for the readset and writeset of transaction `T` respectively.

[4]Coda maintains a version vector for each individual object required by server replication. Since this thesis research does consider server replication, a version vector in this dissertation is treated in the same way as a version stamp.

Value Certification for Attributes The main drawback of version certification is the possibility of false invalidation because version vectors in Coda are maintained at an object level. As discussed in Chapter 3, sub-object level transaction specification is needed to more accurately represent the transactional file access behaviors and avoid unnecessary conflicts. However, maintaining version vectors for each sub-part of an object demands substantially higher server space cost and performance overhead.

Our strategy to reduce the likelihood of false invalidation is to apply value certification on attributes. Each element of the transaction readset and writeset must record not only which attributes of an object are actually accessed by a transaction but also the value of those accessed attributes. Suppose that a pending transaction `T` updated the `mode` bits of an object `obj` during the disconnected operation while `obj`'s data content is updated on the corresponding server. During the certification of `T`, even though the version comparison on `obj` will show the difference between its local and global version vectors, the success of value certification on the `mode` bits of `obj` can help to avoid the unnecessary invalidation of `T`.

Value Certification for Directory Contents Similarly, directory access operations also need value certification to avoid unnecessary conflicts. For every directory access operation performed by a disconnected transaction `T`, `R(T)` and `W(T)` will record the actual names that are accessed by `T`. For example, suppose that `T` performs an operation "`mkdir home/foo`", `W(T)` will contain an element for directory `home` which records that the name "foo" is inserted under `home`. The value certification of `T` on directory `home` succeeds as long as the name "foo" is still unbound under the corresponding server replica of `home`. This kind of directory value certification can correctly avoid the unnecessary conflict on a pair of independent partitioned operations such as "`mkdir home/foo`" and "`mkdir home/bar`".

4.3.5 Transaction Commitment

The second most frequent activity during the incremental propagation process is committing a validated transaction. As in connected transaction execution, we rely on Coda's underlying mutation logging and reintegration mechanisms to commit the local result of a transaction to the servers, using the same transaction mutation log structures and the new transaction reintegration process described in Figure 4.3.

4.3.6 Transaction Resolution

The least frequent but most challenging activity during the incremental propagation process is resolving an invalidated transaction. We devote the next two chapters to describing support for this aspect of our design.

Chapter 5

Detailed Design: Conflict Representation

When a pending transaction is invalidated, the IOT consistency model requires it to be resolved using one of the four resolution options. However, a number of important issues need to be addressed before resolution takes place. How can the local and global state of relevant objects be conveniently accessed by the resolver? Which portions of the local and global state should be visible? How are the users notified that an invalidated transaction needs to be manually resolved if it chose that option? The underlying theme of these questions is how to properly represent the information about invalidated transactions so that they can be effectively resolved by automatic resolvers or human users. Because conflict representation is crucial to the success of conflict resolution, this chapter is devoted to the detailed design issues of representing information about conflicts.

The central focus of this chapter is to present a systematic study on important issues and design trade-offs in providing a concise conflict representation mechanism for the IOT extension to the Coda file system. The first section identifies the fundamental problems and key requirements in conflict representation. The second section mainly describes how to notify users and applications about detected conflicts. The third section discusses how to expose conflict information to the resolvers. In addition, detailed mechanisms of a novel conflict representation scheme are presented.

5.1 Basic Issues of Conflict Representation

5.1.1 Inconsistent Objects

As discussed in Chapter 3, the notion of conflict in this dissertation is associated with invalidated transactions. Intuitively, if a transaction `T` is invalidated, it means that some of the objects it accessed are likely to be in an inconsistent state, i.e., their value does not satisfy the necessary consistency requirements. Because transaction `T` is a past thread of computation, it is those objects that embody the conflicts caused by `T`. We define an object `obj` to be *inconsistent* if it satisfies both of the following two conditions:

1. `obj`$\in$(`R(T)`$\cup$`W(T)`) where `T` is an invalidated transaction. In other words, an inconsistent object must have been accessed by an invalidated transaction.

2. `lv(obj)`$\neq$`gv(obj)` where `lv(obj)` and `gv(obj)` stand for the local and global version of `obj` respectively. This means that `obj` is either mutated by transaction `T` or updated on the corresponding server during the disconnection, or both.

We use `inc(T)` to denote the set of inconsistent objects caused by transaction `T`. Any object in `inc(T)` is either accessed by `T` and updated on the corresponding server or mutated by `T`. It is not difficult to see that further accesses to objects in `inc(T)` could lead to cascading conflicts. Suppose that an object `obj`$\in$`inc(T)` is accessed by an ongoing transaction `T'`. `T'` will certainly be invalidated if `obj` has already been updated on the corresponding server. Even if `obj` is only mutated by `T`, `T'` is still likely to be invalidated because the resolution of `T` may cause the server replica of `obj` to be updated.

Objects in `inc(T)` capture the key information about the conflicts caused by `T` for two reasons. First, any object that is known to have a different current server state than `T` originally accessed is a member of `inc(T)`. Second, `inc(T)` contains the complete effect created by `T` because it is a superset of `W(T)` by definition. Thus, `inc(T)` embodies what `T` would have seen and produced differently had it not been disconnected. In essence, conflict representation for transaction `T` is exposing information about objects in `inc(T)` under different circumstances.

5.1.2 Two Venus Operation Modes

A transaction `T` may not be immediately resolved after its invalidation for two reasons. First, `T` may have chosen the manual resolution option, and the user may defer repairing it. Second, even though `T` has chosen one of the automatic resolution options, the transaction system may not be able to resolve it right away because new failures could cause the client to lose some of the necessary server connections. Hence, there could be an arbitrarily long period between `T`'s invalidation and resolution. During this period, objects in `inc(T)` must be marked as inconsistent and prohibited from any new accesses to prevent cascading conflicts. However, as soon as the resolution starts, both the local and global state of objects in `inc(T)` must be made visible to `T`'s resolver so that it can analyze and resolve the conflicts. Therefore, the same inconsistent objects must possess different visibilities depending on whether `T` is being resolved or not. The IOT-Venus adopts two basic operation modes to serve these two distinct needs.

The IOT-Venus operates in the *service mode* when no transactions are being resolved. In this mode, accesses to inconsistent objects are denied while other non-inconsistent objects behave normally. When an invalidated transaction `T` starts resolution, the IOT-Venus switches to the *resolution mode* and dynamically adjusts the internal representation of relevant inconsistent objects so that their local and global state can be accessed by the resolver. To avoid interference, transactions are resolved one at a time and regular transaction executions are blocked during the resolution mode. The IOT-Venus returns to service mode as soon as `T`'s resolution is completed. At any given moment, it operates in either of the two modes and there is no overlap.

5.1.3 Conflict Representation Requirements

The central mission of conflict representation is to provide necessary information about the conflicts caused by invalidated transactions to the resolvers. When operating in the service mode, the IOT-Venus has two responsibilities other than the regular cache management duties such as servicing cache misses. First, it must notify the users about the existence of conflicts. Second, it must prevent inconsistent objects from being accessed. During the resolution mode, the IOT-Venus must provide the resolver with convenient access to both the local and global state of all the inconsistent objects accessed by the transaction being resolved. In addition, only the global state of all other objects should be made visible to the resolver.

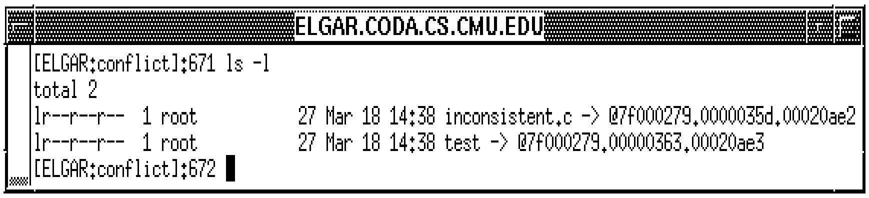
5.2 Conflict Representation in Service Mode

5.2.1 Conflict Notification

Design Alternatives There are two basic approaches to notifying the users about detected conflicts. The first approach utilizes outside communication facilities such as sending email [56], displaying messages on terminals, sending on-line *zephyr* [10] messages, etc. This *external* approach has the advantages of being informative and simple to implement. However, there are many disadvantages. First, notification messages are often transient and non-repeatable. If the users do not pay attention or the messages get lost, the users will remain ignorant of the detected conflicts. Second, this approach is not suitable for informing ongoing applications trying to access inconsistent objects. Third, it renders the proper functioning of the transaction system dependent upon the availability of specific message communication facilities.

To overcome these shortcomings, our design adopts an *internal* conflict notification approach by changing the visibility of inconsistent objects inside the Coda namespace. The key idea is to make those objects look different from they otherwise would under normal circumstances so that both the users and the applications can be aware of the fact that they are inconsistent. This approach is *persistent* in the sense that the notification message is attached to the objects themselves and is repeated every time an inconsistent object is accessed. It also makes the transaction system more self-contained.

Dangling Symbolic Link The mechanism for internal conflict notification is converting an inconsistent object into a dangling symbolic link, i.e., a symbolic link that does not point to any real object. The link content uses a special format of `@x.y.z` where `x`, `y` and `z` are a hexadecimal number encoding the three components, `volume-id`, `vnode-id` and `uniquifier` of the `fid` of the inconsistent object. Note that such dynamic conversion is performed only on the client where inconsistent objects are detected and they appear as normal objects on other clients. Figure 5.1 shows an actual example of dangling symbolic links. To ensure that the symbolic links are always dangling and never point to any real objects, the Coda file system forbids any object to have a name of the above format. In essence, we give up a tiny portion of legal names so that accesses to inconsistent objects are guaranteed to yield an unexpected result.



This `xterm` image displays the dangling symbolic links for a file `inconsistent.c` and a directory `test` that are detected to be inconsistent. The three hexadecimal components of each symbolic link correspond to the three components of the actual `fid`.

Figure 5.1: An Example of Dangling Symbolic Links

5.2.2 Access Prevention

The dangling symbolic link representation can not only visually notify the users about conflicts but also serve the purpose of preventing new processes from accessing inconsistent objects.

Performing common file access operations such as `open`, `read` and `write`, etc. on inconsistent objects will all fail because the corresponding symbolic links point to nowhere. The only exception is when the inconsistent object is a symbolic link itself because a `readlink` operation on it can still succeed. But the `readlink` operation will return the specially formated string instead of the real link content.

However, it is possible for an object `obj` to be detected as inconsistent while an ongoing process `P` has already obtained a reference to the `vnode` of `obj`, allowing `P` to continue accessing the current server replica of `obj`. This causes a sudden content switch because `P` was accessing the local version of `obj` and changes to the global version without knowing it. Obviously, we must prevent this from happening because it can result in serious consequences.

Our solution is to create a table containing the `fid` of all the inconsistent objects. For every file access operation, the IOT-Venus will search the table to see if any operand has a `fid` already marked as inconsistent. If so, the error code `EACCES` will be returned to deny the operation. An obvious drawback of this method is the performance overhead of searching the table for every file access operation. To alleviate this problem, we attach a flag to each cached volume to indicate whether it contains any inconsistent objects or not. Thus, we can check the volume tag to avoid unnecessary table searching when the corresponding volume does not have any inconsistent objects. Because conflicts are rare and tend to localize in a few volumes, this technique allows us to avoid paying a significant performance price under normal circumstances.

5.2.3 Visibility Maintenance

Visually notifying the users about inconsistent objects and preventing further access to them are only part of the general task of maintaining tentative computation results for live transactions.

5.2.3.1 Maintaining Live Transaction Results

The IOT-Venus has two main responsibilities in service mode. First, it must perform the regular cache management duties such as servicing cache misses and maintaining cache coherence. Second, it needs to maintain the tentative computation results produced by the live transactions so that they are visible in the client local state.

Object Asynchrony The essence of maintaining tentative computation results is to keep relevant cached objects asynchronous from their server state. Here, we use the terms asynchrony or asynchronous to refer to the situation where the client local state remains different from the server global state. In order to keep the result of a running or pending transaction visible in the client local state, the IOT-Venus must hold onto the cache copies of objects that are mutated by the transaction without fetching their server replicas even if they are updated. Furthermore, cache copies of objects only read by those transactions need to remain in the client cache even though their server counterparts might have been changed. Thus, inconsistent objects are only a portion of the asynchronous objects that are made inaccessible to avoid spreading inconsistencies.

Visibility Control The key to managing asynchronous objects is to control their visibility, i.e., the way they are visible to the users and applications. Object visibility has many aspects such as object form (regular file, directory and symbolic link) and object accessibility. The IOT-Venus dynamically adjusts different aspects of object visibility of asynchronous objects to serve different needs under different circumstances. The dangling symbolic link representation for inconsistent objects is just one form of visibility control where the accessibility of an inconsistent object is revoked and the form of the object is dynamically transformed to make a visual difference.

5.2.3.2 State-Based Visibility Maintenance

The difficulty of maintaining asynchronous cached objects is that their visibility needs to be dynamically adjusted in response to different kinds of system activities. Events such as a new transaction starting execution, a pending transaction being committed, an invalidated transaction being successfully resolved, and the server replica of an asynchronous object being updated, require the visibility of the relevant objects to be dynamically changed. Because live transactions may create complicated inter-dependencies among themselves, questions such as when an inconsistent object can become a normal object again become very difficult to answer.

Our strategy is to use a state-based approach to maintain visibility for cached objects. The key idea is to classify all the cached objects into a small number of states such that any system activity only causes the relevant objects to transit from one state into another. By assigning a specific visibility control policy to each state, visibility maintenance becomes the disciplined action of following state transitions and adjusting visibility accordingly.

Basic States of Cached Objects Any asynchronous cached object `obj` must have been accessed by at least one live transaction. Each such transaction is called a *guardian* of `obj`. We classify all cached objects into the following four basic states.

- **Consistent State**

  An object `obj` is in the *consistent* state if it does not have any guardian, i.e., it has not been accessed by any live transaction. The visibility control policy for consistent objects uses the regular callback cache coherence protocol to keep their cache copies fully synchronized with the corresponding server replicas. In other words, consistent objects are *synchronous* objects.

- **Clean-Local State**

  An object `obj` is in the *clean-local* state if it has at least one guardian and its local version is identical to its current server version, i.e., `lv(obj) = gv(obj)`. This means that `obj` is accessed by some live transactions but has not been mutated either locally on the client or globally on the server. It is *clean* because it has not been updated and it is *local* because it must remain in the client cache until all its guardians are committed or resolved. The visibility control policy for clean-local objects exposes their local state in their original form but pins them inside the client cache so that their server state will not be visible.

- **Dirty-Local State**

  An object `obj` is in the *dirty-local* state if it has been updated by at least one of its guardians and none of the guardians that mutated `obj` are invalidated. `obj` is local because it needs to stay in the client cache until all its guardians are committed or resolved. It is *dirty* because it has been updated on the client (and by definition not updated on the server). The visibility control policy for dirty-local objects is the same as clean-local objects. Their local state is accessible in their original form but their global state remains invisible.

- **Inconsistent Object**

  As previously defined, an object `obj` is in the *inconsistent* state if it has been accessed by at least one invalidated guardian `T` such that it has either been mutated by `T` or it has been updated on the server. Because accessing inconsistent objects will lead to cascading inconsistencies, the visibility control policy for inconsistent objects prohibits their local or global state from being visible and their form is changed into a dangling symbolic link to visually notify the users about the conflicts.

Under normal circumstances, most of the cached objects are in the consistent state and fully synchronized with their server replicas. A small portion of them remain asynchronous because they are accessed by live transactions. Some of them display their local state in order to represent the tentative computation results of live transactions. Some of them are temporarily inaccessible due to invalidated transactions. Figure 5.2 depicts a visibility distribution among all the cached objects in a client cache.

Object State Transitions The four basic states characterize all the possible situations of any cached object. All the legal transitions among them are shown in Figure 5.3 and explained as follows.

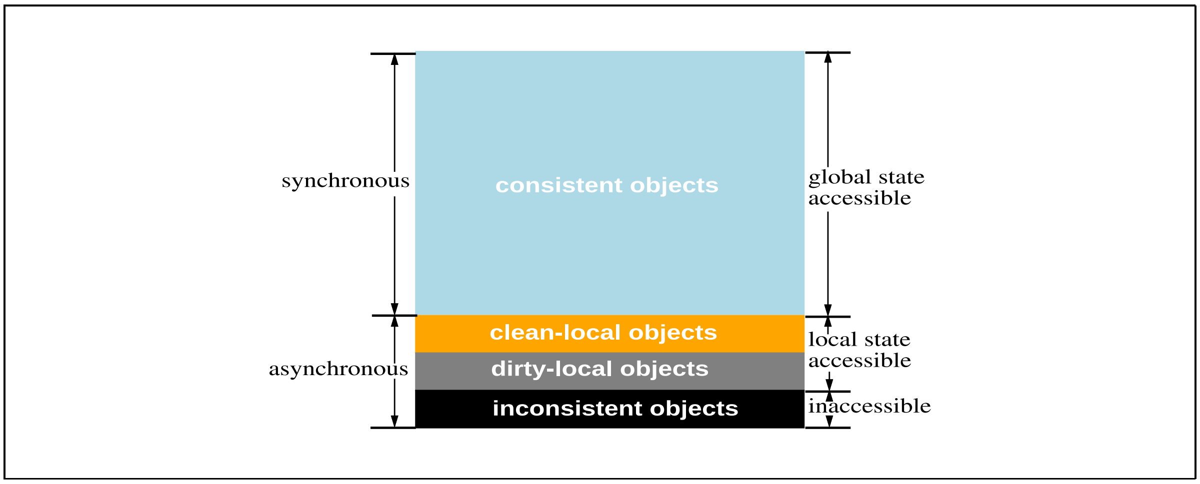


Figure 5.2: Visibility of Cached Objects

- **Consistent Object**

  A consistent object can become a clean-local or a dirty-local object when it is first read or written by an ongoing transaction respectively. It *cannot* directly become an inconsistent object because it has to be accessed by a running transaction first in order to be inconsistent.

- **Clean-Local Object**

  A clean-local object can go back to be a consistent object when its last guardian is committed or resolved. It can become a dirty-local object when it is updated by an ongoing transaction. In the worst case, it will degenerate into an inconsistent object when its server replica is updated.

- **Dirty-Local Object**

  A dirty-local object can similarly go back to be a consistent object when its last guardian is committed. It can become a clean-local object when its last update-guardian is committed while still having other read-guardians. It will fall into the inconsistent state when one of the following two things happens: its server replica is updated or one of its update-guardians is invalidated.

- **Inconsistent Object**

  When the last guardian of an inconsistent object `obj` (it has to be an invalidated transaction by definition) is resolved, the object is reborn as a fully synchronized consistent object. The resolution or commitment of `obj`'s read-guardians does not change its state. However, if `obj` still has other guardians when an update-guardian `T` is resolved, it could remain inconsistent if it has other invalidated guardian(s) that have accessed a local version of `obj` different from the current global version. Otherwise, `obj` will become either a clean-local or a dirty-local object depending on whether it has been mutated by the remaining guardians or not.

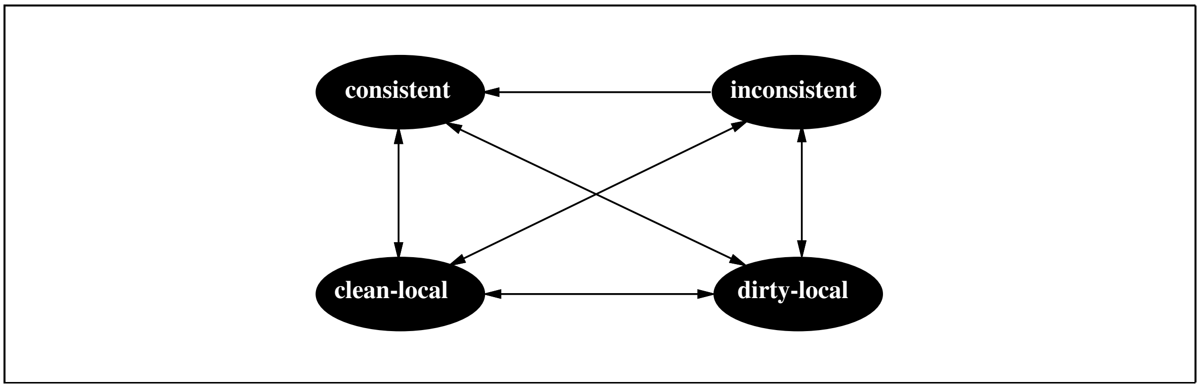


Figure 5.3: Cached Object States and Their Transitions

5.3 Conflict Representation in Resolution Mode

The resolution mode is tied to the resolution of one and only one invalidated transaction. The main responsibility of the IOT conflict representation mechanism is to provide the resolver with convenient access to the local and global states of all inconsistent objects accessed by the transaction. For all other objects, only their global state should be visible to the resolver to guarantee that the resolution outcome does not depend on the result of any other live transactions.

Despite its importance in conflict resolution, the issue of how to best represent conflict information to the resolvers has not been adequately addressed in previous research. Our design principle here is to expose conflict information to the resolvers in as simple and concise a form as possible to reduce the burden of programming application-specific resolvers and make the process of manual resolution easier. This section not only presents our novel conflict representation scheme but also studies key alternatives to justify our design decisions.

5.3.1 Exposing Local and Global State of an Inconsistent Object

5.3.1.1 Naming Replicas

The Need for Naming Replicas Conceptually, the local and global replicas of an inconsistent object `obj`, denoted `lr(obj)` and `gr(obj)`, are two physical copies recording the client and server state of `obj`. Under normal circumstances, replicas are not *first class citizens* because they cannot be directly named and accessed though the standard UFS API. However, during the resolution of an invalidated transaction `T`, the local and global replicas of objects in `inc(T)` need to be accessed by `T`'s resolver in the same way as normal objects. This is vital to effective conflict resolution. Otherwise the resolver will require a separate interface for accessing replicas. It means that all the existing system facilities, applications and tools cannot be directly used for resolving conflicts. This will severely limit the capability and effectiveness of resolvers and also complicate their programming.

There are two important usability issues for the IOT conflict resolution mechanism. For manual conflict resolution, we should allow the users to take advantage of whatever facilities exist at their disposal (such as `emacs`, `latex`, `make` and `gcc`) to repair an invalidated transaction. For automatic conflict resolution, we should facilitate the programming of resolvers by allowing replicas to be accessed via the standard UFS API. This allows resolvers to use existing applications and tools. It is clear that the practical usability of IOT conflict resolution demands that replicas be treated as first class citizens during conflict resolution.

Design Alternatives In Unix file systems, objects are named with their unique locations in the hierarchical namespace. Any object `obj` can be named by its unique pathname denoted `pn(obj)` starting at the top of the namespace. Allowing a replica to be named in the same way as a normal object requires it to be stored at a specific location of the file system. In addition, there needs to be a convention that maps the pathname of an inconsistent object into the pathnames of its local and global replicas. In other words, for any inconsistent object `obj`, there needs to be a deterministic way of computing `pn(lr(obj))` and `pn(gr(obj))` based on `pn(obj)`.

The specific replica naming issues for an inconsistent object `obj` include the following: What are the pathnames of `lr(obj)` and `gr(obj)` and where are they stored? What is the visible content at the original location of `pn(obj)`? What is the convention to translate `pn(obj)` into `pn(lr(obj))` and `pn(gr(obj))`? There are different alternatives that can address each of the questions. For example, an earlier Coda implementation stored the local replica of an inconsistent object in a *closure file* using the `tar` format on a local disk file system [26]. The local replica is not directly accessible until extracted from the closure file. The corresponding global replica is accessible at the object's original location.

The In-Place Naming Strategy Our design uses an *in-place* replica naming strategy that has been used in Coda for representing conflicts among server replicas [29]. It has two key characteristics that are summed up by the phrase "in-place". First, all replicas are represented within the Coda namespace. We do not rely on any other file system to maintain replicas of inconsistent objects. Second, the local and global replicas of an inconsistent object `obj` are stored in locations near the original location `pn(obj)`. The closeness in namespace makes it easier for the resolver to locate and inspect needed replicas. It is also easier for users doing manual resolution. When directly accessed via the UFS API, replicas are read-only and their main purpose is to provide information about the local and global state of relevant inconsistent objects.

The in-place naming convention of an inconsistent object `obj` uses `pn(obj)/local` and `pn(obj)/global` as the pathnames of its local and global replicas. The visible content at location `pn(obj)` becomes a directory containing only two children named "local" and "global" representing the local and global replicas of `obj` respectively. For example, if a file `/coda/misc/test` is inconsistent, its local and global replicas will be located at `/coda/misc/test/local` and `/coda/misc/test/global` respectively.

5.3.1.2 Dealing with Directory Replicas

For a simple object such as a file or a symbolic link, its local or global replica is just another file or symbolic link storing its local and global state. However, the representation of a directory replica needs to be different because the data content of a directory contains references to other objects.

Subtree Representation For compatibility reasons, we inherit the *subtree representation form* for directory replica that is also employed by the Coda file system to represent conflicts among server replicas [29]. Note that Coda employs a different set of mechanisms for supporting optimistic server replication and disconnected operation. This subtree form of replica representation was employed by the original Coda system to only represent conflicts among server replicas, not in disconnected operation which is the focus of this dissertation. The conflict representation mechanisms discussed in this document are all newly added to the existing Coda system for supporting IOT.

The local replica of an inconsistent directory `dir` is represented by the *local subtree* consisting of cache copies of objects that are descendents of `dir`. Similarly, the global replica of `dir` is represented by the *global subtree* consisting of server replicas of objects that are descendants of `dir`. It is only during conflict resolution that they are represented by the corresponding local subtree and global subtree respectively. We regard subtrees as the general form of replica representation. Note that the local and global subtrees of an inconsistent file or symbolic link contain only one node.

The advantage of subtree representation is that it retains the original hierarchical structure among replicas within the local and global subtrees of an inconsistent directory, making it easier for the resolver to probe around inspecting and comparing relevant replicas. The drawback is that it may cause significant client space overhead because two subtrees are needed to represent an inconsistent directory. When a conflict occurs in a directory at a high level in the file system hierarchy, hundreds of nodes could be involved.

Space Cost Analysis There are two kinds of client space costs associated with the subtree representation for directory replicas. Because the entire local subtree of any inconsistent directory needs to be kept in the client cache, the internal representation of cached objects within the subtree could consume a fair amount of persistent storage space. In addition, the cache files of those objects could occupy a significant amount of disk space. When the server replicas of the corresponding global subtree are being accessed, more persistent storage and disk space are needed.

We conducted a feasibility study by analyzing the persistent storage and disk space needed for storing a typical subtree. The key observation is that directory update activities tend to occur at the bottom levels of the namespace hierarchy. Since any inconsistent directory must be caused by a directory update operation, the typical subtree corresponding to an inconsistent directory is small. The analysis was performed based on previously collected file reference traces [46] and statistics about file system object distributions [12]. It shows that the space cost for storing a typical subtree is modest and acceptable in normal circumstances. Details are presented in Chapter 9.

The Caching Factor Another important factor that enhances the feasibility of subtree representation is that only the local subtree of an inconsistent directory needs to be resident in the client cache. Because the server replicas within a global subtree can be accessed as normal objects, they are demand fetched to the client cache and swapped out when the client runs out of cache space. Furthermore, local subtrees of inconsistent directories already reside in the client cache. Thus, both the local and global subtrees of any inconsistent directory can be accessed through demand fetching and cache replacement activities.

Handling Uncached Objects The local subtree of an inconsistent directory sometimes is incomplete because some of the objects in the subtree may not have been cached when the client was disconnected. If we do not include those un-cached objects in the local subtree, there will be an ambiguity problem because the resolver cannot distinguish whether those objects were initially un-cached or locally removed. To avoid this problem, we bind the names of un-cached objects to specially-generated `fids` so that their names are visible in the local subtree but any attempt to access them will result in an error code of `ETIMEDOUT`. Although this approach may seem confusing to the users doing manual repair, it achieves the important goal of presenting the local subtree in exactly the same way when the client was disconnected.

5.3.1.3 Providing Resolution Workspace

The Need For Workspace Resolving conflicts needs a workspace to build up the resolution result. For example, suppose that an invalidated `make` transaction $\mathtt{T}_{make}$ compiled an object file `work.o` and its resolution needs to compile a new version for `work.o`. The IOT conflict representation mechanism needs to provide a workspace for the inconsistent object `work.o` so that it can be used as the place-holder for storing the new compilation result.

Design Alternatives Several different strategies can be used to provide resolution workspace. One method used by the Coda file system for repairing conflicts among server replicas of a file object allows the repairer to use any file from any file system as the workspace for holding repair results. The repairer must provide information about the location of the workspace file to Coda so that its data content can be fetched and installed as the repair outcome for the corresponding inconsistent file.

The second alternative is to extend the current form of conflict representation so that an inconsistent object `obj` will be dynamically converted into a directory containing three children. In addition to the original local and global children, a third child named "workspace" is added and initialized with an identical copy of the corresponding global subtree. The workspace subtree is directly mutable via the UFS API so that the resolver can use it to store the resolution result for `obj`. When resolution is over, the transaction system will automatically install the current state of the workspace subtree as the final resolution outcome for `obj`.

The Dual Replica Representation The second alternative is appealing because it fits well with the overall design philosophy of localizing conflict representation for access convenience. However, its implementation will be very complicated because Venus needs to maintain the internal representation of three different replicas of the same object, and allow two of them to be mutable. Thus, we decided to settle for a variation. The key idea is to overload the global and workspace subtrees of an inconsistent object `obj`. The global subtree of `obj` serves both purposes of providing access to the current global state of `obj` and the place-holder for storing the resolution result of `obj`. We call this approach the *dual replica representation (DRR)* and Figure 5.4 shows its basic structure. A slight disadvantage of this approach is that the resolver needs to access all the needed global replicas before updating them to store the resolution result. Figure 5.5 presents an actual example of dual replica conflict representation.

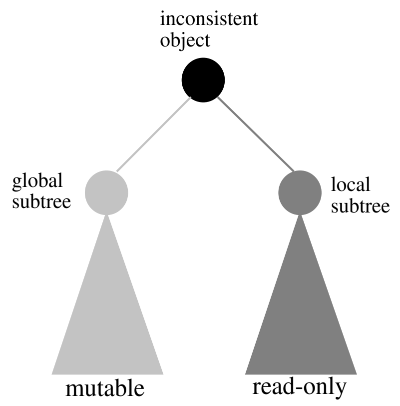
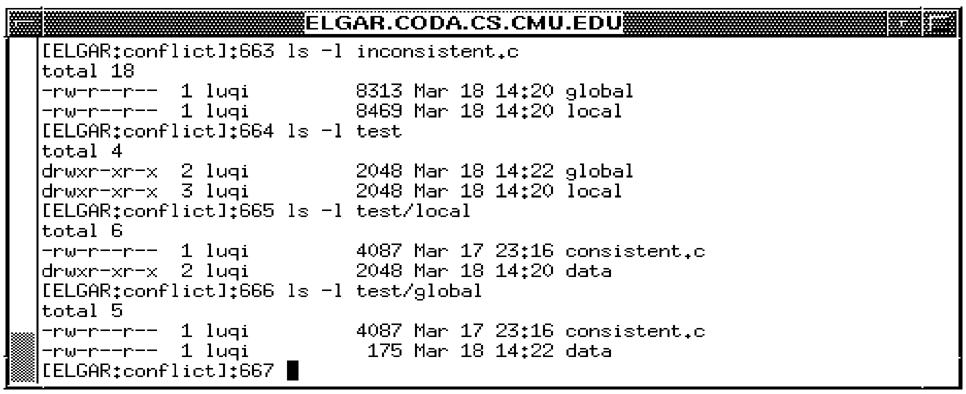


Figure 5.4: The Basic Structure of Dual Replica Representation



This `xterm` image shows the dual replica conflict representation of the two inconsistent objects shown in Figure 5.1. As can be seen, file `inconsistent.c` has two replicas with different sizes. The local subtree of directory `test` contains a directory named `data` while the global replica contains a file named `data`.

Figure 5.5: An Example of Dual Replica Conflict Representation

5.3.2 The Realization of Dual Replica Representation

The main difficulty of implementing DRR comes from the fact that the IOT-Venus must maintain both the local replica and the cache copy of the global replica of the same inconsistent object in the cache. The main data structures and methods used for DRR implementation are discussed below.

5.3.2.1 Key Data Structures

A Special Local Volume As both a system administration unit and a key design component, the concept of volume is deeply rooted in every aspect of the design and implementation of Coda. Because the Coda implementation requires that every object must belong to a volume, we use a special *local volume* to serve as the home of all local replicas. Unlike a normal volume, the local volume does not correspond to any actual server and remains resident at every client. It is a read-only volume and stays in the connected state, making local replicas always accessible.

Logically, each client is the server of its own local volume because it provides the home for storing the cache copies of all local replicas. In the actual implementation, the IOT-Venus treats the local volume in the same way as a normal cached volume except that it must suppress any attempt at server communication on behalf of the local volume. From the viewpoint of a resolver, local replicas appear the same as normal objects except that they are read-only.

Internal Structure of Dual Replica Representation For an inconsistent object `obj`, its dual replica representation uses an internal structure shown in Figure 5.6 and it will be referred to as the DRR subtree of `obj`. The IOT conflict representation mechanism dynamically converts `obj` from a dangling symbolic link into a directory containing the local subtree and global subtree of `obj`. There are six key nodes in the DRR subtree and their roles are as follows:

- The topmost node is the parent of `obj` and is called the *DRR-base* because it provides the basis for planting the DRR subtree of `obj` in the client local state. It plays the role of a connector that links the regular Coda namespace hierarchy with `obj`'s DRR subtree. It must be pinned in the client cache until the relevant conflicts are resolved because fetching its server replica will cause the DRR subtree to be orphaned.

- The object that occupies the original location of `obj` in the Coda namespace is called the *fake-root* of the DRR subtree. It uses a fake `fid` that does not correspond to any real object in the file system. Its main functionality is to serve as a place-holder directory to host the local and global subtrees of `obj`. Note that fake-root is a read-only directory.

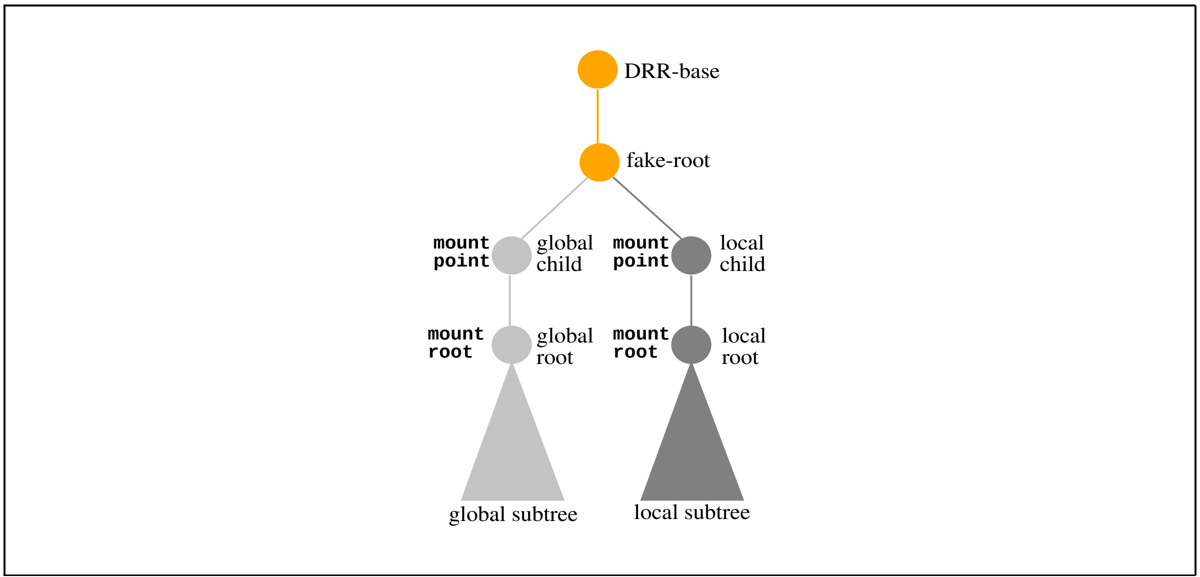


Figure 5.6: Internal Structure of Dual Replica Representation

- *local child* is a temporary object with a fake `fid`. It is the child of fake-root named "local" and serves as the mount-point for the local subtree of `obj` to be hooked with the DRR subtree.

- Similarly, *global child* is also a temporary object with a fake `fid`. It is the child of fake-root named "global" and is used as the mount-point for connecting the global subtree of `obj` with the DRR subtree. Since the top three nodes play the role of a joint linking the `obj`'s local and global subtrees with its parent and they all use the fake `fid`, we call them the *fake-joint* of the DRR subtree.

- *local root* corresponds to the local cache copy of `obj`. When `obj` is a file or a symbolic link, its local subtree only contains the local root itself. When `obj` is a directory, local root is the root of the local subtree of `obj`. The mounting of the local subtree to the local child is lazy. In other words, it is deferred until the resolver tries to access the local replica of `obj`.

- Similarly, *global root* corresponds to the server replica of `obj`, which is root of the global subtree of `obj` if it is a directory. The mounting of the global subtree to the global child is also lazy.

A DRR Subtree Table In order to keep track of all the inconsistent objects and their associated DRR subtrees, we maintain a persistent table storing information of all the DRR subtrees. Each entry of the table contains key information such as `fids` of the top six nodes of a DRR subtree. This table plays an important role in situations where the detection of new conflicts or the resolution of existing conflicts require adjustment to existing DRR subtrees, as will be explained later in this section.

A Local/Global Fid Map In the dual replica conflict representation, the global replica of an object retains its original `fid` while the local replica must use a generated local `fid` so that it can belong to the special local volume. Since the maintenance of DRR subtrees needs the correspondence between local and global replicas, we maintain a persistent table to provide this mapping.

5.3.2.2 DRR Subtree Construction

Building the Fake-Joint To create a DRR subtree for an inconsistent object `obj`, the first step is to locate the cache copies of `obj` and its parent denoted `p(obj)`. After de-linking the parent/child relation between `p(obj)` and `obj`, we fashion a new directory with a fake `fid` and use it as the fake-root by making it a child of `p(obj)` with the original name of `obj`. In addition, two mount-point objects with fake `fids` are manufactured and inserted as the children of fake-root named “local” and “global” respectively.

Creating the Local Subtree The second step is to create the local subtree of `obj` by traversing the cache copies of objects that are descendants of `obj`. For each traversed cached object, we generate a new local `fid` and use it to replace its original `fid` and insert the pair of `fids` into the local/global `fid` map. This step is called *localization* because the `fid` replacement effectively *localizes* the entire cached subtree rooted at `obj`. In other words, it virtually copies the cached subtree of `obj` into the local volume. Note that the cache manager guarantees that the ancestors of any cached object are also cached.

Forming the Global Subtree Because the global replica of each individual object retains the original `fid`, the formation of the global subtree of `obj` becomes a natural result of the demand fetching and cache replacement activities performed by the IOT-Venus. The global subtree is mounted when the server replica of `obj` is first accessed by the resolver. The server replicas within the global subtree are brought to the client cache only when they are accessed by the resolver. At any given moment, the global subtree may be partially or fully cached depending on the activities performed by the resolver and the IOT-Venus.

5.3.2.3 DRR Subtree Destruction

When an invalidated transaction is resolved, its associated DRR subtrees need to be discarded immediately. The first step is to de-link fake-root from DRR-base, unmount the local and global subtrees, and remove the three fake-joint objects. The second step simply restores the parent/child linkage between DRR-base and global root and discards those local replicas that are not pinned by any other live transactions.

However, throwing away all the un-pinned local replicas could be very wasteful when the subtree contains a large number of nodes and few of them are involved in conflicts. This problem is particularly acute on a mobile client operating in a weakly connected environment with low network bandwidth because re-fetching many of the discarded objects that are unchanged on the servers takes a long time to complete. We address this problem by recovering local replicas that have not been mutated and whose corresponding server replicas are not cached. Their original `fids` will be restored and their cache status will be marked as questionable. The next time they are accessed, the client only needs to validate their cache status instead of fetching their data content as long as they are not updated on the servers.

5.3.2.4 DRR Subtree Maintenance

When new conflicts are detected or old conflicts are resolved, some existing DRR subtrees need to be adjusted.

Subtree Merge Suppose that an inconsistent object $\mathtt{obj}_1$ is represented by a DRR subtree $\mathtt{ST}_1$ and one of its ancestors $\mathtt{obj}_2$ is later detected to be in conflict. Because of the subtree representation, the local and global subtrees of $\mathtt{obj}_2$ contain those of $\mathtt{obj}_1$ respectively. Thus, creating the DRR subtree of $\mathtt{obj}_2$, denoted $\mathtt{ST}_2$, requires a merge with $\mathtt{ST}_1$. The construction of $\mathtt{ST}_2$ can proceed with the normal steps except when the localization encounters the DRR-base of $\mathtt{ST}_1$. The fake-joint of $\mathtt{ST}_1$ is removed and its local subtree is merged with the partially formed local subtree of $\mathtt{ST}_2$ before the localization resumes. The DRR subtree table still maintains an entry for $\mathtt{ST}_1$ but marks it as a *covered* subtree.

Subtree Split Suppose that the above two inconsistent objects $\mathtt{obj}_1$ and $\mathtt{obj}_2$ belong to two independent, invalidated transactions $\mathtt{T}_1$ and $\mathtt{T}_2$, respectively, and $\mathtt{T}_2$ is resolved before $\mathtt{T}_1$. This requires $\mathtt{ST}_2$ to be discarded and $\mathtt{ST}_1$ to be recovered because $\mathtt{obj}_1$ remains inconsistent. The destruction of $\mathtt{ST}_2$ can proceed with the regular procedure except that the original local subtree of $\mathtt{ST}_1$ needs to be split out from that of $\mathtt{ST}_2$ and the full DRR subtree of $\mathtt{ST}_1$ must be re-established based on recorded information in the DRR subtree table.

5.3.3 The Multiple View Capability

The Need for Multiple Views Conflict resolution often needs to apply existing Unix applications and tools on replicas of relevant inconsistent objects. However, the proper functioning of many applications such as `make` requires involved objects to be positioned in specific locations within the namespace hierarchy. In other words, the pathnames of the relevant objects must satisfy certain configuration requirements. For example, system buildings using `make` often require a particular structure for source and object directories. This requirement is violated by the insertion of "local" and "global" into pathname in DRR subtrees.

Our solution is to provide the ability for resolvers to view global or local replicas of inconsistent objects in their original locations. We provide three different views. The default is called *mixed view* which is the canonical form of DRR subtree where both local and global replicas are accessible with twisted pathnames. The resolver can select the *global view* to access the global replica of an inconsistent object `obj` using its original pathname. It can also choose the *local view* to view the local replica of `obj` at its original location. The multiple view capability is the key to allow existing applications to be utilized for conflict resolution.

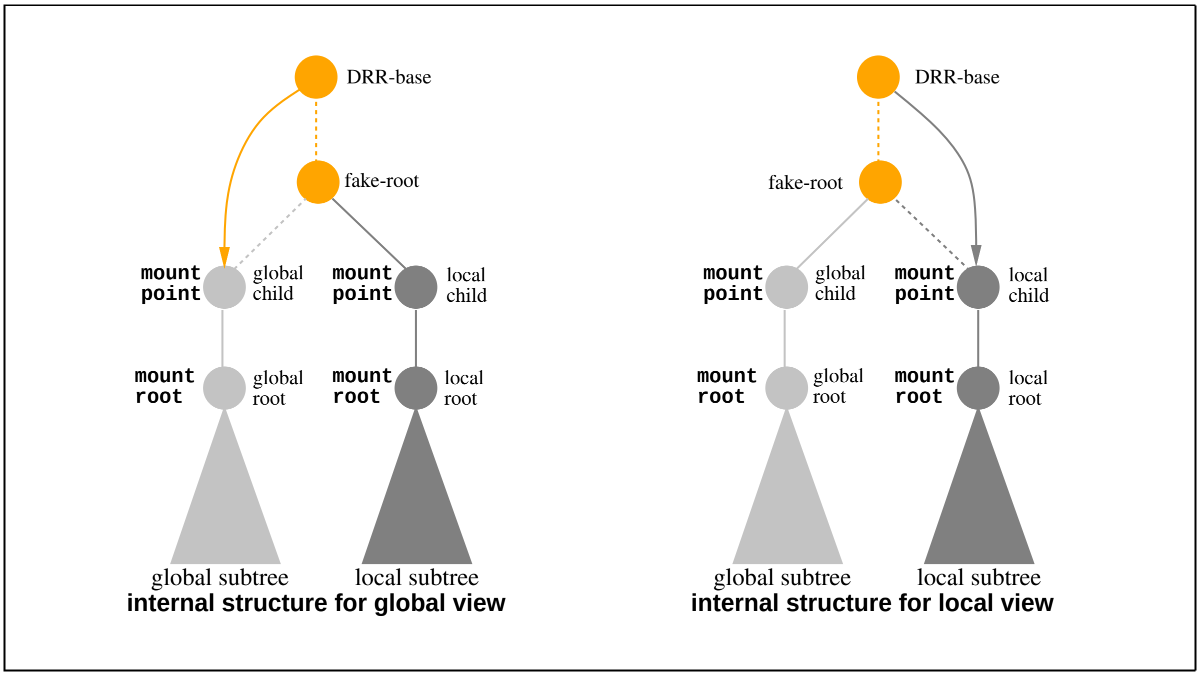


Figure 5.7: The Internal Structure of Local and Global Views

The Global View Intuitively, the global view allows the resolver to *see through* the local mutations of live transactions and access the server replica of selected objects in their original namespace locations. It can be used in several different ways to support conflict resolution.

First, automatic transaction re-execution only needs to access the global replicas of relevant inconsistent objects. Setting the global view for their DRR subtrees would allow the corresponding server replicas to be accessed using their original pathnames while none of the local replicas are visible.

Second, the automatic execution of an application-specific resolver for an invalidated transaction `T` may need to access objects that `T` originally did not access. If some of them are currently inconsistent due to other invalidated transactions, their global state must be exposed to the resolver and this can be accomplished by setting the global view for the corresponding DRR subtrees.

Third, the global view capability allows the users to access the global state of certain inconsistent objects to serve immediate needs without having to resolve the conflicts first.

The Local View The local view allows the local replica of an inconsistent object to be viewed with its original pathname. Although not quite as useful as the global view, it can be used in the situations where the local state of certain inconsistent objects are needed to serve immediate purposes without having to resolve the conflicts. This is particularly useful for mobile clients operating in a weakly connected environment where resolving conflicts is too time consuming due to fetching needed global replicas over slow connections.

The Realization of Multiple Views The implementation of the global and local views are almost identical. The key is to alter the internal structure of a DRR subtree so that its global or local subtree is directly hooked with the DRR-base. Figure 5.7 depicts the updated internal structures for both views. To create the global view, the IOT-Venus will temporarily de-link the parent/child relations between DRR-base and fake-root as well as fake-root and global child, as indicated by the dotted lines in the picture. In addition, it establishes a temporary parent/child linkage between DRR-base and global child, which enables the global subtree to be accessed at the original location of the inconsistent object. The local view can be created in the same manner.

5.3.4 Establishing Transaction Resolution Object View

At the start of conflict resolution of an invalidated transaction `T`, the IOT conflict representation mechanism will perform *resolution initialization* to create an appropriate object view for the corresponding resolver. At the conclusion of the resolution, *resolution finalization* is performed to discard the relevant DRR subtrees, and to adjust and restore object views according to the resolution outcome.

Two Basic Requirements First, the resolver must be able to access both the local and global replicas of any object in `inc(T)`. In addition, the local replicas must contain the original content that was last accessed by `T`. This is necessary for `T`'s resolver to find out what `T` has performed locally and what has been changed on the servers. Furthermore, the resolver needs to use the corresponding global replicas to store resolution results. Second, for any object not in `inc(T)`, only its global state should be visible to the resolver. This guarantees that the resolution outcome will not depend on any other live transactions.

Mixed View for Accessed Inconsistent Objects To satisfy the first requirement, resolution initialization will create a DRR subtree for each object in `inc(T)`. By setting the mixed view for the relevant DRR subtrees, the resolver can access the local and global replicas of any object in `inc(T)`. Resolution finalization will discard these DRR subtrees immediately after the resolution succeeds.

There are two important issues to mention here. First, the local replicas of relevant DRR subtrees must reflect the data content that was originally accessed by the execution of `T`. For any object $\mathtt{obj} \in \mathtt{inc(T)}$ updated by subsequent transactions, resolution initialization will restore its original content using the following method. If `obj` is a file object and was overwritten by a later transaction, the transaction system can restore its original content by binding `obj` to the corresponding shadow cache file in `shdset(T)` which saved the data last accessed by `T`. If `obj` is a directory object, its original content can be restored by performing the inverse operation for every directory mutation operation performed on `obj` by the subsequent transactions. For example, if a new object `foo` is created under `obj`, the inverse operation is to remove `foo` from under `obj`. Finally, if `obj`'s attributes were updated by subsequent transactions, the original values can be restored using the corresponding attribute values recorded in `TML(T)` or the *initial attributes* stored in the internal representation of `obj`. Note that a reverse process needs to be performed by resolution finalization.

Second, transaction `T` may have multiple inconsistent objects and some of them may have ancestor/descendant relationships. In order to avoid unnecessary DRR subtree merges, resolution initialization will first compute the set $\mathtt{root(T)} = \{\mathtt{obj} \mid \mathtt{obj} \in \mathtt{inc(T)} \wedge \mathtt{obj}$ does not have any ancestor in $\mathtt{inc(T)}\}$. Creating a DRR subtree for each object of `root(T)` can cover the entire `inc(T)` without any subtree merges.

Global View for Other Inconsistent Objects When `T` is being resolved, there could be other inconsistent objects that are not in `inc(T)`. The resolution initialization for `T` needs to construct DRR subtrees for those inconsistent objects and set global view for them so that only their global state is visible. Because the resolution of `T` tends to only access objects it has already accessed, the actual creation of the global view DRR subtrees for those inconsistent objects can be performed lazily until they are accessed by the resolver. Resolution finalization will revert their representation back to the dangling symbolic link form.

Hiding Other Local Mutations If there are pending transactions when `T` is about to be resolved, resolution initialization needs to hide all the local mutations made by those transactions so that only the corresponding global state is visible. This can be accomplished by creating a global view DRR subtree for each dirty-local object. Similarly, the actual creation of the global view DRR subtrees can be delayed until the resolver tries to access the corresponding dirty-local objects. Resolution finalization will simply discard all the DRR subtrees associated with dirty-local objects.

Optimization for Automatic Re-execution Because automatic transaction re-execution does not need the local state of any inconsistent object, resolution initialization will simply set global view for all DRR subtrees that are created.

Chapter 6

Detailed Design: Conflict Resolution

Conflict resolution is the process of restoring consistency to the inconsistent objects updated by an invalidated transaction. This chapter presents detailed designs for realizing the four conflict resolution options provided by the IOT consistency model. We first describe a conflict resolution framework based on the cooperation between the transaction system and the resolver. We then discuss the basic mechanisms that support both automatic and manual conflict resolutions.

6.1 A Cooperation-Based Resolution Framework

Resolving conflicts in general requires knowledge about the application associated with the invalidated transaction being resolved, referred to as the *target transaction* in the rest of the discussion. Resolution actions that understand application semantics must be supplied by either a pre-programmed resolver or a human user. Our main design objective is to minimize the burden on the resolver to enhance the practical usability of the IOT conflict resolution mechanisms. To that end, the transaction system must fully cooperate with the resolver by performing those resolution actions that do not require application-specific knowledge. This section focuses on establishing a cooperation-based resolution framework where the transaction system and the resolver are each responsible for a core set of resolution actions necessary for supporting the four IOT conflict resolution options.

6.1.1 A Resolution Session Model

Resolution Session and Its Process Model As discussed in Chapter 4, invalidated transactions are resolved one at a time. We use the term *resolution session* to refer to the process of resolving an invalidated transaction. Sometimes, a resolution session is also called a *repair* *session* if the resolution is performed manually. Figure 6.1 presents a resolution session process model describing the main control flow.

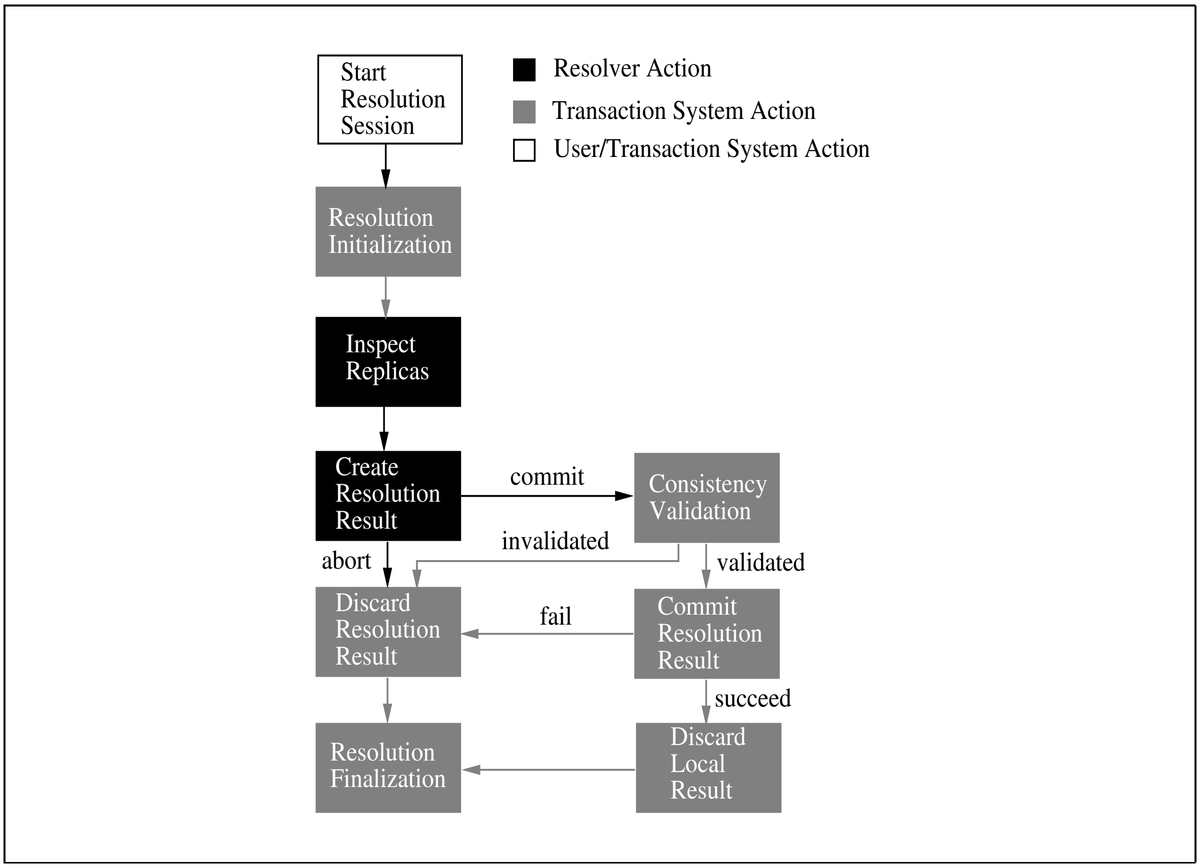


Figure 6.1: A Cooperation-Based Resolution Session Model

Transaction Encapsulation The cooperation-based session model requires an entire resolution session to be treated as if it is a connected isolation-only transaction. Such transaction encapsulation provides three key properties. First, the mutations used by the resolver to create the resolution result are performed only locally and will not be visible on the servers until the session is successfully completed. Second, a resolution session must be validated in the same way as a normal transaction to guarantee that the resolution result is consistent with the most recent server state. Third, the resolution result is committed to the servers atomically. These properties are very important for conflict resolution to achieve its ultimate goal of restoring consistency for the inconsistent objects caused by the target transaction. Conceptually, a resolution session is just another computation and is subject to the inconsistency problems caused by partitioned sharing. Transaction encapsulation enables the resolution result to obtain the same level of consistency guarantees provided by the IOT model.

Basic Resolution Actions As shown in Figure 6.1, each resolution session must go through a sequence of basic steps:

- **Start Resolution Session**

  The beginning of a resolution session is triggered either implicitly by the propagator thread descried in Chapter 4 in an attempt to automatically resolve the target transaction, or explicitly by the user using the *transaction repair tool* to be discussed later in this chapter.

- **Resolution Initialization**

  The purpose of resolution initialization is to create the resolution object view described in Chapter 5, so that only the original local state accessed by the target transaction and the up-to-date global state are visible.

- **Inspect Replicas**

  The first step a resolver usually undertakes is inspecting the relevant local and global replicas to find out the cause of inconsistency (if any) and decide the resolution actions that can restore consistency.

- **Create Resolution Result**

  The central act of a resolution session is when the resolver performs the necessary computation to create the resolution result. As in connected transaction execution, all the mutation operations issued by the resolver are logged and performed locally. Their effect is not visible on the servers until the resolution session is successfully completed.

- **Consistency Validation**

  New updates on the servers during a resolution session may cause the resolution result to be in conflict with the latest global state. Consistency validation safeguards the validity of the resolution outcome by ensuring that none of the objects accessed by the resolver have been updated on the servers during the resolution session.

- **Commit Resolution Result**

  This step gathers all the logged mutations performed by the resolver and atomically commits them to the corresponding servers.

- **Discard Resolution Result**

  In case the resolution session needs to aborted due to failures in consistency validation or result commitment, all the mutations performed by the resolver will be discarded immediately so that the local client state appears to the users as if this resolution session has never happened.

- **Discard Local Result**

  After the resolution result has been successfully committed to the servers, the original local result of the target transaction is no longer needed and is immediately thrown away.

- **Resolution Finalization**

  As described in Chapter 5, the main responsibility of resolution finalization is to properly restore and adjust object views to reflect the resolution outcome.

Model Generality The four IOT conflict resolution options can be fully accommodated by the resolution session model, although not all of them need to go through every step.

- **Automatic ASR Execution**

  The automatic execution of an application-specific resolver is the most powerful resolution option and can be realized by the transaction system automatically invoking a user-supplied resolver immediately after resolution initialization. The resolver will typically inspect replicas and create the resolution result using application-specific knowledge. Upon termination of resolver execution, an `exit` status of zero indicates that the resolver actions have succeeded and the transaction system will immediately commit the resolution result. Any non-zero `exit` status will cause the resolution session to be aborted.

- **Automatic Re-execution**

  The resolution session model regards automatic transaction re-execution as a special case of the previous option where the resolver is the transaction itself. The only difference is that the resolver starts creating the resolution result right away without examining the relevant local replicas.

- **Automatic Abort**

  The automatic abort option can be accomplished by the transaction system alone performing only the steps that discard the local result of the target transaction and adjust object views accordingly. Note that this option is equivalent to a resolver that does not perform any mutation.

- **Manual Repair**

  The default resolution option is supported by a stand-alone transaction repair tool whose details will be provided at the end of this chapter. After the user initiates a repair session with the `begin_repair` command, the transaction system will perform resolution initialization so that the user can repair the conflicts by directly operating on the relevant objects. When the user issues a `commit_repair` or `abort_repair` command, the transaction system will go through the necessary steps to commit or abort the repair session respectively.

6.1.2 Supporting Application-Independent Resolution Actions

Detailed designs on supporting a resolver to perform application-specific resolution actions will be presented in the next two sections. This section focuses on the basic mechanisms that are application-independent and performed by the transaction system to support the various steps in the cooperation-based resolution model described in Figure 6.1.

6.1.2.1 Handling the Resolution Result

Almost all resolution sessions involve creating, committing or discarding the resolution result. The proper handling of resolution mutations in the following three areas is the key to supporting these activities.

Logging Resolution Mutations The resolution result is created by the resolver performing mutation operations on three kinds of targets: the global replica of an inconsistent or local-dirty object, the cache copy of a clean-local object, or the cache copy of a consistent object. Because global replicas can be accessed in the same way as normal objects, all resolution mutations are treated uniformly. Similar to connected transaction execution, mutation operations are performed locally and logged in the CML of the corresponding volumes. Internally, the entire resolution session is treated as a special transaction and a unique transaction-id is used to identify all the CML records performed by the resolver. One thing worth pointing out here is that overwriting a clean-local file object by the resolver will cause a corresponding shadow cache file to be created if the object has live guardians other than the target transaction. This is because its local content needs to be saved for possible future resolution as explained in Chapter 4.

Committing Resolution Mutations The resolution result is committed to the servers using the underlying reintegration mechanism. Because a special transaction-id identifies all the logged resolution mutation operations, the commitment can be performed by the transaction reintegration process described in Figure 4.5. Although the resolution session model requires atomicity for result commitment, it is only guaranteed on a single-volume granularity as explained in Chapter 4. Since most transactions and therefore their corresponding resolution sessions only mutate objects within a single volume, either all or none of the resolution result will be globally visible under normal circumstances. If the underlying reintegration process fails because of lost server connections or new conflicts against the latest server updates, the resolution session will be aborted. After the resolution mutations are successfully reintegrated, special care must be taken to restore the original state for any clean-local object that is updated by the resolver and has other live guardians. Such state restoration can be accomplished by the same methods described in Chapter 5, using shadow cache files and inverse directory mutation operations.

Discarding Resolution Mutations When a resolution session needs to be aborted, all the resolution mutations must be immediately discarded. For mutation operations performed on consistent objects or the global replicas of inconsistent and local-dirty objects, this can be accomplished by simply throwing away their cache copies and the corresponding CML records. For mutation operations performed on clean-local objects, the transaction system must similarly restore their original state prior to the resolution session as mentioned above. The corresponding CML records are discarded afterwards.

6.1.2.2 Handling the Local Transaction Result

Discarding Local Transaction Result Ultimately, any invalidated transaction `T` will be successfully resolved and its original local result needs to be discarded. The first step is to eliminate `T`'s effect on the cache copy of objects in `W(T)`. If an object `obj`$\in$`W(T)` will become a consistent object again after resolution, its cache copy is discarded and fetching the server replica will wipe out `T`'s mutation on `obj`. If `obj` has other live guardians, the transaction system needs to revert its local state to the one prior to the resolution session if resolution initialization had adjusted the local replica of `obj` to represent the state last accessed by `T`. This can be achieved through a process that is the inverse of the one described in Chapter 5 for restoring previous object state. The second step simply throws away all the CML records belonging to `TML(T)`.

Preserving Local Transaction Result The ASR conflict resolution option can be used for application-specific consistency re-validation where a resolver utilizes application semantics to determine whether the local result of the target transaction is actually consistent with the up-to-date global state. If such re-validation succeeds, the local result can be reused and committed to the corresponding servers as is. To facilitate application-specific consistency re-validation, the transaction system provides a special *preserve-operator* (an IOT library call) that can be used by the resolver to preserve the local result of the target transaction. The operator automatically replays all the logged mutation operations performed by the target transaction on the corresponding global replicas. Note that such replay may fail because some of the operations could be in conflict with the latest server state.

6.1.2.3 Other Issues

Resolution Initialization/Finalization Resolution initialization is responsible for creating an appropriate object view for the resolution session, while resolution finalization is responsible for restoring and adjusting object views based on the resolution outcome. Details about these two steps have been presented in the previous chapter. There is a minor adjustment in the object view for a manual repair session which will be discussed in the last section of this chapter.

Resolution Consistency Validation The transactional encapsulation of a resolution session requires the transaction system to automatically record all the objects that are accessed by a resolver in the same way as recording the transaction readset and writeset. Since the consistency validation of a resolution session is intended to ensure that none of those objects have been changed on the servers during resolution, the transaction certification technique discussed in Chapter 4 can be directly applied for this purpose.

6.1.3 Extending Transaction State Transitions

To fully account for the difference between automatic and manual resolution, as well as the handling of resolution failures, the original transaction state transition model described in Figure 3.3 needs to be extended, as shown in Figure 6.2. The original resolving state is expanded into several new states. Their meanings and state transitions are discussed below.

To-be-resolved State When a pending transaction `T` is invalidated and has chosen one of the automatic resolution options, it goes into the *to-be-resolved* state. The transition from the pending state to the to-be-resolved state can be triggered by any event that would make it known to the transaction system that there is an object whose global version cannot possibly be equal to the local version that `T` has accessed. For example, it can happen when `T` fails the GC validation, when an object accessed by `T` is updated on the corresponding server, or when the resolution of an earlier transaction that `T` read from causes `T` to have accessed an object value that no longer exists. `T` will remain in the to-be-resolved state as long as it is not fully connected or has at least one predecessor in the `SG`. As soon as it is fully connected, all its predecessors

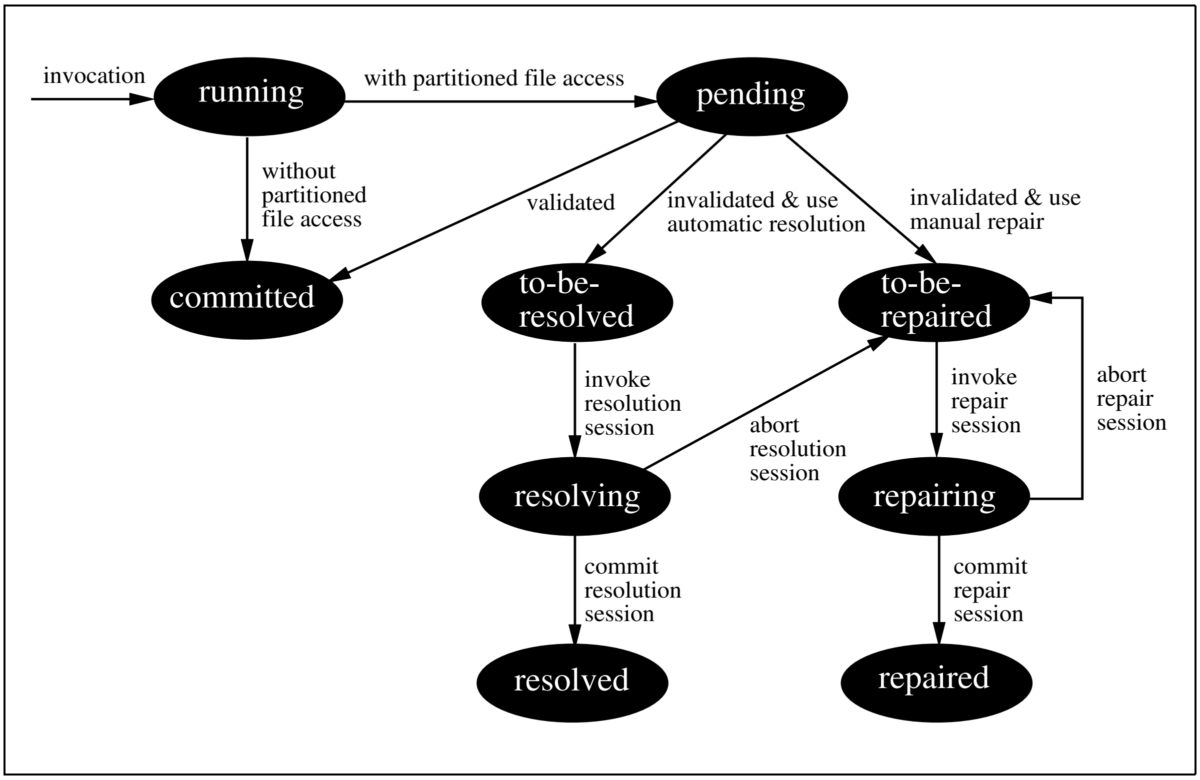


Figure 6.2: Extended IOT State Transitions

are resolved or committed, and there is no other transaction execution or resolution going on, `T` will transit into the *resolving* state and start its automatic resolution session.

To-be-repaired State Similar to the to-be-resolved state, a pending transaction `T` goes into the *to-be-repaired* state when it is invalidated and has selected the manual resolution option. The transition from the pending state to the to-be-repaired state can be triggered by the same events described above. `T` will go into the *repairing* state when the user successfully invokes a repair session for it using the repair tool. More details about starting a repair session are presented at the end of this chapter.

Resolving State A transaction `T` remains in the resolving state as long as its automatic resolution session is still going on. If the session is successfully committed, `T` goes into the *resolved* state. Otherwise, the transaction system will force `T` to be manually repaired by transiting it into the to-be-repaired state.

Repairing State Similarly, a transaction `T` stays in the repairing state as long as its manual repair session continues. If the repair session is successfully committed, `T` goes into the *repaired* state. Otherwise, it goes back to the to-be-repaired state and the users have to start another repair session to repair it again.

Resolved and Repaired State Both the resolved and the repaired states are conceptually identical to the committed state in that they are the final state of a terminated transaction. Once again, for the purpose of enhancing the practical usability of IOT, we decided to use different states to inform the users about the different paths the transaction has gone through in its lifecycle.

6.2 Automatic Conflict Resolution

The ability to resolve conflicts automatically is vital to the overall practicality of the IOT conflict resolution mechanisms. This section concentrates on important design issues related to the automatic execution of a resolver, whether it is a pre-programmed application-specific resolver or the target transaction itself. Note that the original Coda file system provides a different application-specific resolution mechanism for resolving write/write conflicts among server replicas for individual objects [29]. The new automatic resolution mechanisms discussed in this section are specifically designed for resolving conflicts caused by invalidated transactions as required by the IOT consistency model.

6.2.1 Site of Resolver Execution

A fundamental design issue about automatic conflict resolution pertains to the site of resolver execution. There are two main choices: on the client machine where the target transaction was originally executed or on a selected server machine. We decided to use the same strategy adopted by earlier work in Coda [32] and execute the resolver on the client machine for the following reasons. First, allowing arbitrary resolvers to be executed on a server machine would violate the Coda security model's basic requirement that server machines only run trusted software. The second reason is for maintaining system scalability. Since the resource cost of resolver execution could be significant for many applications, overall system scalability is better preserved by shifting such burden to the client machines. The third and the most important reason is that much of the supporting machinery such as conflict representation is only present on the client machine. Furthermore, the server machines may not be able to execute resolvers at all if they use different architectures (CPU and operating system).

6.2.2 Resolver Invocation

Recording Necessary Information In order to automatically resolve the target transaction `T`, the transaction system must record the following information when `T` is initiated.

- The resolution option chosen by `T`, denoted `opt(T)`.
- The pathname of the application executable file of transaction `T`, denoted `app(T)`. This is necessary when `opt(T)` is automatic re-execution since the resolver is just `app(T)` itself.
- The pathname of the user-supplied resolver executable file, denoted `asr(T)`. This is necessary when `opt(T)` is automatic ASR execution.
- The original execution environment of `T` including the command line arguments (denoted `argv(T)`), the environment variable list (denoted `env(T)`), the `umask` value of the master process (discussed in Chapter 3) of `T` (denoted `umask(T)`), and the pathname of the working directory (denoted `pwd(T)`).

Restoring Original Execution Environment The automatic invocation of the resolver for target transaction `T` must be performed under an appropriate environment. Our design restores the recorded original environment of `T`. The rationale behind this decision is to regard the automatic resolution of `T` as an alternative computation for achieving the original goal of `T` under a different system state. Because any slight difference in execution environment could result in drastic changes in the computation outcome, restoring the recorded execution environment before the resolver invocation is necessary to provide an identical starting point. Immediately before the invocation starts, the transaction system will `cd` into `pwd(T)`, set the environment variable list to `env(T)` and pass `argv(T)` as the command line arguments to the resolver.

Process Structure Automatic resolution sessions are initiated by the propagator thread. To invoke the resolver, the propagator thread spawns a subprocess and uses the `exec` system call to launch the resolver after appropriately closing the open file descriptors. Note that the propagator thread cannot use the `wait` system call to await the completion of the resolution process because that will cause the entire Venus process to be blocked. Instead, it will yield and sleep until it is awakened up by the Venus signal handler when it catches a SIGCHILD generated by the termination of the resolution process. The file access operations on Coda objects performed by the resolver are serviced by a worker thread after the corresponding requests are passed through the kernel, as shown in Figure 6.3.

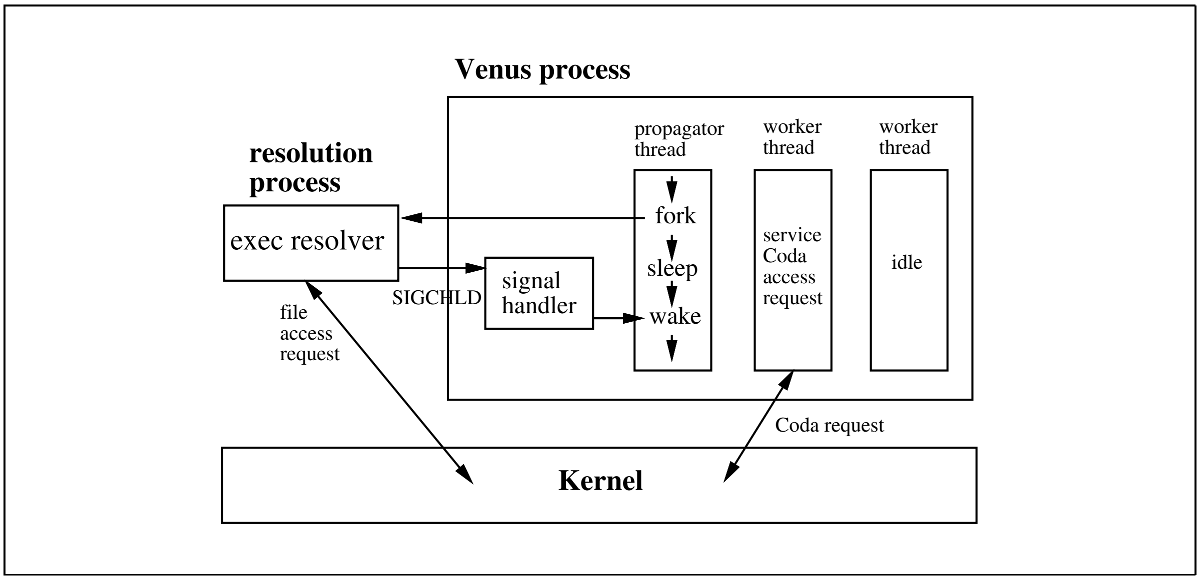


Figure 6.3: The Process Structure of Automatic Resolver Execution

6.2.3 Resolver Execution

6.2.3.1 Transactional Encapsulation

The automatic execution of a resolver for the target transaction `T` is performed within the scope of a special isolation-only transaction denoted `res(T)`. The transaction system manipulates the internal representation of `res(T)` so that it implicitly inserts the `begin_iot` and `end_iot` calls before the `exec` and after the `exit` of the resolution process respectively. The entire execution of `res(T)` is performed in a manner similar to a normal connected transaction. All the file access operations performed by the resolver are recorded in the readset and writeset of `res(T)`. In addition, mutation operations are performed locally and logged in the CML of the corresponding volumes.

6.2.3.2 Handling Failures

The only differences between the execution of `res(T)` and a regular connected transaction are the way in which failures are handled.

Validation Failure The consistency of an automatic resolution session for target transaction `T` is validated by performing the OCC validation for `res(T)`. For a normal connected transaction, the response to OCC validation failure is automatic abortion and re-execution. However, this approach is not suitable for `res(T)` because the automatic resolver execution holds exclusive control over many client resources and automatically re-executing `res(T)` could deny the users needed services for a long period of time. For simplicity, our design aborts the result of `res(T)` as well as the whole resolution session and conservatively forces the users to manually repair `T` by transiting it into the to-be-repaired state.

Abnormal Termination For a regular connected transaction, the termination of its execution means the completion of the transaction regardless of the `exit` status. With respect to `res(T)`, however, any abnormal termination such as crashes due to segmentation fault, etc., must be treated as a failure and the resolution session must be aborted. Thus, it is very important for the resolver to carefully check for abnormal conditions such as getting an `ETIMEDOUT` error code when opening a Coda file, and exit with non-zero status appropriately.

6.2.3.3 Managing Interactive I/O

The Challenge of Interactive I/O Unix applications often employ interactive I/O to inform the users about the status of the execution and request feedback for further computation. Previous research has demonstrated that including interactive I/O into the transaction model is a very difficult problem and there is no general solution [54]. Handling interactive I/O during automatic resolution is even more of a challenge because the I/O environment is often different from when the target transaction was originally executed. The user who executed the target transaction may not even be logged in on the client machine while the resolution is going on. In general, it is very difficult for the transaction system to automatically select the proper devices for the resolver execution to conveniently interact with the users. The current IOT implementation only provides limited support for the standard input/output operations performed by the resolver so that automatic conflict resolution for typical Unix applications can be performed. Tackling the complete problem of interactive I/O in automated conflict resolution is beyond the scope of this dissertation.

Supporting Automatic Transaction Re-execution When the resolver is the target transaction itself, standard I/O during resolution is performed in the same way as during the original transaction execution. Our strategy of supporting standard output is to maintain resolution transparency while preserving output messages. The specific mechanism binds the standard output of the re-execution process to a special disk file so that a complete history of standard output during the resolution is recorded. The output messages are not visible to the users unless specifically requested. A monitoring tool is provided to continuously display the growing content of the special file for any user who wishes to closely follow the resolution process.

Standard input during automatic re-execution is much harder to deal with and there is no satisfactory solution. Although it is possible for the transaction system to create a designated window or pseudo-terminal for the users to type input data, resolution transparency has to be sacrificed because standard output messages must be made visible to prompt for user input. The current IOT implementation does not provide special support for standard input other than binding it to the terminal where the current Venus incarnation was initiated. Thus, the automatic re-execution resolution option should be used only for applications that do not need interactive input such as `make` and `gcc`. Fortunately, this is not a severe limitation in practice because Unix applications for which automatic re-execution is likely to be selected as the resolution option typically read input from files instead of interactively from the users.

Supporting Automatic ASR Execution The default support for interactive I/O during the automatic execution of an application-specific resolver is the same as for transaction re-execution. However, an application-specific resolver does not have to perform interactive I/O in the same way as its corresponding transaction. It has the complete freedom to manage interactive I/O in its own way. For example, the resolver can take advantage of existing facilities such as `tcl/tk` [52] to dynamically create windows for displaying messages to the users and reading input data from them. An actual example using this approach will be presented in Chapter 9. In summary, the transaction system only provides support for resolvers that perform standard output but not standard input. If reading interactive input is a necessity for a resolver, it must manage its own standard input and output together.

6.2.3.4 Local Concurrency Control

Strict local concurrency control is imposed to protect a resolution session from being interfered with by the execution of other ongoing transactions. New isolation-only transactions are not allowed to start until an ongoing resolution session is completed. Furthermore, all the cached volumes are write-locked by the resolution process.

6.2.4 Safety Issues

Enforcing safety is a critical issue for automatic conflict resolution because an application-specific resolver or a target transaction itself is not a piece of trusted software. A wide range of catastrophes from simple coding mistakes to full-fledged Trojan horse attacks have to be guarded against. The problem is made more difficult because resolution is performed transparently. In other words, a user may be completely unaware of the damages caused by an erratic or even malicious resolver. We address the following three resolution safety issues using measures similar to those used in previous Coda research [32].

Security To limit the potential damage that can be done by the automatic execution of a resolver, the transaction system provides two levels of security protection. At the first level, we employ the `setuid` mechanism so that the execution of a resolver only has the privilege of the user who originally invoked the target transaction. Thus, the potential damage is limited to those portions of the Coda namespace where the target transaction's invoker has update rights.

The second level provides control over which resolvers can be automatically executed. The transaction system will only execute resolvers residing at certain trusted locations. Users and system administrators can store trusted resolvers in designated directories where only they have the privilege to make changes and provide the transaction system with the pathname of those directories. The transaction system will verify that the resolver is from one of the trusted directories before automatically executing it. This scheme can be further extended to include a fingerprinting mechanism [73] to detect tampering of resolvers.

Robustness Erratic resolvers with programming errors can seriously degrade the robustness of a client. For example, a resolver trapped in an endless loop can hold the client hostage forever without allowing the users to obtain needed services. As another example, a resolver waiting for the user to provide input data without properly informing them will also lead to prolonged service denial. The transaction system addresses this issue by limiting the total elapsed time of resolver execution. Because no statically chosen time limit can meet the demand for all possible resolution tasks, the key is to impose a limit that is proportional to the expected amount of work needed by the resolution task at hand. The best estimation known to the transaction system is the target transaction's original execution time. Since the resolver is expected to spend some extra time probing relevant local and global replicas, we set the time limit of resolver execution to be twice that of the original transaction execution time. There is a Venus daemon thread that periodically checks the time limit for the resolution process. If its limit is exceeded, the transaction system will stop the resolver execution and abort the resolution session.

Another measure that can enhance the robustness of automatic resolution is to periodically check the progress of the resolution process. If the resolver stays idle for more than a threshold value, there is reason to believe that it may be waiting for user input. Warning messages can be displayed by the `codacon` tool [60] so that the users watching the monitoring window are notified.

Atomicity Resolver execution can crash for various reasons such as coding errors, abnormal conditions and client machine crashes. Coping with partial resolution results can be very messy and needs failure atomicity support. Globally, the reintegration mechanism assures that the resolution result will show up on the servers atomically. Locally, the partial resolution result left behind by a resolver crash is automatically cleaned up by the relevant steps of the resolution session. If the client machine crashes while the resolution is going on, special cleanup is performed during the ensuing IOT-Venus start-up. The transaction system will analyze the persistent image of the internal system state to discover that a resolution session did not complete because of the machine crash. It will automatically abort the unfinished resolution session and transit the target transaction into the to-be-repaired state.

6.2.5 Programming Application-Specific Resolvers

As a fundamental component of the IOT consistency model, executing application-specific resolvers plays a key role in automatic conflict resolution. The viability of the application-specific resolution approach very much depends on how effectively the task of programming the resolver of a given target application can be accomplished. In order to make resolver programming as convenient as possible, the overall IOT design envisions a basic paradigm within which resolvers can be methodically developed for their corresponding target applications. Such a paradigm relies on a set of basic assumptions and is supported by the combination of various system components as well as specially designed facilities.

6.2.5.1 Programming Model

Basic Assumptions A fundamental assumption about resolver programming is that the developers of a resolver must possess intimate knowledge about the internal details of the target application. The best candidates for writing a resolver are the target application's original developers or installation site maintainers. For other programmers to develop a resolver for an existing application, they need to devote a significant amount of time to learn the internal mechanisms of the application. In our experience of writing resolvers for some typical Unix applications, much of the time is spent on studying the source code of the applications. The availability of the source code of a target application is essential to its resolver development.

Scope Isolation The incremental transaction propagation framework described in Chapter 4 is designed very much with the ease of resolver programming in mind. Invalidated transactions are resolved one at a time and the target transaction does not depend on any other live transactions when it is being resolved. In addition, the conflict representation mechanisms guarantee that the effect of other live transactions are not visible during the resolution. Therefore, the scope of inconsistency is greatly reduced since the resolver is relieved from the burden of considering complicated interactions between the target transaction and other live transactions. Logically, the resolver can regard the target transaction as the only one performed during the disconnected operation session. Hence, the task of a resolver is confined to investigating what the target transaction has done locally and what has been changed globally, determining the nature of inconsistency (if any), and performing the necessary actions to restore consistency.

Object View The design of the IOT conflict representation is directly aimed at supporting resolver programming. The in-place dual replica representation form allows the resolution result to be created via direct mutations on the relevant global replicas, eliminating the need for the resolver to use an extra workspace for resolution computations. The multiple view capabilities enable the resolver to utilize existing software tools and packages to resolve conflicts more effectively. Finally, the automatic elimination of the target transaction's local result further reduces the resolver's responsibilities.

Overall Structure For brevity, we use `APP` and `asr(APP)` to denote the target application and its application-specific resolver respectively. A key design rationale in supporting resolver programming is to regard the task of a resolver as performing an alternative computation to achieve the original goal of the target transaction under a different system state. This enables `asr(APP)` to be programmed with an overall structure that parallels that of `APP`. The main control flow of `APP` can usually be decomposed into performing a sequence of specific tasks. For each task that `APP` would attempt under the new conditions (reflected by the automatically restored original execution environment and the relevant global replicas), `asr(APP)` can perform the following actions.

It first checks whether the same task has already been performed by the target transaction and whether the local result is still compatible with the global state. If so, `asr(APP)` can reuse the local result by copying it into the relevant global replicas. Otherwise, `asr(APP)` needs to perform the task using the up-to-date global replicas. If the original goal of the task needs to be adjusted due to changes in the global state, `asr(APP)` must come up with the new goal and perform the necessary computation to achieve it. Actions for compensating external side effects such as sending `email` are typically performed under such circumstances.

Several special resolution outcomes are worth mentioning. They are effectively degenerate cases. If the resolver execution reuses the entire local result of the target transaction, this is equivalent to a successful application-specific revalidation rescuing a syntactically invalidated transaction. If the resolution ends up performing every attempted task using the up-to-date global replicas, it is the same as resolution via automatic re-execution. If the global state is changed so much that `asr(APP)` does not attempt any task, the target transaction is effectively aborted.

6.2.5.2 Programming Facilities

The main support for resolver programming is embodied in the incremental transaction propagation framework and the way conflicts are represented to the resolvers. There are two groups of library routines designed to provide information and convenience to the resolvers. The first group includes routines that allow the resolver to test which objects are read or written by the target transaction. In addition, given the original pathname of a Coda object, there are routines that can translate it into the pathnames of the corresponding local and global replicas. The second group contains routines that enable the resolver to dynamically adjust object views. It also includes the preserve-operator that automatically *reproduces* the target transaction's local result on the corresponding global replicas. The specific details of these routines are presented in the next chapter.

6.3 Manual Conflict Resolution

Manual conflict resolution is the safety net of the IOT conflict resolution mechanisms. It is used not only as the default choice of resolution options but also the fall-back mechanism when other alternatives fail. This section first discusses the main areas where a manual repair session differs from an automatic resolution session and then describes a repair tool provided for manually repairing invalidated transactions.

6.3.1 Maintaining A Repair Session

Transaction Group The transaction system allows multiple transactions to be repaired in a single session because sometimes repairing a group of related transactions together is more effective and easier for the users to handle. Thus, each manual repair session is associated with a *transaction group* $\mathtt{TG} = \{\mathtt{T}_1.....\mathtt{T}_n\}$ $(n > 1)$ that satisfies the following two conditions. First, all the transactions in TG are fully connected and in the to-be-repaired state. Second, transactions in TG do not depend on any live transactions that are not in TG.

Identifying Repair Mutations A difficult problem in supporting a repair session is identifying which mutation operations are performed for the purpose of repairing conflicts. This is because manual repair actions can be issued by a user from different processes (e.g., from different windows) and it is quite possible for the user to perform some mutation operations that are completely unrelated to repairing conflicts. Since there is no reliable means for the transaction system to exactly identify the mutation operations belonging to the repair session, the current implementation includes all the mutation operations performed on those volumes originally updated by the target transactions. They will be performed locally and logged in the CML of the corresponding volumes with a special transaction-id.

Relaxed Consistency Validation Because a repair session is carried out interactively, the user may browse many files for other purposes during the repair session. It is highly inappropriate to abort a repair session just because the user read some irrelevant objects that are updated on the servers during the repair session. Thus, the original resolution consistency validation is relaxed so that it only requires those objects mutated during the repair session to remain unchanged on the servers.

Relaxed Local Concurrency Control Local concurrency control during a repair session still prohibits any new isolation-only transaction to be started. However, cached volumes are no longer write-locked and there is no limit on accessing Coda objects.

Repair Object View The users repairing a group of invalidated transactions are usually less knowledgeable about the internal details of the applications involved than the corresponding automatic resolvers. However, they typically know more about the current client state and other live transactions. Since providing full conflict information about other transactions generally helps the user to better repair conflicts, the mixed view is adopted for all the DRR subtrees during a repair session.

6.3.2 The Transaction Repair Tool

```
begin_repair <tid> [<tid>]
commit_repair
abort_repair
list_predecessor <tid>
list_successor <tid>
set_global_view <tid>
set_local_view <tid>
set_mixed_view <tid>
list_local_mutations <tid>
preserve_local_mutations <tid>
```

Figure 6.4: A List of Transaction Repair Tool Commands

We have developed a transaction repair tool that provides a set of commands supporting manual transaction resolution, as shown in Figure 6.4. The usage and functionality of these commands are as follows:

- The `begin_repair` command takes a list of transaction identifiers (integers) as arguments and starts a new repair session for the corresponding transactions. It will verify the necessary conditions required by a transaction group discussed above and perform the resolution initialization to create the repair object view.

- The `commit_repair` command tries to commit the current repair session. If the commitment fails, the repair session is automatically aborted. Because some of the inconsistent objects may remain inconsistent even after the session is successfully committed and cause confusion to the users, this command prints out a list of such objects and explains that they will remain inconsistent until some of their guardians are resolved or committed.

- The `abort_repair` command simply aborts the current repair session.

- The `list_predecessor` and `list_successor` commands take a transaction identifier as an argument and print out a list of `SG` predecessors or successors of the corresponding transaction respectively. The main purpose of these two commands is to provide information about the inter-dependency among live transactions.

- The `set_global_view`, `set_local_view` and `set_mixed_view` commands set the selected view for all the inconsistent objects corresponding to the transaction whose identifier is given as the argument.

- The `preserve_local_mutations` command replays all the logged mutations for the given transaction on the relevant global replicas. It is intended to support reusing the local result of a target transaction being repaired by the current session.

- The `list_local_mutations` command prints out the local mutation operations performed by the target transaction whose identifier is given as the argument.

Chapter 7

Detailed Design: User Interface

Presenting the IOT functionality to users and application programmers in a simple and easy-to-use manner is extremely important to the viability of the IOT model. This chapter describes an IOT programming interface that consists of a set of library routines, an interactive interface of a special C Shell, and related facilities.

7.1 Programming Interface

7.1.1 Interface for Programming Isolation-Only Transactions

7.1.1.1 Library Routines

Because the IOT service is intended as an extension to Unix file systems, there are two basic alternatives to presenting its programming interface: using new system calls implemented in the kernel or library routines at user level. We choose the latter for the following two reasons. First, placing the IOT programming interface at system level runs against the long-held tradition of keeping the system call interface intact while enhancing operating system services for various purposes. Second, a library interface is more convenient to implement and port to other platforms. Although the current interface is provided only in the C programming language, it can be extended to other programming languages straightforwardly if needed.

The IOT programming interface contains two basic routines indicating the beginning and the end of a transaction. Because of the need to specify a conflict resolution option, they are not mere syntactic tokens and carry crucial information from the application to the transaction system. The definition of the two routines is shown in Figure 7.1 and their usage and functionality are described below.

```
typedef struct { char opt, char *asr } iot_spec;
typedef struct { char **argv, char **env,
                 char *pwd, int umask
               } iot_env;
int begin_iot(iot_spec *, iot_env *);
int end_iot(int, iot_spec *);
```

Figure 7.1: Library Routines for Programming Transactions

The `begin_iot` routine is used to start a new transaction. Its return value is either a positive integer which represents the identifier of the newly created transaction or a negative integer which records an IOT-defined error code. The first argument specifies the conflict resolution requirement of the transaction. The `opt` component uses pre-defined constants to select the resolution option, and the `asr` component supplies the pathname of the resolver executable file if the selected option is ASR. The second argument provides the environment information necessary for automatic conflict resolution. The `argv` component is a list of strings containing the pathname of the transaction application executable file and the command line arguments. The `env` component is a list of strings representing the environment variable definitions. The `pwd` component records the pathname of the working directory where the transaction execution is started.

The `end_iot` routine is used to terminate a currently running transaction. A return value of zero means that the call succeeded. Otherwise it contains an IOT-defined error code. The first argument is the identifier of the transaction to be terminated, while the second argument is the same as the first argument of the `begin_iot` routine. Theoretically, `iot_end` is a better place than `begin_iot` to specify the resolution option because the application has already known what happened during the transaction execution when `end_iot` is called. However, from the reliability point of view, `begin_iot` is a more appropriate choice because the transaction execution may encounter abnormal situations and exit before the corresponding `end_iot` ever gets called. As a compromise, we allow resolution requirement to be specified in both routines.

7.1.1.2 Programming A Transaction

Well Structured Code Adaptation The programming of a transaction is straightforward by simply using the two routines to wrap up the code segment whose execution needs to be treated as a transaction. When the source code of the target application is available, the additional transaction code only needs to make up the resolution requirement specification, obtain the necessary environment information and put a pair of `begin_iot` and `end_iot` calls at the appropriate locations, as shown in Figure 7.2. Even if the source code of the target application is not available, it is still possible to put the transaction wrappers around it as shown in Figure 7.3.

```
#include "iot.h"
/* other decls.  */
extern char **environ;
main(char **argv, int argc) {
  iot_spec spec = { ASR, "/coda/misc/bin/resolver" };
  iot_env env;
  char pwd[MAXPATHLEN];
  int tid;
  /* other definitions */
  getwd(pwd);
  env.argv = argv; env.env = environ; env.pwd = pwd;
  env.umask = umask(0); umask(env.umask);
  tid = begin_iot(&spec, &env);
  /* main body */
  (void) end_iot(tid, &spec);
}
/* the rest of the program */
```

This is a template that shows the overall structure of a transaction program. `ASR` is a pre-defined constant used to indicate the corresponding conflict resolution option. The error handling for the IOT interface calls is intentionally left out for clarity.

Figure 7.2: A Template Transaction Program Using Target Application Source Code

Transactional Application Development The development of a transactional application is rather simple. The IOT system provides a standard header file `iot.h` that contains all the necessary declarations and definitions for using the two IOT interface routines. The transaction program only needs to include `iot.h` and link with the provided IOT library file `libiot.a`.

Transaction Scope Limitation In principle, a transaction should be able to cover any segment of a program and obtain the standard IOT properties for the execution of that segment. In the current implementation, it is only suitable to bracket the entire application, i.e., the whole `main` function of the program, as a single transaction. The main reason is that when the transaction needs to be OCC re-executed, the transaction system is only capable of re-executing the entire application instead of the selected segment. Note that this restriction is more the consequence of the transaction system's lack of run-time transaction execution knowledge rather than the design decision of using OCC as the concurrency control algorithm. If the transaction system were fully integrated with the application run-time system, it would be possible to provide the ability to re-execute any segment of a program.

```
#include "iot.h"
extern char *environ;
main(char **argv, int argc) {
  iot_spec spec = { MANUAL, (char *)0 };
  iot_env env;
  char pwd[MAXPATHLEN], cmd[1024];
  int tid, i;
  getwd(pwd);
  env.argv = argv; env.env = environ;
  env.pwd = pwd; env.umask = 0;
  tid = begin_iot(&spec, &env);
  sprintf(cmd, "/usr/misc/bin/latex");
  for (i = 1; i < argc; i++)
    sprintf(cmd + strlen(cmd), " %s", argv[i]);
  system(cmd);
  (void)end_iot(tid, &spec);
}
```

This program can execute `latex` as a transaction using the manual conflict resolution option, as indicated by the pre-defined constant `MANUAL`. The `system` call will create a `sh` sub-process to execute the assembled `latex` command. The error handling for the IOT interface calls is left out for clarity.

Figure 7.3: A Transaction Program not Using Target Application Source Code

7.1.2 Interface for Programming Application-Specific Resolvers

We provide a set of library routines to assist the programming of application-specific resolvers for target applications. They are listed in Figure 7.4 and their usage and functionality is as follows:

```
int in_read_set(char *);
int in_write_set(char *);
int set_global_view();
int set_local_view();
int set_mixed_view();
int get_local_replica(char *, char *);
int get_global_replica(char *, char *);
int preserve_local_result();
```

Figure 7.4: Library Routines for Programming Resolvers

- The `in_read_set()` and `in_write_set()` routines allow the resolver to determine whether a particular Coda object has been read or written by the target transaction respectively. The argument is the pathname of the object to be tested. Any positive return value means that the test is positive, i.e., the object has been read or written by the target transaction respectively. On the other hand, a zero return value means that the test is negative. A negative return value corresponds to an IOT-defined error code.

- The `set_global_view()`, `set_local_view()` and `set_mixed_view()` routines enable the resolver to dynamically adjust views for the relevant inconsistent objects according to its need. A zero return value means that the call has succeeded and negative return values are IOT-defined error codes.

- The `get_local_replica()` and `get_global_replica()` routines can translate the pathname of a regular Coda object into the pathnames of its local or global replicas respectively. The first argument is the input pathname and the second argument contains the result pathname of the corresponding replica. The return value has the same meaning as the previous three routines.

- The `preserve_local_result()` routine implements the preserve operator discussed in chapter 5 which allows the resolver to reuse the local result of the target transaction. The return value has the same meaning as the previous routines.

7.1.3 Other Issues

Communicating Between Application and IOT-Venus Since the IOT interface routines are linked and executed as part of the transaction application, they must be able to communicate with the IOT-Venus. There are several alternatives for supporting such communication, such as using a standard RPC package. We decided to use the I/O control call `ioctl` [1] on the pseudo device where the Coda file system is mounted. The information exchange between the interface routines and the transaction system is encoded in the `ioctl` buffer and passed through Coda's Mini-Cache [67] in the kernel. The main benefit of this approach is implementation convenience. Another benefit is that there is less object code needs to be linked in the transaction executable.

Handling Automatic OCC Re-execution We decided to implement automatic OCC Re-execution through the cooperation between the `end_iot` routine and the transaction system so that standard I/O operations during re-execution can be performed in the transaction's original execution environment. When a transaction `T` fails OCC validation, the transaction system will put the internal representation of `T` into a special queue and notify `end_iot` that `T` needs to be OCC re-executed. The `end_iot` routine will then restore `T`'s original execution environment obtained from the transaction system and employ the `exec` call to re-execute the transaction program. When the ensuing `begin_iot` call issued by the re-execution is received by the transaction system, it will be able to locate `T` from the special queue and manipulate its internal representation so that `T` appears as if it has just started its execution.

7.2 Interactive Interface

Although the IOT programming interface is simple to use, it still requires the target applications to be adapted and re-compiled before they can be executed as transactions. To enhance Unix compatibility, we decided to develop an interactive IOT interface so that existing Unix applications can be executed as transactions without change. Since Unix users typically interact with the operating system through a shell command interpreter, we extended the CMU C-Shell so that transactions can be specified and executed. We use *IOT-Shell* to refer to the extended C-Shell that contains a set of new built-in commands for interactive transaction specification, execution and monitoring.

7.2.1 Interactive Transaction Manipulation Using the IOT-Shell

Transaction Specification Figure 7.5 shows the two new commands that support interactive transaction specification. The `setiot` command takes one mandatory argument which is the pathname of the executable file of the application to be specified as a transaction. The other two arguments are optional. The `option` argument uses one of the four strings {"manual", "reexec", "abort", "asr"} to indicate the corresponding conflict resolution option. If the "asr" option is selected, the third argument must be supplied with the pathname of the resolver executable file. The effect of this command is to treat any subsequent invocation of the specified application from this shell as a transaction using the conflict resolution requirement provided by the command arguments. The `unsetiot` command allows users to eliminate a transaction specification when the corresponding application no longer needs to be treated as a transaction. Figure 7.5 shows transaction specification examples for some commonly used Unix applications. The similarity between `setiot/unsetiot` and the commonly used `setenv/unsetenv` commands enables the new commands to be used in traditional Unix styles. For example, the users can use a profile to automatically set transaction specifications at login time just like setting environment variables.

Command Syntax

setiot <pathname> [<option>] [<asr>]
unsetiot <pathname>

Examples

setiot /usr/misc/bin/latex reexec
setiot /usr/cs/bin/make asr /coda/usr/luqi/bin/make-asr
setiot /usr/cs/compress abort
setiot /usr/cs/bin/emacs manual

Figure 7.5: Transaction Specification Commands and Examples

Transaction Invocation The invocation of a transaction is the same as any normal application. When transactions are specified using the `setiot` command, the IOT-Shell maintains an internal table containing all the current transaction specifications. When a new command is issued, the IOT-Shell will search the transaction specification table to check whether the pathname of the command executable file is specified in the table. If so, the IOT-Shell will automatically insert the appropriate `begin_iot` and `end_iot` calls so that the entire command execution is enclosed within the scope of a single transaction.

On-line Transactions There are situations where users want to enclose a sequence of activities into the scope of a single isolation-only transaction. For example, writing a report on a disconnected client machine often involves browsing, editing and typesetting a collection of related files. It is desirable to treat these actions as a unit and be notified about server updates to any of the involved files. To support this goal, the IOT-Shell provides a pair of commands `begin_iot` and `end_iot` so that the users can bracket a sequence of shell commands into an *on-line transaction*. The standard IOT properties are guaranteed for on-line transactions except that their only conflict resolution option is manual repair.

Transaction Monitoring We provide an additional command `lt`(short form for *list transactions*) for displaying transaction information. When used without argument, this command will print out the identifier, the state and the application name for all the live transactions as well as the recently terminated ones. It can also take a transaction identifier as the argument and display detailed information about the corresponding transaction including resolution specification, readset, writeset, and other statistics.

7.2.2 Internal Mechanisms for Interactive Transaction Execution

Inserting Transaction Wrappers The key to supporting interactive transaction execution is the automatic insertion of transaction wrappers, i.e., the `begin_iot` and `end_iot` calls, by the IOT-Shell as shown in Figure 7.6. For every new command received by the IOT-Shell, it spawns a child process which will lookup the pathname of the command's executable file in the transaction specification table. If the command has been specified as a transaction, the child process will automatically issue a `begin_iot` call using the resolution specification from the table and spawn a grandchild process to execute the received command as a transaction. The corresponding `end_iot` call is made as soon as transaction execution is completed. Note that a modified `end_iot` routine is used so that it will pass the OCC re-execution request from the transaction system back to the IOT-Shell, which in turn will repeat the previous two steps to perform automatic OCC re-execution.

Executing On-line Transactions Because an on-line transaction consists of a sequence of commands, its execution needs special support to ensure that all the relevant file access operations are properly included in the scope of the transaction. Because the IOT-Shell spawns a child process for every new command it receives and assigns a new process group identifier to it, the file access operations performed by an on-line transaction will be associated with different process group identifiers. This creates difficulty for the transaction system to correctly recognize the file access operations belonging to the on-line transaction. Our solution is to let the IOT-Shell inform the transaction system of the new process group identifier every time a new command is to be executed within the scope of an on-line transaction. The transaction system can then dynamically update the process group identifier associated with the on-line transaction to recognize the corresponding file access operations.

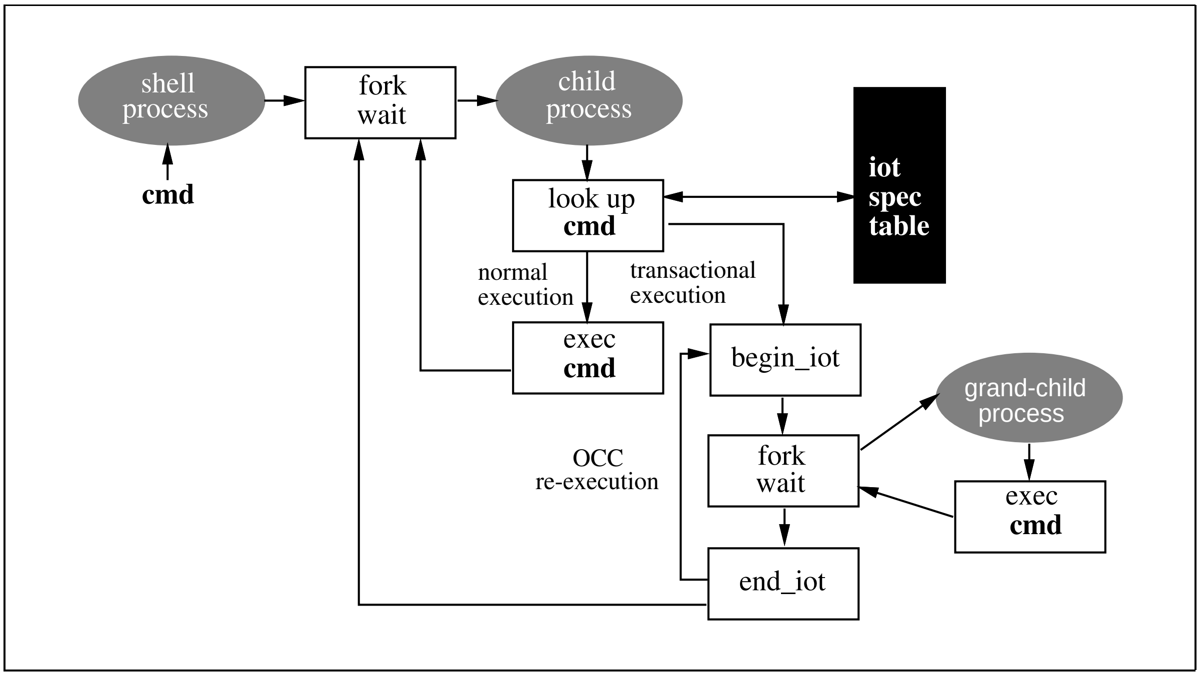


Figure 7.6: Interactive Transaction Execution in the IOT-Shell

7.2.3 Controlling and Monitoring Facilities

We extended Coda's `cfs` [60] utility program to provide facilities for controlling the behavior of the transaction system and displaying transaction information. The "`cfs shd <pct>`" command takes an integer argument (between 0 and 100) and sets it as the percentage limit on how much cache space can be allocated for storing shadow cache files. The "`cfs asrd <path>`" command takes the pathname of a directory as the argument and notifies the transaction system that the directory contains trusted resolver programs. The "`cfs lt [tid]`" command prints out transaction information in the same way as the IOT-Shell's `lt` command. In addition, the transaction system uses a dedicated local file `/usr/coda/iot/log` to record the messages about important events such as the start of a new transaction, the commitment of a pending transaction and the resolution of an invalidated transaction. It also includes the standard output messages from any automatic resolver execution. There is a tool to open a *transaction monitor window* that continuously displays the growing content of the message file.

7.2.4 A Practical Example

Figure 7.7 is the screen image of a Coda laptop showing an actual example of using the interactive IOT interface. The work window displays the process of disconnecting the client, specifying `make` and `latex` as transactions, executing a `make` transaction and a `latex`

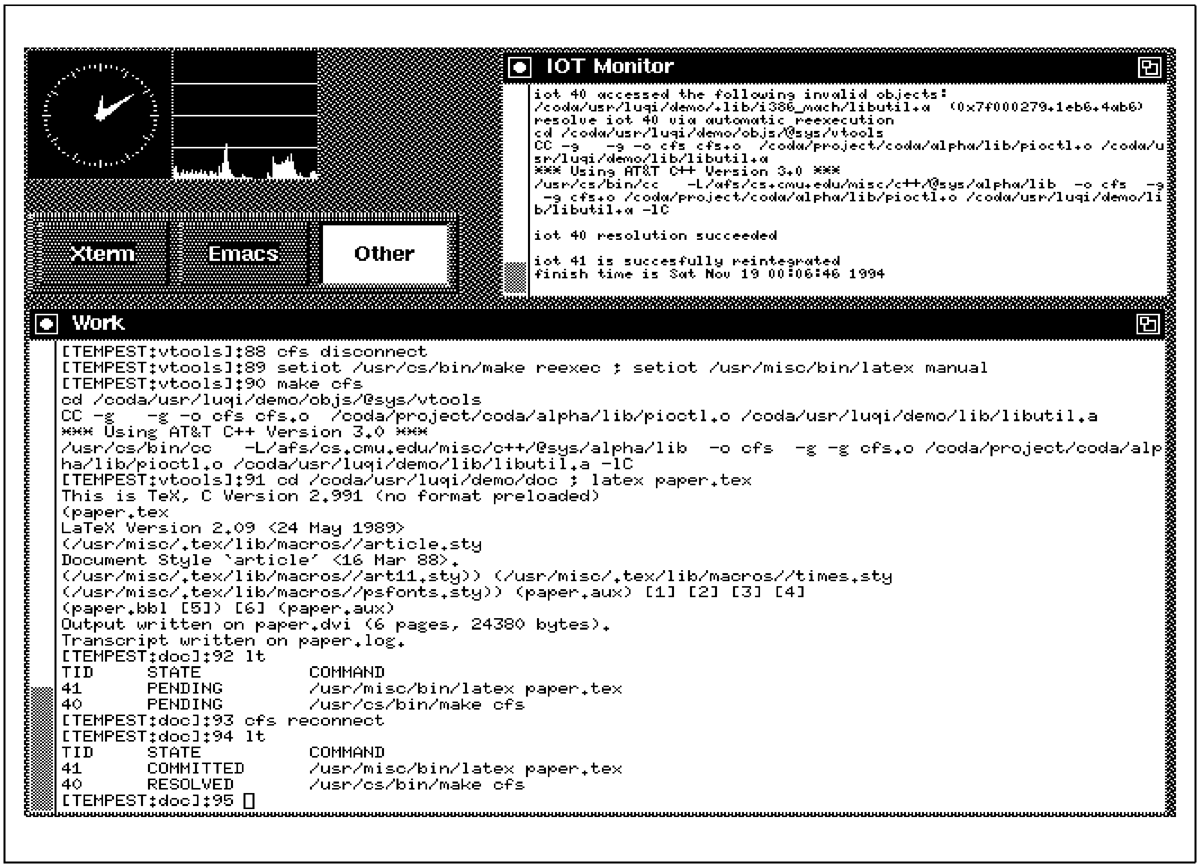


Figure 7.7: A Practical Transaction Example

transaction, reconnecting the client, and checking the transaction status using the `lt` command. At reconnection time, the `make` transaction is invalidated because it has linked a library `libutil.a` which was updated on the server during the disconnection. The IOT monitor window shows the automatic re-execution of the `make` transaction and the commitment of the `latex` transaction.

Chapter 8

Implementation Issues

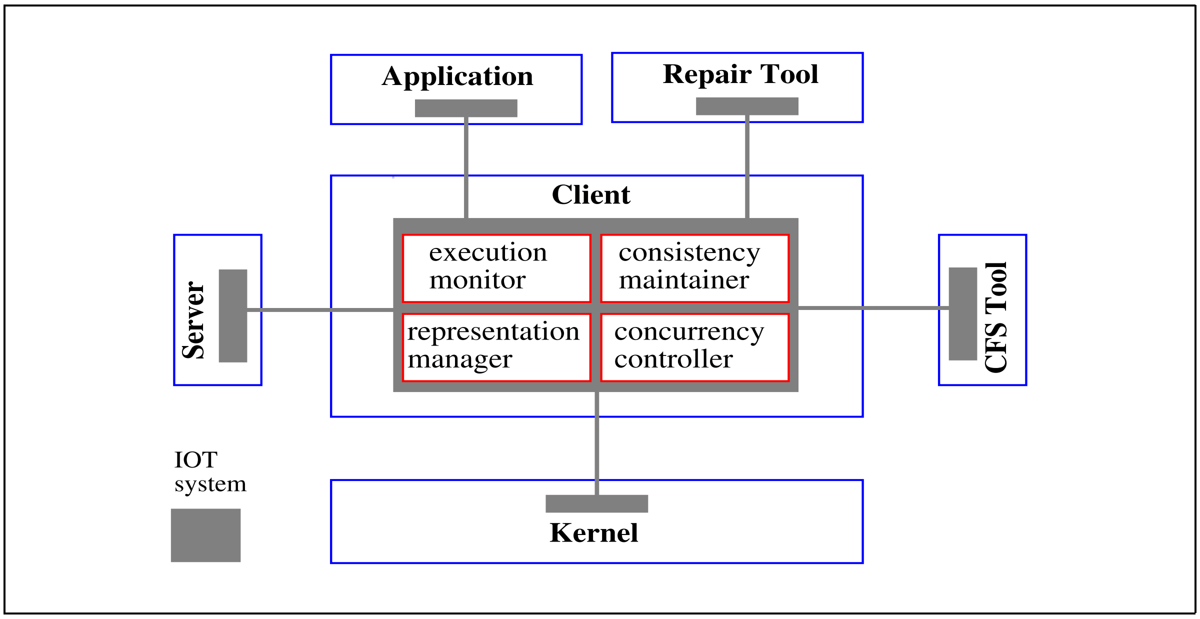
The last four chapters have presented detailed designs on how to enforce the IOT consistency model for transactions executed under various circumstances and how IOT can be used by users and application programmers. To complete the picture, this chapter addresses the remaining important implementation issues.

We begin by describing the overall architecture and main components of the transaction system. Because many of the basic mechanisms have already received in-depth coverage in previous chapters, we focus on issues that have not yet been addressed. These include internal transaction representation, space management for shadow cache files and tuning of the implementation for reduced performance overhead and resource cost.

8.1 Overall Architecture

As shown in Figure 8.1, the entire transaction system is predominately implemented inside the user-level Venus cache manager. There are two main reasons for such an architectural organization. First, because of the strong distinction between the roles of a client and a server in both the underlying Coda architecture and the IOT consistency model, conducting much of the transaction operations on the client side better preserves Coda's security model and overall system scalability. Second, building transaction mechanisms at user level is more convenient for system development and maintenance.

The transaction system inside the IOT-Venus consists of four components. The *execution monitor* is in charge of recording transaction execution information such as readset/writeset and maintaining important data structures such as the transaction serialization graph. The *concurrency controller* performs two levels of concurrency control: OCC across clients and 2PL within a client. The *representation manager*'s main responsibility is maintaining conflict representation, providing accesses to local and global replicas, and managing shadow cache files. The *consistency maintainer* is responsible for the central tasks of validating, committing and resolving transactions.



This picture presents the overall architecture of the IOT extension to Coda. Each rectangular box represents a major component of the underlying Coda system. The shaded areas represent system components that are modified or created to support transaction operations. Note that the shaded area in `Application` refers to the IOT interface.

Figure 8.1: The IOT System Architecture

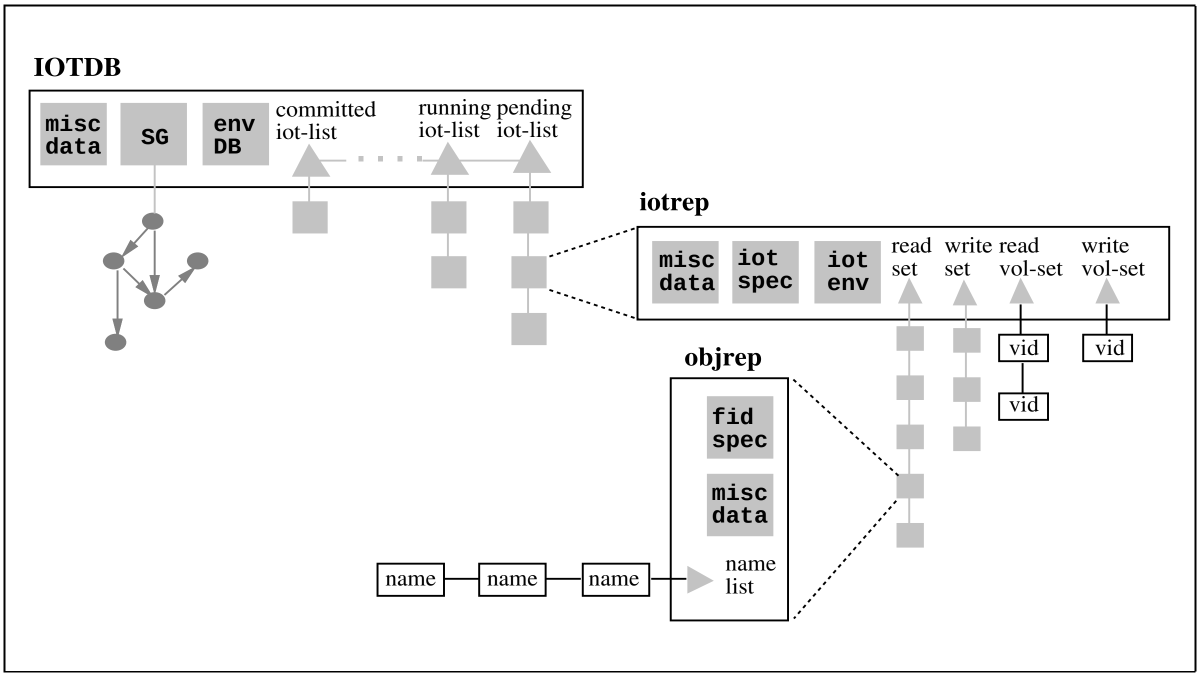
Minor modifications are made to other parts of the Coda system. The interface between the kernel and Venus is extended so that process information can be obtained and employed to identify the file access operations belonging to ongoing transactions. There are some slight adjustments to the server to simplify the process of reintegrating a subsequence of the CML of a volume. In addition, a stand-alone transaction repair tool is provided so that invalidated transactions can be manually repaired. Finally, the `cfs` utility tool is extended with new commands for controlling the transaction system behavior and displaying transaction information.

8.2 Maintaining Internal Transaction Representation

The design of internal transaction representation is very important to overall system performance and resource consumption. This section describes the main data structures shown in Figure 8.2 and how transaction readset and writeset are recorded.

8.2.1 Main Data Structures

A Client-Wide Transaction Database To centralize data management, all the important transaction information is stored in a client-wide data structure called `IOTDB`, which contains the following items.



This figure shows the main data structures used by the internal transaction representation. Each rectangular box corresponds to a major data item and the shaded areas represent data structures that are further explained in the figure or the subsequent discussion.

Figure 8.2: Main Data Structures in Internal Transaction Representation

- **Transaction Lists**

  The most important `IOTDB` component is a group of transaction lists, each containing all the transactions in a particular state. For example, all the running transactions are in the `running-iot-list` and all the pending transactions are in the `pending-iot-list`. Each element of the list is a pointer to `iotrep`, the internal representation of an IOT. The main purpose of using multiple transaction lists is to reduce the performance overhead resulting from frequent internal search activities. Note that terminated transactions are temporarily maintained in their lists and garbage collected by a periodic daemon.

- **Serialization Graph**

  The transaction serialization graph (`SG`) maintains the local dependency among all live transactions with each node representing an IOT or an IFT. `SG` nodes and edges are inserted and removed as transaction activities proceed. Internally, `SG` is represented by a group of doubly linked lists.

- **Environment Database**

  An environment database (`envDB`) is created to allow different transactions executed by the same user to share common environment variables.

- **Miscellaneous Data**

  There are miscellaneous data items mainly used by the basic internal transaction operations. An example is the wait-for graph for detecting transaction deadlock.

Transaction Representation The internal representation of an isolation-only transaction, referred to as `iotrep` in Figure 8.2, contains the following key elements.

- **Transaction Specification**

  This group of information contains both the identity and the conflict resolution requirement of the transaction. It includes the transaction identifier, the process-id and process group-id of the Unix process that invoked the transaction, the transaction's selection of resolution option and the pathname of the resolver executable file if the selected option is ASR.

- **Environment Information**

  `iotrep` stores the environment information needed for possible automatic resolution. As described in Chapter 7, such information contains the pathname of the transaction executable file, the command line arguments, the environment variable list, the `umask` of the master process, and the pathname of the working directory.

- **Readset and Writeset**

  Readset and writeset are the most important components of `iotrep`. They are represented by a doubly linked list with each element containing a pointer to `objrep`, the internal representation of an object accessed by the transaction. `objrep` records the information about all the access operations this transaction has performed on the object.

- **Volume Lists**

  Because of the need to frequently check transaction connectivity, a list of volumes read by a transaction is included in its `iotrep` with each element containing the internal identifier of a volume. Similarly, the `iotrep` also maintains a list of volumes that are updated by the transaction.

- **Miscellaneous Data**

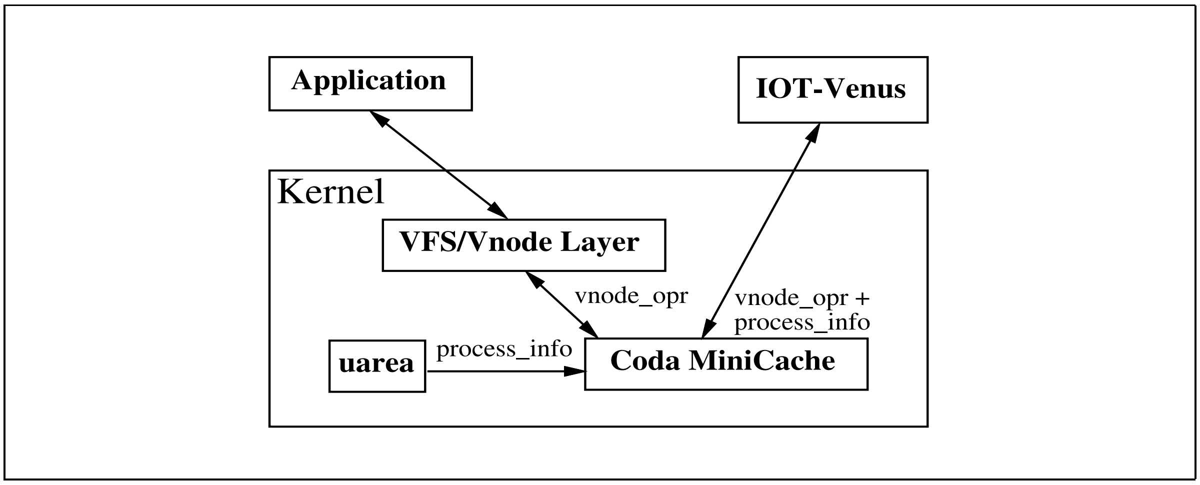
  There are miscellaneous data items in `iotrep` for recording information such as the current transaction connectivity, transaction execution time, etc.

Representation of a Transactionally Accessed Object The most important information recorded in `objrep` is the `fid` of the object. The data item named `spec` in Figure 8.2 is a bitmap recording the sub-parts of the object that are actually read or written by the transaction. For a directory object, its `objrep` contains a list of the names that are accessed in the directory. Miscellaneous data items include a pointer to the shadow cache file of the object if one exists.

8.2.2 Recording Transaction Readset/Writeset

The most frequent internal bookkeeping activity during transaction execution is recording readset and writeset. For every file access operation, the transaction system must detect whether it is performed on behalf of an ongoing transaction. Because such detection needs the process group-id associated with each file access operation, the communication interface between the kernel and the IOT-Venus is extended to pass such process information.

Extending the Kernel/Venus Interface Client support in Coda is divided between a small in-kernel *Mini-Cache* [67] and a much larger user-level Venus cache manager. The main purpose of Mini-Cache is to reduce the frequency of kernel/Venus communication by caching a small amount of information (such as the result of successful `lookup` calls) in the kernel. The Mini-Cache intercepts file system calls on Coda objects from the kernel Vnode layer [28, 57] and redirects them to the user-level Venus by exchanging messages through the Coda pseudo-device. The information about each operation passed from Mini-Cache to Venus is defined in the *Vnode interface* including the operation code, the internal identifier of the operands and the `ucred` data about the user who issued the operation [28, 57]. Unfortunately, such information does not include the needed process information associated with the operation. To address this problem, we extended the Mini-Cache/Venus communication interface to pass this information, as shown in Figure 8.3.



This figure illustrates the kernel extension needed for the IOT-Venus to obtain the necessary process information for every file access operation on Coda objects. The Mini-Cache packs the process information obtained from the kernel `uarea` into messages sent to the IOT-Venus.

Figure 8.3: Extending Kernel/Venus Communication with Process Information

Recording Readset/Writeset Recording transaction readset and writeset involves searching and updating the relevant data structures. Upon receiving a new file access operation `opr` from the kernel, the first step the transaction system undertakes is checking whether `opr` belongs to an ongoing transaction. This is accomplished by linearly scanning all the transactions in the `running-iot-list` using the attached process information. If `opr` is found to belong to a currently running transaction `T`, we must search `T`'s readset or writeset depending on whether `opr` is a read or update operation.

Suppose that `opr` is a read operation and has only one operand `obj`. The transaction system will linearly search through the linked list representing `T`'s readset. If `obj` is not in the list, a new `objrep` is created and inserted into the list, storing information about `obj` and the sub-parts of `obj` accessed by `opr`. If `obj` is already in the readset, the `spec` bitmap in the `objrep` of `obj` is updated to include the sub-parts of `obj` accessed by `opr`. If `obj` is a directory, a new name may need to be inserted into the name-list of the `objrep` depending on the actions performed by `opr`. Update operations involving multiple operands can be processed in a similar manner. Note that the performance overhead caused by the linear search activities can be reduced by using more advanced data structures such as a hash table, particularly for large transactions accessing hundreds of objects.

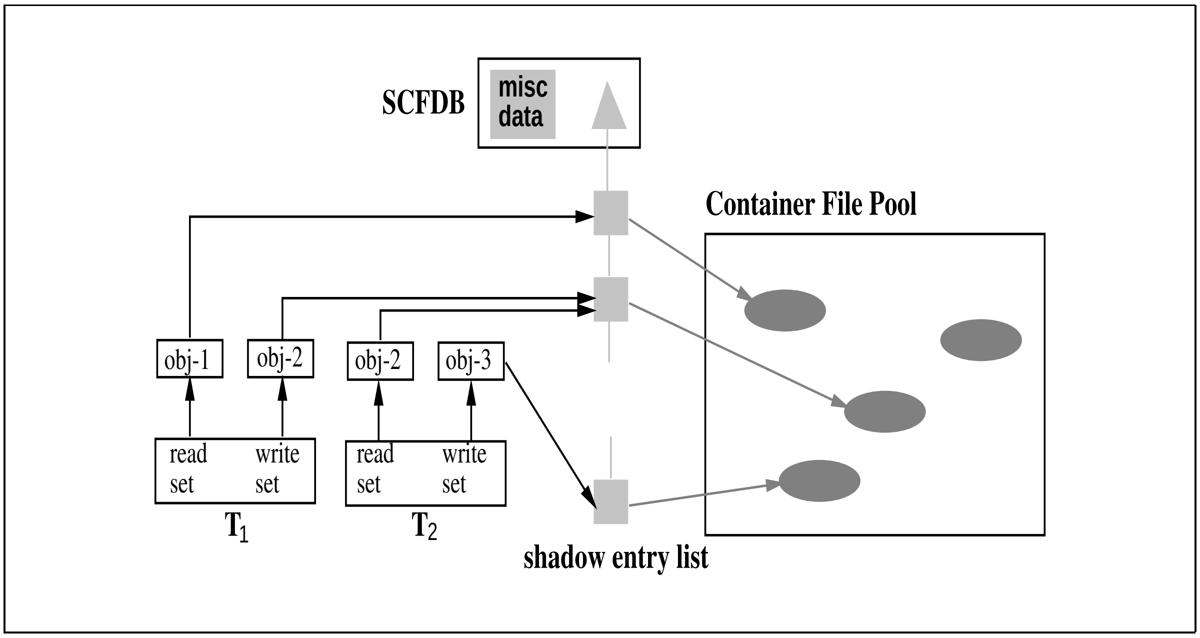
Detecting Abnormal Termination The ability to accurately identify the scope of a running transaction influences the amount of search activity needed for recording transaction readset and writeset. If a transaction `T` forgot to issue the `end_iot` call or its program exits before the `end_iot` call can be made, `T` will remain in the running state and cause unnecessary internal search activities. Thus, we need a reliable mechanism to detect such abnormal transaction termination. Intuitively, solving this problem requires either the kernel to notify the IOT-Venus every time a process exits or the IOT-Venus to poll the kernel about whether the master process of a running transaction has exited. Both approaches are costly in performance and increase complexity to the kernel/Venus communication interface.

We use a much simpler solution based on the observation that whenever a process exits, the kernel always closes all its open descriptors. We designate a special Coda object `/coda` and internally open it for read on behalf of any transaction at the beginning of its `begin_iot` call. The transaction system maintains a counter in `iotrep` which is incremented whenever `/coda` is opened by the transaction and decremented whenever it is closed by the transaction. If the counter reaches zero while the transaction is still in the running state, this means that the transaction's master process has exited without calling `end_iot` and the final decrement causing the counter to reach zero resulted from the kernel closing the open `/coda`. Note that this approach assumes that transactions do not close `/coda` without opening it first.

8.3 Shadow Cache File Management

8.3.1 Shadow Cache File Organization

As discussed in Chapter 4, the transaction system maintains two entities for each shadow cache file, a disk container file holding the shadow content, and a shadow entry containing a pointer to the contained file and a counter recording the number of live transactions that accessed the shadow content. An example of the internal organization of shadow cache files is shown in Figure 8.4. There is a central database (`SCFDB`) containing key information about shadow cache files and their management. The main components of `SCFDB` are a list of shadow entries and some data items used for managing shadow space allocation. The relation between a transaction and its shadow cache files is maintained by the shadow entry pointer stored in the relevant `objrep` belonging to the transaction.

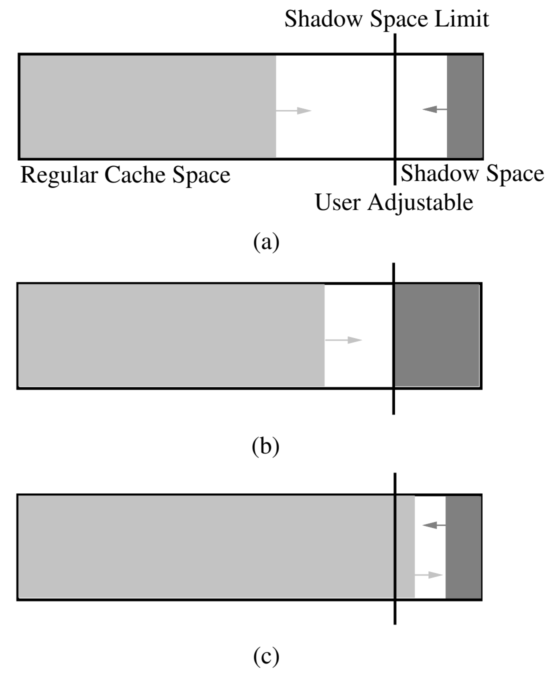


This figure presents an example to illustrate the internal organization of shadow cache files. `SCFDB` is a central data structure that contains a list of all the shadow entries, each of them pointing to a container file in the container file pool. The same shadow entry can be shared by multiple transactions. There are three highlighted shadow entries shared by two live transactions T_1 and T_2. The first entry corresponds to the shadow cache file for `obj-1` that is read by T_1, while the second entry corresponds to the shadow cache file for `obj-2` that is shared by the writeset of T_1 and the readset of T_2. The last entry corresponds to the shadow cache file for `obj-3` that is written by T_2.

Figure 8.4: An Example of Internal Organization of Shadow Cache Files

8.3.2 Prioritized Cache Space Management

The main challenge in maintaining shadow cache files is space management. Our overall principle is *best-effort allocation*, allocating shadow space as long as the physical capacity and the user-specified policy permit. Specifically, we adopted a strategy that manages regular cache space and shadow cache space under the same pool by assigning lower priority to shadow cache files. In addition, we introduce a user adjustable hard limit on how much shadow space can be allocated, with the default set at 20% of the total cache capacity. The main advantage of this combined and prioritized space allocation scheme is that it maximizes cache space utilization while giving preferential treatment for normal cache files. When allocating space for a regular cache file, the total available space is the capacity minus all the occupied space. When allocating space for a shadow cache file however, the total available space is restricted by the hard limit, as illustrated in Figure 8.5.



The picture in (a) illustrates that regular and shadow cache space allocation is similar to the stack and heap space allocation scheme where each side starts at the end and grows toward the middle. However, there is a hard limit on how much shadow space can grow while there is no limit on regular cache space until the capacity is exhausted. Pictures in (b) and (c) show the situations where shadow space exhausts its limit and regular cache space allocation reduces the amount of space available for shadow cache files respectively.

Figure 8.5: Prioritized Cache Space Allocation

8.3.3 Reclaiming Shadow Space

When the cache capacity is exhausted, a variety of techniques can be employed to reclaim shadow space while minimizing the negative effect on potential conflict resolution. The current IOT implementation only supports the automatic compression mechanism.

Manual Deletion One approach is to provide information about shadow space usage and to allow users to select files to be deleted. However, this approach exposes the details of resource management and puts extra burden on the users.

Incorporating User Heuristics We allow the users to supply heuristics to the transaction system so that the shadow cache files for those objects that are deemed unimportant to conflict resolution can be automatically reclaimed. For example, a shadow cache file recording the content of a large object file created by a `make` transaction can be safely discarded because it can be easily regenerated.

Compression Compression is a commonly used technique for reducing file size without information loss. In the context of reclaiming shadow space, it has an extra advantage because most shadow cache files are discarded without ever being used, eliminating the need for decompressing.

8.4 Implementation Optimizations

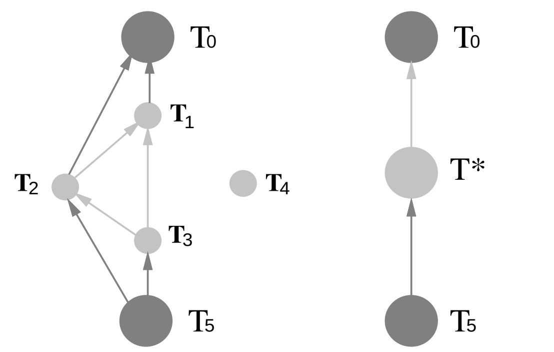
Many aspects of the transaction system have been optimized to minimize the performance degradation and resource consumption resulting from transaction operations. This section describes the most important cases of these optimizations.

8.4.1 Lazy Serialization Graph Maintenance

One performance optimization is lazy `SG` maintenance. Our initial implementation followed a straightforward strategy: a new `SG` node was added as soon as a transaction was invoked and new edges were inserted as soon as dependencies among transactions are detected. However, this approach results in significant performance overhead during disconnected sessions where isolation-only transactions are rarely used. The reason is that each disconnected mutation operation is treated as a separate inferred transaction and requires adding a new node and related edges to `SG`. Because `SG` is maintained in persistent storage, each update entails an asynchronous disk I/O operation, which substantially increases the performance overhead.

To alleviate this problem, we switched to a lazy maintenance scheme where `SG` updates are delayed until necessary. There are two situations where `SG` needs to be updated immediately. The first situation is when an isolation-only transaction completes its execution and enters the pending state. The transaction system must establish all the edges representing the dependency between the newly completed transaction and other live transactions. The second occasion is when a partition is healed and disconnected transactions must get ready for validation and commitment/resolution. The propagator thread requires the correct dependency among all the live transactions during the incremental transaction propagation process. This technique substantially reduces the performance overhead for disconnected operation, particularly when isolation-only transactions are seldom used. When isolation-only transactions are not used at all during a disconnected operation session, a further optimization is adopted to skip `SG` building all together and avoid penalizing the users for the service that they do not use.

8.4.2 Coalescing the Serialization Graph



(a) Before Coalescing (b) After Coalescing

The picture in (a) shows an `SG` containing two nodes(large and dark) corresponding to IOTs and four nodes (small and light) corresponding to consecutively executed IFTs. As described in Chapter 4, an edge from $\mathtt{T}_i$ to $\mathtt{T}_j$ means that transaction $\mathtt{T}_i$ must be serialized before transaction $\mathtt{T}_j$. The picture in (b) shows the result of coalescing with the large light node representing the compound node.

Figure 8.6: An Example of Coalescing a Serialization Graph

Another SG maintenance optimization is to reduce the space cost by coalescing the nodes corresponding to a consecutive sequence of inferred transactions. For a long-lasting disconnected operation session containing few isolation-only transactions, there could be a large number of SG nodes and edges corresponding to inferred transactions. We have experienced SGs with hundreds of nodes and over two thousand edges. A huge SG costs a significant amount of persistent storage space, a scarce resource on a mobile client. To mitigate this problem, we have modified the SG maintenance mechanism so that the nodes corresponding to a long sequence of consecutively executed inferred transactions will be automatically coalesced into a compound node whenever transaction propagation is triggered. Edges among the coalesced nodes are eliminated while those in and out of the group are preserved and connected to the compound node, as shown in Figure 8.6.

This method can drastically reduce the size of SG and its persistent space cost. The price paid for this optimization is that all the inferred transactions corresponding to the coalesced nodes must now be treated as a unit, validated and committed together. The increased granularity can lead to unnecessary dependency among transactions. In the example shown in Figure 8.6, if transaction T_0 is invalidated, transaction T_4 must wait until T_0 is resolved to be propagated to the server. Without SG coalescing, the transaction system will know that T_4 does not depend on T_0 and can be propagated without waiting. However, thanks to the low likelihood of conflict in practice, this strategy is acceptable and consistent with the batch reintegration adopted in the vanilla Venus.

8.4.3 Sharing Environment Variables

Another optimization that can significantly reduce persistent space cost is sharing environment variables among live transactions. As will be shown in the next chapter, the persistent space used to store environment variables is about 2KB per transaction in our environment. The key observation that leads to this optimization is that users rarely change their environment variable definitions. Even if there is modification, the changes are limited to only a few variables. Instead of maintaining an entire environment variable list for each individual transaction, the transaction system maintains a global environment variable list in IOTDB for each user who has invoked at least one live transaction. The environment variable information stored in the iotrep of a particular transaction only records the difference between the variables defined in the global list and those specific to the transaction.

8.5 Persistence and Crash Recovery

Because the client can crash for various reasons such as fatal runtime errors and machine shutdown, maintaining critical information in persistent storage is crucial for the transaction system to resume normal operations after system restart. This section focuses on recording a persistent image of the transaction system and recovering from crashes.

8.5.1 Persistent Data Structures

The RVM Package Like the rest of the Coda system, the IOT implementation uses *RVM* [63], a lightweight transaction facility, for maintaining persistent data structures. RVM exports the abstraction of *recoverable virtual memory* to its host application which can map *regions* of recoverable segments onto portions of its virtual address space. Accesses to mapped data are performed using normal memory read and write operations. However, if such accesses are bracketed with RVM's begin and end-transaction statements, failure atomicity is automatically provided. RVM asynchronously flushes updates to recoverable memory to the backing disk and allows the application to control the frequency of such flushes. The transaction system inherits existing Coda policies of scheduling asynchronous RVM flushes during disconnected operation [26].

Recoverable Image Almost all the important information included in the transaction data structures discussed previously such as transaction readset/writeset, transaction specification and environment information are stored in RVM. Because RVM space is a scarce resource, particularly on a portable client machine, the design of the data structures minimizes the portion that must remain persistent. Data items used to record transient system state, such as the current connectivity of a transaction are not kept in RVM because their values can be re-computed from other data.

8.5.2 Crash Recovery

Upon startup, the transaction system on a client loads necessary information from RVM and restores the internal transaction data structures to a consistent state so that previous transactions are retained and normal transaction services can be resumed.

Recovering Live Transactions Recovering transactions in the pending, to-be-resolved and to-be-repaired state is straightforward and merely consists of reloading the RVM image of the relevant data structures and resetting those data items that can be re-computed from other data. The recovery of transactions in the running, resolving and repairing state needs additional work. The current implementation forces such active transactions into a steady state because their original execution or resolution before the crash cannot be automatically resumed. For a running transaction, the recovery mechanism will transit it into the pending state because its execution result is still in the client cache. This allows the transaction to be propagated to the servers via commitment or resolution. For a resolving transaction, the recovery mechanism will automatically abort the resolution result and transit it into the to-be-repaired state. Thus, a client crash during the resolution is considered the same as a resolver crash. Similarly, a repairing transaction will be recovered as a to-be-repaired transaction again and the original repairing result is automatically discarded.

Recovering Conflict Representation Because updates to the persistent data structures for conflict representation can be originated from different modules of the transaction system and not enclosed within a single RVM transaction, it is possible for a client crash to occur during certain operations such as splitting a DRR subtree and to leave the corresponding data in a bad state. Thus, the key to recovering conflict representation is to gather the relevant persistent data and reorganize them into a previous state that satisfies the requirements of conflict representation for the involved objects. Note that some of the work up to the point of crash could be lost. For example, a local subtree could be left unmounted due to resolution activities when a crash happened. The recovery mechanism must clean up the fake-joint objects of the corresponding DRR subtree so that the local subtree can be automatically re-mounted when accessed again. As another example, all the DRR subtrees that switched to the local or global views before the crash will be automatically reset to the default mixed view.

8.6 Transaction Validation

As discussed in Chapter 4, both OCC validation for connected transactions and GC validation for disconnected transactions are performed by the underlying transaction certification mechanism. This section discusses three important implementation issues about transaction validation that have not been addressed in the previous chapters.

8.6.1 Overloading with Cache Coherence Maintenance

Transaction validation is an expensive operation involving exchanging a variable number of messages between the client and the relevant servers depending on the transaction being validated. However, it can be overloaded on Coda's underlying cache coherence mechanisms.

Coda servers maintain a callback for every cached object on a connected client and send a message to break the callback of an object when its server replica is updated by another client. As soon as the callback of an object `obj` is broken, the transaction system iterates through all the live transactions that have accessed `obj` and immediately invalidate them if they have not already been. If such a transaction is still in the running state, the transaction system will make an internal mark so that it can continue its execution and be automatically invalidated upon completion. On the other hand, new callbacks for cached objects are also established whenever possible after a transaction validation contacted the servers for the corresponding objects.

During disconnected operation, all the cached objects are *demoted* by marking their CCS (cache coherence status) as suspect. When the client is reconnected to the corresponding servers, Venus (via a periodic daemon) will try to verify for each demoted object whether it has a newer version on the server or not, and re-synchronize the local and server versions by establishing the callback relationship and promoting the CCS. When validating a disconnected transaction, the transaction system first checks the CCS for every accessed object. If the CCS is demoted, it will check with the servers in the same way as the Venus daemon would do. As a result, it will also re-establish callback and promote CCS whenever possible.

8.6.2 Object Version Maintenance

Certifying a transaction involves comparing the local version-id and the global version-id of objects accessed. As indicated in Figure 8.2, the transaction data structures do not include version-id's in the readset/writeset. This significantly reduces the space cost because the current Coda implementation uses a version-vector as the version-id. This means that the IOT-Venus cache manager must maintain the following invariant for any transaction `T` when it is being OCC or GC validated: For any object $\texttt{obj} \in (\texttt{R(T)} \cup \texttt{W(T)})$, $\texttt{lvv(obj)} \neq \texttt{gvv(obj)}$ *if and only if* `T` accessed a version of `obj` that is different from its current server replica. Recall that `lvv(obj)` and `gvv(obj)` are the local and global version-vector of `obj` respectively.

The key to maintaining the invariant is to keep `lvv(obj)` unchanged when `obj` is locally updated. Only the successful commitment of a local transaction that updated `obj` will cause both `lvv(obj)` and `gvv(obj)` to change to an identical new version-vector. When `T` is being validated, there are two basic scenarios about `obj`. If `obj` had not been locally updated before being accessed by `T`, `lvv(obj)` represents the server version of `obj` when it was last fetched by the client. If `obj` had been locally updated by previous transactions before being accessed by `T`, those transactions must have already successfully committed their update on `obj` to the server. Either way, `lvv(obj)` represents a version of `obj` that has not only been accessed by `T` but has also appeared on the server. Thus, a different `gvv(obj)` means that `T` has accessed a different version of `obj` than its current server replica.

8.6.3 Validation Atomicity

To guard against race conditions among concurrent transactions across clients, the validation of a transaction and its ensuing commitment need to be performed in an atomic unit. As explained in Chapter 4, such atomicity has not been fully provided in the current implementation. The purpose of this discussion is only to present a relatively simple scheme that can satisfy the atomicity requirement.

Integration with 2PC The standard technique for achieving atomicity among a set of distributed activities is the two phase commitment protocol (2PC). Previous research proposed a strategy that can integrate transaction validation and commitment into a 2PC framework so that they can be performed atomically [70, 55].

The validation and the possible commitment of a transaction `T` starts with the client selecting one of the involved servers as a coordinator and sending it information about ($R(T) \cup W(T)$) and `TML(T)`. In the first phase, the coordinator sends a `PREPARE` message to all the involved servers and their share of the objects to be validated as well as the mutations to be committed (if any). Upon receiving the message, a participating server will perform its local validation. If such validation fails, a `FAIL` message is sent back to the coordinator. Otherwise, it will replay the necessary mutations in shadow space [38, 5], log a `PRE-COMMIT` record and send an `OK` message back to the coordinator.

The second phase begins when the coordinator receives all the responses from the participating servers. If all the local validations are successful, a `COMMIT` record is logged and the `COMMIT` message is sent to all those servers. When a remote server receives a `COMMIT` message, it simply commits the shadowed replay result (if any). If any of the local validations fails, the coordinator will send an `ABORT` message to those servers that voted `OK` so that their replay result can be discarded. Finally, the coordinator will notify the originating client about the outcome of the validation and commitment.

Global Mutual Exclusion In addition to atomicity, the global validation of transactions must ensure that the corresponding local validations are processed in the same order at all the involved servers. Any server participating in the validation of one transaction cannot be involved in the same task for another. Mutual exclusion among global transaction validations is necessary to ensure the correctness of the validation outcome. A simple scheme to achieve this can be the following. Each server dedicates a shared resource such as a `mutex` and requires that any server thread trying to participate in the validation of a transaction must acquire exclusive control of the resource before it can start the process. Thus, whenever a server receives a request from a coordinator or the originating client, it will first try to grab the resource and will have to wait if the server is already engaged in the validation of another transaction. A problem of this approach is that servers may deadlock. However, this can be addressed by selecting an appropriate interval to timeout and re-try.

Chapter 9

Evaluation

A working implementation of an IOT extension to the Coda file system has been operational for nearly a year. This realization of the IOT model in a practical distributed file system establishes a solid foundation for us to demonstrate the viability of this thesis. In this chapter, we provide data from the implementation to quantify the cost of supporting IOT. We also provide some qualitative usage data to augment the quantitative data.

9.1 Overview

9.1.1 System Evolution and Status

An early version of an IOT extension to the Coda file system with basic transaction functionalities and the automatic re-execution resolution capability became operational in the summer of 1994. It served as a basic prototype for experimenting and fine tuning various design and implementation alternatives. The IOT programming interface as well as the C-Shell interactive interface were developed shortly afterwards. However, further development was briefly stalled while design and implementation complications pertaining to conflict representation and resolution were being resolved. In late 1994, a major system overhaul and re-implementation were conducted for the integration of the conflict representation and resolution components. A new version supporting most of the IOT model has been operational and stable since early 1995. Since then, most of the development activities have been of the nature of bug fixing and minor enhancement. The current IOT implementation supports most functionality of the IOT model. The two major exceptions are: (a) only the global certification consistency guarantee is provided; (b) the commitment of distributed transactions is not fully atomic.

As of this writing, the IOT implementation is maintained as a separate branch of the mainline Coda system with the portions for conflict representation, kernel changes and server changes being incorporated in the production Coda release. The IOT facility is available to the Coda user community for anyone willing to install the client with the IOT extension. The transaction system is supported on two client platforms shown in Table 9.1. Besides being used by the author on a daily basis for over a year, the IOT service has also been used by another Coda user as a base for implementing a mobile CSCW repository system [47].

Client	**Brand**	**CPU**	**Memory**	**Disk**	**OS**
desktop	DEC 5000/200	R3000(25MHz)	32MB	400MB	Mach 2.6
laptop	DECpc 425SL	i486(25MHz)	32MB	200MB	Mach 2.6

This table displays key information about the two kinds of client machines that IOT supports. In the rest of this chapter, we will simply use the term `laptop` and `desktop` to refer to them.

Table 9.1: Client Platforms Supported by IOT

The current IOT system configuration consists of the following components. The binary of a special Venus that embodies most of the transaction system (referred to as the *IOT-Venus*) must be used in place of the regular Venus(referred to in contrast as the *vanilla Venus*). The executable of the special IOT C-shell needs to be installed at a proper location so that it can be used as a login shell or conveniently invoked when needed. In addition, a special repair tool is needed for manually repairing invalidated transactions. Programming transactions or application-specific resolvers requires the installation of a set of IOT libraries and header files.

The IOT source code lives in several Coda modules, all maintained as branches off the mainline Coda source. The *iot* module implements the core IOT functionality and contains over 9000 lines of C++ code. The *venus* module contains IOT related modifications and extensions to the vanilla Venus. It is a shadow of the mainline *venus* module and has 24 files containing about 1000 lines of modified or added C++ code. The source module for the special IOT C-Shell is also a shadow of the CMU Mach C-Shell source containing eight source files. The conflict representation mechanisms implemented by over 5000 lines of C++ code have been merged into the production release of Coda. There are separate modules for the IOT programming interface and the IOT repair tool totaling about 1000 lines of C++ code. Other miscellaneous IOT related system items include the IOT data collection mechanism, minor Coda server changes and Mach kernel extensions, which all have been promoted to production Coda release.

The current IOT implementation in Coda is more complex than anticipated. The primary source comes from the need to operate the client in two different modes for conflict representation and resolution. The design is optimized for simplifying the resolution process and resolver programming. However, it requires using complicated data structures and algorithms to manage replicas and the relevant system resources. The implementation could be much more complicated if we were to provide the G1SR consistency guarantee because it would require the servers to maintain a complete transaction history and a distributed graph during consistency validation. Tasks such as managing server space for recording transaction history and handling failures during consistency validation could add a great deal of complexity to the overall system.

9.1.2 Basic Evaluation Approach

The ultimate purpose of this dissertation is to verify the thesis that an explicit transaction extension to the Unix file system with serialization-based isolation guarantees can substantially improve consistency support for mobile file access using disconnected operation. Recall that the specific goals we set to achieve are: offering improved consistency support for disconnected operations; maintaining upward Unix compatibility; and seeking good performance, low resource cost and practical usability. The previous chapters have shown how the IOT model, by design, meets the first two goals. This chapter focuses on the third goal: showing that the performance and resource cost for supporting transaction operations in Coda are indeed acceptable.

Our evaluation approach is to rely on carefully controlled experiments. Transaction performance and resource cost are evaluated based on quantitative experimental results. Other facilities such as collected file reference traces and previous file system study data are utilized to enhance the realism and quality of the evaluation. Section 9.2.1 and Section 9.2.2 employ both controlled experiments and trace replay to measure IOT incurred performance overhead. Section 9.3.1 relies on controlled experiments to measure global system resource costs such as server CPU and I/O time and network traffic. Section 9.3.2 uses trace simulation and analysis to examine local system resource cost such as client disk and RVM space. Because of its subjective nature, only a preliminary assessment of IOT usability is presented in Section 9.4. Finally, a plan is presented for further usability evaluation in section 9.5.

9.2 Transaction Performance

The design and implementation of IOT in Coda has pursued a lightweight strategy, stripping away as many unnecessary features as possible to minimize performance overhead. The key performance evaluation question we want to answer is: *can the Coda file system with an IOT extension offer satisfactory overall system performance for executing common applications with or without using the transaction service?* Because the performance is predominantly influenced by the underlying Coda operations, our evaluation will focus on comparing performance with and without using transactions to determine IOT-incurred overhead.

We use the term *transactional operation* to stand for the file access operations that are executed within the scope of an explicit isolation-only transaction and *normal operation* for those that are outside the scope of any transaction. Obviously, an application executed as a transaction will have to pay a performance cost. What is not so obvious is that even normal file access operations suffer a small performance penalty when executed in a system that supports IOT. This is because both transactional and normal operations share the same underlying infrastructure and the file system often has to commit internal resources to differentiate and coordinate these two kinds of operations. We measure the IOT incurred performance overhead for normal operations and transactional operations in section 9.2.1 and Section 9.2.2 respectively.

9.2.1 Performance Overhead for Normal Operations

Minimizing the performance overhead for normal file access operations is critical to the success of the IOT model. Because IOT is an optional facility intended to be used only for selective applications, normal operations usually dominate file system activities. Slowing down these operations significantly will render the entire system unattractive to the users.

In the current implementation, almost all of the internal transaction activities occur in the IOT-Venus, which works with the production Coda servers, kernel and client/server communication facilities. This means that our evaluation only needs to compare the performance between the IOT-Venus and the vanilla Venus running the same set of applications. In addition, the experiments focus only on disconnected operation because the current IOT implementation performs the same amount of extra work for normal operations regardless of whether the client is connected or disconnected. The specific question we investigate here is: *what is the IOT incurred performance overhead on normal file access operations for common activities during disconnected operation?*

9.2.1.1 Methodology

The main source of performance overhead comes from the need to distinguish and coordinate the transactional and normal file access operations. For example, we need to decide for every file access operation whether it belongs to a running transaction or not. This involves getting the additional process group information from the kernel and searching linked lists representing active transactions. Also contributing to the overhead are the searching and bookkeeping activities performed by internal tasks such as local concurrency control and shadow cache file maintenance.

We use a variety of experiments to evaluate this performance overhead. Each of them involves executing a certain workload on both the laptop and desktop clients described in Table 9.1 and measuring the total elapsed time. The first experiment uses the Andrew Benchmark, a widely used file system benchmark [24]. The second experiment consists of trace replay of file references traces collected by Lily Mummert from workstations in our environment [46]. Four segments of file reference traces are replayed to emulate the execution of the applications that generated the references originally. From the two main target application domains of software development and document processing, we select two representative tasks from each: compiling the Coda server and client, and typesetting a Ph.D. dissertation and a thesis proposal using `latex`.

	Laptop			Desktop		
	Vanilla Venus(sec)	IOT Venus(sec)	Over-head	Vanilla Venus	IOT Venus	Over-head
MakeDir	1.3 (0.5)	1.3 (0.5)	0.0%	0.9 (0.6)	1.0 (0.5)	11.1%
Copy	12.8 (0.9)	13.3 (0.9)	3.9%	11.3 (0.9)	11.3 (0.9)	0.0%
ScanDir	14.7 (0.7)	15.6 (0.7)	6.1%	14.6 (0.7)	15.6 (0.7)	6.9%
ReadAll	23.6 (0.8)	24.6 (0.8)	4.2%	25.2 (1.0)	25.6 (1.0)	1.6%
Make	85.0 (1.2)	85.4 (1.0)	0.5%	59.2 (2.4)	59.3 (1.9)	0.2%
Total	137.4 (0.7)	140.2 (0.9)	2.0%	111.2 (3.3)	112.8 (2.5)	1.4%

This table shows the elapsed time of executing the Andrew Benchmark as a normal application on disconnected client machines. The time values represent the mean over ten runs of the benchmark. Numbers in the parentheses are standard deviations. In addition, the performance overhead of IOT-Venus versus vanilla Venus is also listed.

Table 9.2: Normal Operation Performance of Andrew Benchmark

9.2.1.2 Results

Andrew Benchmark The Andrew Benchmark performs file access operations in five phases. The `MakeDir` phase creates four directories in the test area; the `Copy` phase copies files from the source test tree; the `ScanDir` phase opens all the directories and examines the status of all the files; the `ReadAll` phase opens and reads all the files; and the `Make` phase compiles an application from those files. The measured elapsed time of all five phases on both laptop and desktop clients is listed in Table 9.2.

The results on both laptop and desktop clients indicate that the maximum performance overhead incurred by the IOT extension is about 7%, ignoring the `MakeDir` phase which is too short. The overall performance overhead is 2% or less. The `MakeDir` phase has a higher overhead than that of the entire benchmark and the probable cause is that the metric of seconds is too coarse relative to its duration. Any noise in the measurement can lead to a significant skew in the result. This can also explain why the standard deviations as a percentage of the execution duration for this phase are higher than that of any other phase.

Trace	**Laptop**			**Desktop**		
	Vanilla Venus(sec)	IOT Venus(sec)	Over-head	Vanilla Venus(sec)	IOT Venus(sec)	Over-head
Concord1	1655.8 (5.3)	1659.8 (16.5)	0.2%	1615.8 (10.1)	1624.6 (20.6)	0.5%
Concord2	1564.8 (7.8)	1572.4 (5.4)	0.5%	1541.0 (11.6)	1546.0 (12.7)	0.3%
Messiaen	1523.4 (8.1)	1537.4 (8.8)	0.9%	1526.8 (5.1)	1529.8 (6.7)	0.2%
Purcell	1564.6 (5.1)	1570.2 (3.9)	0.4%	1602.6 (5.9)	1607.6 (8.4)	0.3%

Table 9.3: Normal Operation Performance of Trace Replay with $\lambda = 1$

This table shows the elapsed time of running trace replay with $\lambda = 1$ on disconnected laptop and desktop clients. The time values represent the mean over five runs. The numbers in parentheses are standard deviations.

Trace Replay To further examine the performance overhead of normal operations, we replay segments of collected file reference traces as the experiment workloads. File access operations recorded in a reference trace are first extracted using the `untrace` tool [46] and then stored in a command file. We then use the `creplay` tool [46] to re-run the file access operations recorded in the command file. `creplay` reads the operations from the command file and generates Unix system calls that are serviced by the Coda file system as if they have been generated by a human user or an application. Realism in the workload is largely preserved because the only difference is that now a single Unix process issues all the replayed system calls whereas the original trace might have been produced by multiple processes. The four trace segments used in the experiments each lasted 30 minutes in their original execution and are carefully screened to ensure that they contain active file references. Their names, *purcell*, *messiaen*, *concord-1* and *concord-2* come from the workstations from which the traces were taken.

An important issue in trace replay is to incorporate the effect of the delay intervals between the file access operations in the original traces. We adopt a parameter called *think threshold* (λ)

Trace	Laptop			Desktop		
	Vanilla Venus(sec)	IOT Venus(sec)	Over-head	Vanilla Venus(sec)	IOT Venus(sec)	Over-head
Concord1	224.2 (9.0)	225.0 (8.3)	0.4%	156.6 (1.5)	157.0 (4.7)	0.3%
Concord2	277.2 (5.0)	278.6 (8.7)	0.5%	173.0 (3.4)	174.2 (4.7)	0.7%
Messiaen	125.0 (1.6)	125.4 (4.7)	0.3%	80.2 (0.8)	80.8 (0.8)	0.8%
Purcell	24.6 (0.6)	24.8 (0.8)	0.8%	15.4 (0.5)	15.6 (0.5)	1.3%

Table 9.4: Normal Operation Performance of Trace Replay with $\lambda = 60$

This table shows the elapsed time of running trace replay with $\lambda = 60$ on disconnected laptop and desktop clients. The time values represent the mean over five runs. The numbers in parentheses are standard deviations.

proposed in [45] to control the delay effect. It means that any delay greater than λ seconds in the original trace will be preserved in the replay experiment. We choose two different λ values of 1 and 60 in our experiments. When $\lambda = 1$, most of the original delays were preserved so that the trace replay proceeds at a speed close to the original pace. When $\lambda = 60$, there are no delays between file references during the replay, giving us an opportunity to observe the performance overhead under very I/O intensive conditions.

The trace replay results are shown in Table 9.3 and Table 9.4. The measured time is the total elapsed time of executing the `creplay` program on a given trace command file. The observed performance overhead for normal operations is less than 1% in almost all cases.

Software Build Task	Laptop			Desktop		
	Vanilla Venus(sec)	IOT Venus(sec)	Over-head	Vanilla Venus(sec)	IOT Venus(sec)	Over-head
Venus	3662.0 (37.0)	3679.2 (50.7)	0.5%	2960.8 (49.7)	2976.0 (70.6)	0.5%
Server	992.0 (8.0)	998.6 (6.3)	0.6%	642.6 (4.6)	645.6 (3.4)	0.5%

This table shows the elapsed time of building a Coda client and a Coda server on disconnected laptop and desktop clients. The time values represent the mean over five runs. The numbers in parentheses are standard deviations.

Table 9.5: Normal Operation Performance of Building Coda Client and Server

Software Build Tasks We measure the performance of two common software build tasks in the Coda project, building a Coda client and a Coda server. The results in Table 9.5 show that the performance overhead for the two common software build tasks is only about half a percent.

Document Build Task	Laptop			Desktop		
	Vanilla Venus(sec)	IOT Venus(sec)	Over-head	Vanilla Venus(sec)	IOT Venus(sec)	Over-head
Thesis	145.4 (0.6)	146.0 (1.9)	0.4%	96.0 (2.2)	96.4 (1.8)	%0.4
Proposal	33.8 (0.5)	34.2 (0.5)	1.2%	24.0 (1.2)	24.2 (0.8)	%0.8

This table shows the elapsed time of typesetting a Ph.D. dissertation and a thesis proposal using `latex`. The time values represent the mean over five runs. The numbers in parentheses are standard deviations.

Table 9.6: Normal Operation Performance of Typesetting a Dissertation and a Proposal

Document Build Tasks Document processing is one of the most common activities in our target environment. We measure the performance of using `latex` to typeset a 272-page Ph.D. dissertation and a 54-page thesis proposal. The results in Table 9.6 confirm that the IOT incurred performance overhead for normal operations in such tasks is small.

9.2.1.3 Discussion

The above set of experiments cover a broad range of workloads in disconnected operation. The observed performance degradation for normal operations caused by the IOT extension is small across all our workloads. The measured results are also in complete agreement with our qualitative perception of performance in actual usage.

There are two kinds of pathological situations where the performance can be worse than what has been measured. First, when a normal file access operation is trying to access an object that is currently locked by an ongoing isolation-only transaction, it will have to block until the two-phase-locking protocol completes. Second, if a normal mutation operation is trying to update an object accessed by a currently pending transaction, additional internal work needs to be done to create a shadow cache object. We do not take these two factors into account in our experiments because they are rare and because their impact can vary widely, depending on the frequency and the extent of interaction between transactional and normal operations. Only empirical data from actual IOT usage can provide meaningful data on these two factors.

Finally, the current implementation has not been fully tuned for performance. More careful tuning could lead to further reduction in the performance overhead.

9.2.2 Performance Overhead for Transactional Operations

Clearly, there are performance costs to be paid for applications to gain the improved consistency support from the IOT service. The question is, how much?. More specifically, we want to know: *what is the performance overhead for executing common applications as transactions on a disconnected client?*

9.2.2.1 Methodology

The performance overhead for transaction execution comes from a variety of sources. First, maintaining transaction readset and writeset requires frequent operations on the linked lists representing them, such as searching, inserting and deleting. Second, every list mutation operation requires an RVM transaction, triggering asynchronous disk writes caused by RVM flushes. Finally, there are other internal bookkeeping operations such as maintaining the serialization graph, pinning transactionally accessed objects in the client cache, and recording the transaction environment and consistency specification at transaction start-up time.

To quantify this overhead, we repeated the set of experiments described in Section 9.2.1, this time using IOTs. Each experiment involves running the workload first encapsulated in a transaction, and then as a normal application. The experiments are conducted on the IOT-Venus on disconnected laptop and desktop clients as specified in Table 9.1. We measure the elapsed time and compare the results from the two Venii to derive the performance overhead. In order to measure the performance of multiple transaction executions, we also compare running each phase of the Andrew Benchmark as a separate transaction versus running the entire benchmark as a single transaction.

9.2.2.2 Results

Andrew Benchmark Table 9.7 show the results of executing the Andrew Benchmark as a single transaction or one transaction per phase. Figure 9.1 presents a graphical representation of the same data combined with the data presented earlier in Table 9.2. The performance overhead for single transaction execution is around 10% on both platforms. However, different phases exhibit different levels of performance degradation. This is because the key factor in determining transaction performance overhead is the intensity of I/O activity of an application.

Higher I/O intensity usually leads to a bigger performance penalty. As more file access operations are performed per unit execution time, the more likely that the transaction readset and writeset need to be updated using RVM transactions. Hence, there are fewer opportunities for asynchronous RVM flushes to overlap with application computation (or user think) time, thus leading to higher performance overhead.

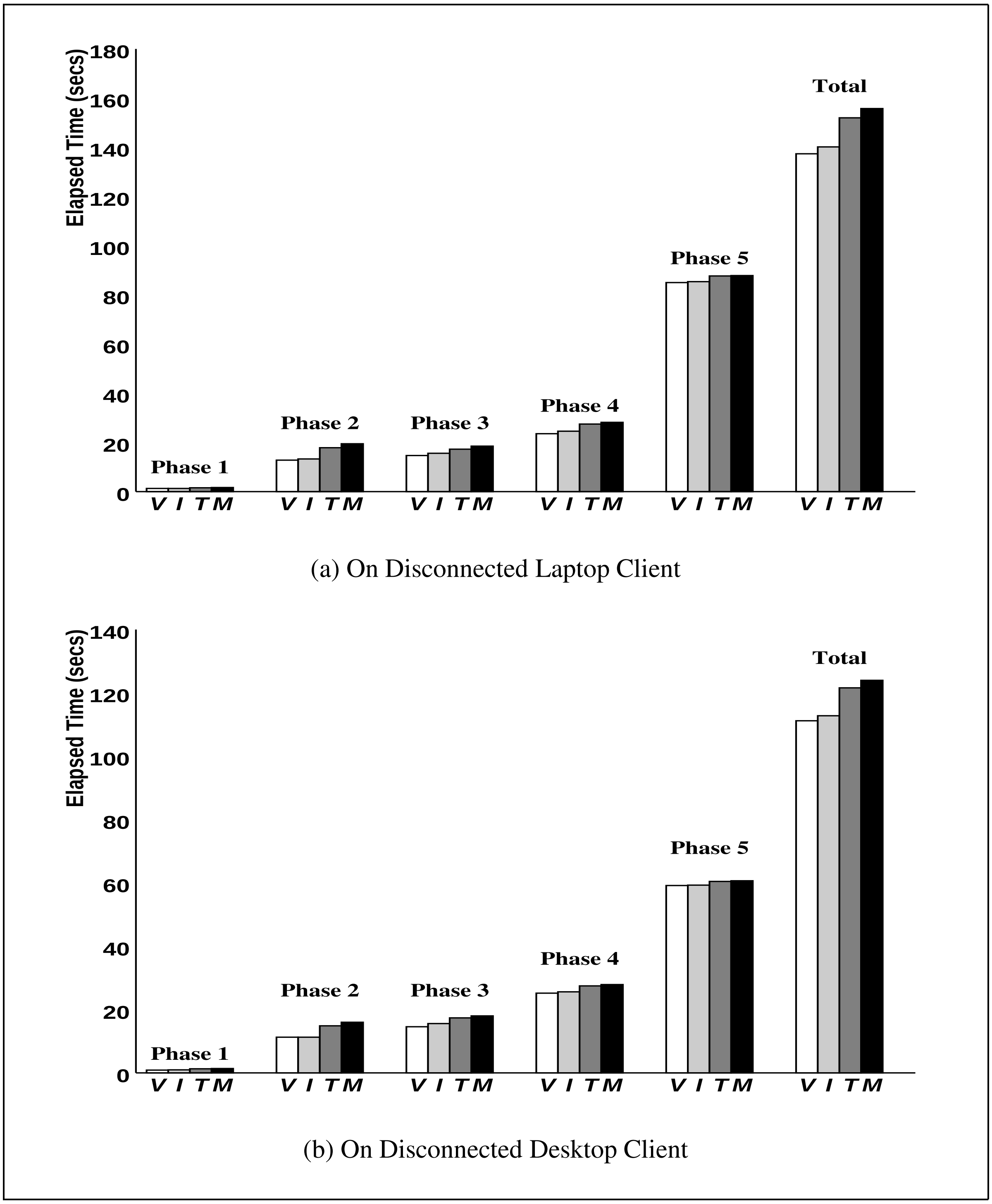
Phase	**Laptop**				**Desktop**			
	Single Transaction		**Multiple Transaction**		**Single Transaction**		**Multiple Transaction**	
	Elapsed Time (second)	Over-head (%)	Elapsed Time (second)	Over-head (%)	Elapsed Time (second)	Over-head (%)	Elapsed Time (second)	Over-head (%)
MakeDir	1.6 (0.5)	23.1	1.7 (0.7)	30.8	1.3 (0.5)	30.0	1.4 (0.5)	40.0
Copy	17.9 (1.0)	34.6	19.5 (1.1)	46.6	14.9 (1.1)	31.9	16.0 (1.1)	41.6
ScanDir	17.3 (0.8)	10.9	18.5 (0.8)	18.6	17.4 (0.7)	11.5	18.0 (0.9)	15.4
ReadAll	27.5 (1.3)	11.8	28.2 (0.9)	14.6	27.5 (1.1)	7.4	27.9 (1.0)	9.0
Make	87.7 (1.6)	2.7	87.9 (2.3)	2.9	60.5 (1.5)	2.0	60.7 (1.8)	2.4
Total	152.0 (2.2)	8.4	155.8 (2.8)	11.13	121.6 (2.7)	7.8	124.0 (1.9)	9.9

Table 9.7: Transaction Execution Performance of Andrew Benchmark

This table shows the elapsed time of executing the Andrew Benchmark first as a single transaction and then with each phase encapsulated in its own transaction, on disconnected laptop and desktop clients. The time values represent the mean over ten runs. The numbers in parentheses are standard deviations. The displayed performance overhead is relative to the elapsed time of executing the benchmark as a normal application on the IOT-Venus as shown in Table 9.2.

For example, the first two phases of the benchmark take a much worse performance hit than the overall benchmark because they contain consecutive `mkdir`, `create` and `store` operations which result in internal RVM transactions. Although the third and fourth phases also require updates in the transaction readset, the overhead is much lower because these phases spend time reading the contents of objects. This permits overlap of I/O from asynchronous RVM flushes. The last phase incurs a very small overhead because there is plenty of compilation time in between file access operations to accommodate more overlapping RVM flushes.

The overhead for each phase is higher in the multiple-transaction execution of the benchmark mainly due to the separate transaction initialization and finalization costs. The higher standard deviations for the `MakeDir` phase is due to the coarse metric of seconds. This magnifies slight timing differences in system internal activities such as RVM flushes.



(a) On Disconnected Laptop Client

(b) On Disconnected Desktop Client

The two graphs in this figure plot the performance data displayed in Table 9.2 and Table 9.7. The capital letters on the X-axis indicate the condition under which the benchmark is executed. **V** means that it is executed on the vanilla Venus as a normal application. **I** means that the execution is on the IOT-Venus as a normal application. **T** means that the benchmark is executed as a single transaction, and **M** means that each phase of the benchmark is executed as a separate transaction.

Figure 9.1: Performance Comparison for Andrew Benchmark

Trace	Laptop			Desktop		
	Normal Execution second)	Transaction Execution (second)	Over-head (%)	Normal Execution (second)	Transaction Execution (second)	Over-head (%)
Concord1	1659.8 (16.5)	1687.4 (20.8)	1.7	1624.6 (20.6)	1649.4 (21.3)	1.5
Concord2	1572.4 (5.4)	1597.8 (9.0)	1.6	1546.0 (12.7)	1553.4 (9.5)	0.5
Messiaen	1537.4 (8.8)	1564.0 (19.5)	1.7	1529.8 (6.7)	1536.8 (8.0)	0.5
Purcell	1570.2 (3.9)	1575.4 (5.0)	0.3	1607.6 (8.4)	1618.4 (6.1)	0.7

Table 9.8: Performance of Transactional Trace Replay with $\lambda = 1$

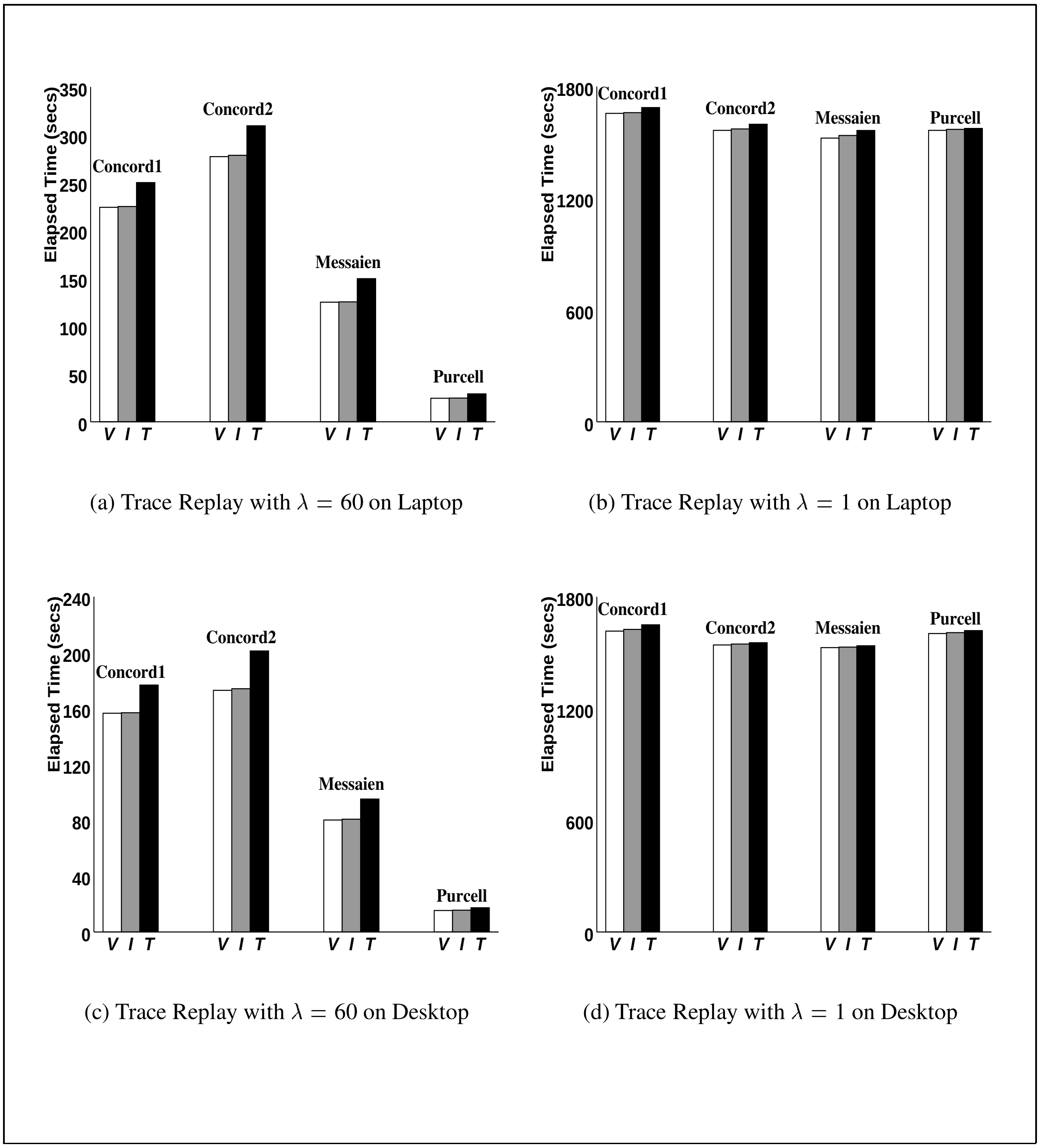
This table shows the elapsed time of running trace replay as a normal application and as a transaction on disconnected clients using the IOT-Venus. Because the λ parameter is set at 1, the total replay elapsed time is close to the original trace duration of 30 minutes. The time values represent the mean over five runs. The numbers in parentheses are standard deviations. The table also displays the performance overhead of transactional trace replay relative to normal trace replay on the IOT-Venus.

Trace Replay We also conducted trace replay experiments to examine the transaction performance overhead under actual workloads. Table 9.8 and Table 9.9 show the elapsed time of running the trace replay experiments as transactions and their performance overhead compared to running them as normal applications on the IOT-Venus. The same data in Table 9.3, Table 9.4, Table 9.8 and Table 9.9 are plotted into more informative graphs displayed in Figure 9.2.

Trace	Laptop			Desktop		
	Normal Execution (second)	Transaction Execution (second)	Over-head (%)	Normal Execution (second)	Transaction Execution (second)	Over-head (%)
Concord1	225.0 (8.3)	250.0 (10.8)	11.1	157.0 (4.7)	176.8 (5.9)	12.6
Concord2	278.6 (8.7)	309.4 (7.4)	11.1	174.2 (4.7)	201.2 (5.8)	15.7
Messiaen	125.4 (4.7)	149.6 (2.9)	19.3	80.8 (0.8)	95.2 (3.6)	17.8
Purcell	24.8 (0.8)	29.2 (1.6)	17.7	15.6 (0.5)	17.4 (0.5)	11.5

Table 9.9: Performance of Transactional Trace Replay with $\lambda = 60$

This table shows the elapsed time of running trace replay as a normal application and as a transaction on disconnected clients using the IOT-Venus. Because the λ parameter is set to 60, the replay is very I/O intensive with few delays between file references. The time values represent the mean over five runs. The numbers in parentheses are standard deviations. The table also displays the performance overhead of transactional trace replay relative to normal trace replay on the IOT-Venus.



The four graphs in this figure plot the trace replay performance data presented earlier in Table 9.3, Table 9.4, Table 9.8 and Table 9.9. Note that there are big differences in the time scale on the y-axis between the horizontally adjacent graphs. The letters **V, I, T** indicate the measurement of running the workload on vanilla-Venus, as a normal application on IOT-Venus, and as a transaction respectively.

Figure 9.2: Comparison of Trace Replay Performance

The performance of transactional replay on the four traces with $\lambda = 60$ demonstrate the negative impact of I/O intensity on transaction performance, causing degradations between 10% to 20%. They are still significantly lower than those of the first two phases of the Andrew Benchmark mainly due to their much longer execution durations, making RVM flushes less influential in total performance. However, they are slightly higher than that of the entire Andrew Benchmark because the trace replay transactions have much larger readsets and writesets, resulting in longer search times on transaction readset/writeset membership testing. Replaying the same traces with $\lambda = 1$ only results in less than 2% performance overhead.

	Laptop			Desktop		
Software Build Task	Normal Execution (second)	Transaction Execution (second)	Over-head (%)	Normal Execution (second)	Transaction Execution (second)	Over-head (%)
Venus	3679.2 (50.7)	3738.8 (34.5)	1.6	2976.0 (70.6)	3020.0 (60.1)	1.5
Server	998.6 (6.3)	1018.6 (3.6)	1.9	645.6 (3.4)	655.8 (11.3)	1.6

Table 9.10: Transaction Performance Overhead for Software Build Tasks

This table shows the elapsed time of building a Coda client and a Coda server both as a transaction and as a normal application on disconnected clients using the IOT-Venus. The time values represent the mean over five runs. The numbers in parentheses are standard deviations. This table also displays the transaction performance overhead comparing the two kinds of performance data listed in the table.

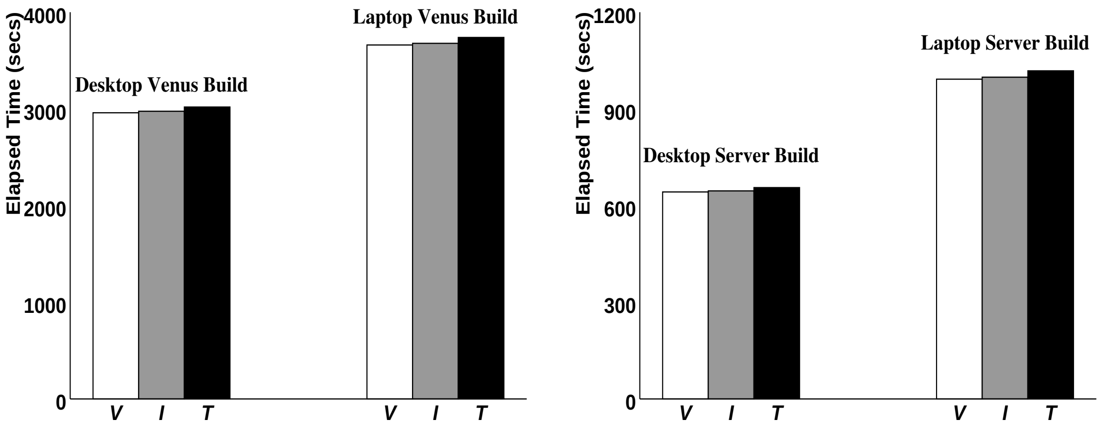
Software Build Tasks The measured elapsed time for the two software build tasks are presented in Table 9.10 and Figure 9.3. Because of the long execution duration, transaction-triggered RVM flushes have plenty of opportunities to be overlapped with computations performed by the compiler, linker, etc. Therefore, the main contributor to performance overhead becomes search activities for the transaction readset/writeset membership test, which incur less than 2% of performance overhead.

Document Build Tasks The measured elapsed time for the two document build tasks are presented in Table 9.11 and Figure 9.4. The performance overhead is a little higher than that of the two long-running software build tasks, mainly due to shorter execution duration. This allows IOT-generated RVM flushes to have a stronger negative impact on the overall performance.

Document Build Task	Laptop Normal Execution (second)	Laptop Transaction Execution (second)	Laptop Over-head (%)	Desktop Normal Execution (second)	Desktop Transaction Execution (second)	Desktop Over-head (%)
Thesis	146.0 (1.9)	150.8 (1.5)	3.3	96.4 (1.8)	99.2 (1.1)	2.9
Proposal	34.2 (0.5)	35.6 (0.6)	4.1	24.2 (0.8)	25.0 (1.2)	3.3

Table 9.11: Transaction Performance Overhead for Document Build Tasks

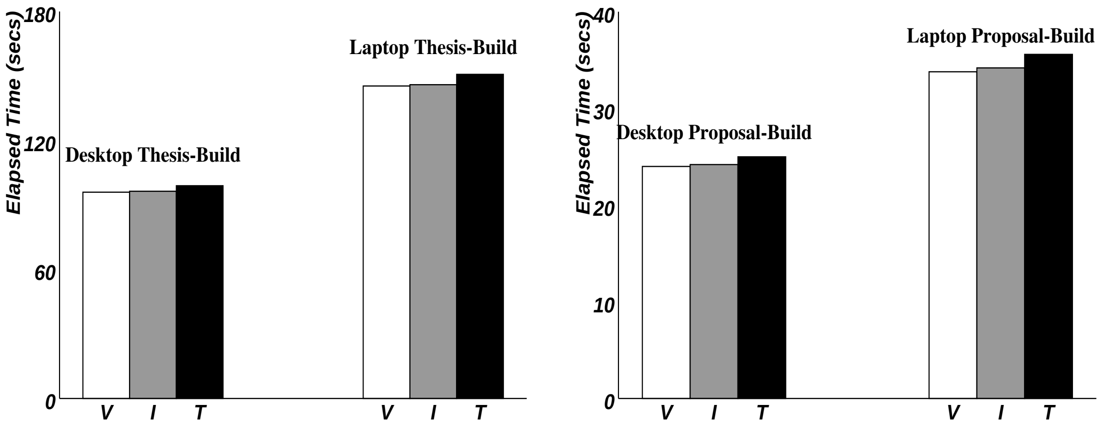
This table shows the elapsed time of typesetting a Ph.D. dissertation and a thesis proposal both as a transaction and as a normal application on disconnected clients using the IOT-Venus. The time values represent the mean over five runs. The numbers in parentheses are standard deviations. This table also displays the transaction performance overhead comparing the two kinds of performance data listed in the table.



(a) Venus Build Performance (b) Server Build Performance

The two graphs in this figure plot the performance data for the software build tasks presented earlier in Table 9.5 and Table 9.10. Note that there is a big difference in the time scale on the y-axis between the two graphs. The letters **V, I, T** indicate the measurement of running the workload on vanilla-Venus, as a normal application on IOT-Venus, and as a transaction respectively.

Figure 9.3: Comparison of Software Build Task Performances



(a) Thesis Typesetting Performance

(b) Proposal Typesetting Performance

The two graphs in this figure plot the performance data for the document build tasks presented earlier in Table 9.6 and Table 9.11. Note that there is a big difference in the time scale on the y-axis between the two graphs. The letters **V, I, T** indicate the measurement of running the workload on vanilla-Venus, as a normal application on IOT-Venus, and as a transaction respectively.

Figure 9.4: Comparison of Document Build Task Performances

9.2.2.3 Discussion

During the IOT design and implementation stages, we expected that the performance overhead for transaction execution would come from two main sources: transaction readset and writeset membership testing and RVM transactions for manipulating the persistent transaction data structures. We were very much concerned about the first source because our implementation uses a simple linked list to represent the transaction readsets and writesets. When the transaction size gets bigger, the quadratic growth of search time for set membership testing could incur significant slow down. But we decided to defer the use of more complex data structures until measurements indicate the necessity.

Our experiment results confirm the impact of the two sources on transaction performance. However, they also reveal the clear dominance of the second source in performance overhead which we did not fully anticipate. The first source turns out to be only mildly influential in transaction performance, even for large transactions such as the trace replay and Venus build tasks where the transaction readsets contain hundreds of files.

In summary, I/O intensive applications should expect a performance degradation between 10% to 20%. Generally speaking, given the same I/O intensity, the longer it takes to run the transaction the less performance penalty it suffers. Fortunately, file access operations are usually interleaved with application computation time and/or user think time in normal disconnected operation. Thus, the typical user observable performance degradation is likely to be around 3%, which is quite acceptable and in agreement with our qualitative usage experience.

Finally, the performance of the current implementation can be further fine tuned. Priority should be given to re-arranging the persistent transaction data structures to reduce RVM flush activities. In addition, using more advanced data structures such as hash tables in place of linked lists to represent transaction readsets and writesets could further reduce the performance overhead.

9.2.3 Performance of Automatic Resolution

Although automatic conflict resolution is expected to be used only occasionally, its performance still needs to be investigated to make sure that it can be done within a reasonably amount of time. Excessively slow resolution could hold up system resources for a long time causing great inconvenience.

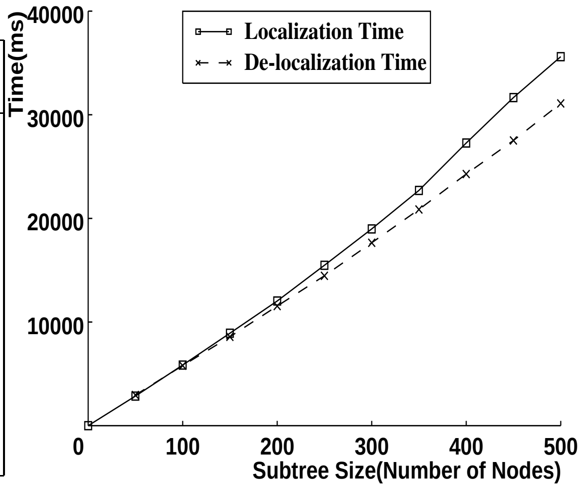
The latency of an automatic conflict resolution task depends mainly on the resolver involved, which could be an application-specific resolver or the application itself in the case of automatic re-execution. In the current implementation, the execution of a resolver is performed in a manner identical to a normal transaction. The performance overhead of this has already been evaluated in section 9.2.2. Therefore, what we evaluate here is the performance cost of resolution *initialization* and *finalization*. The main task of resolution initialization is to create the appropriate object views for the resolver, i.e., the *localization* of subtrees that are in conflict. Resolution finalization is mainly responsible for *de-localization*: i.e., removing the relevant localized subtrees and restoring the normal object view. The specific question we investigate is: *what is the latency associated with localization and de-localization?*

9.2.3.1 Methodology

The latency of localization and de-localization is determined by the size of the subtrees to be localized and de-localized. Since subtree size can vary over a wide range depending on the conflicts involved, we conducted a sensitivity analysis for this parameter. Based on the size of a typical subtree (to be discussed in section 9.3.2.3 on page 176), we can obtain an estimate of the typical latency for resolution localization and de-localization.

Our experiment first creates a subtree of specified size on a disconnected laptop client; it then performs a conflicting mutation through another connected client; finally the first client is reconnected to the servers. We measure the total elapsed time for the laptop client to localize the subtree after the conflict is detected. Similarly, we measure the elapsed time of de-localizing the subtree after discarding all local mutations using the repair tool.

Subtree Size	Localization Latency(ms)	De-localization Latency(ms)
50	2855 (28)	2890 (194)
100	5852 (208)	5804 (232)
150	8921 (221)	8579 (157)
200	12033 (39)	11553 (243)
250	15479 (119)	14473 (176)
300	18985 (69)	17655 (197)
350	22707 (61)	20870 (268)
400	27286 (640)	24282 (237)
450	30735 (264)	27140 (195)
500	35617 (309)	31096 (65)



The table in this figure shows the latency of localization and de-localization of subtrees of different sizes. The time values are in milliseconds and represent the mean over ten runs. The numbers in parentheses are standard deviations. The same data are plotted in the graph to present a visual display of the relationship between localization/de-localization latency and subtree size.

Figure 9.5: Latency of Localization and De-localization

9.2.3.2 Results and Discussion

Figure 9.5 shows the measured latencies of localization and de-localization for different subtree sizes. A simple linear regression finds good fit for both curves. For localization latency, the regression coefficient is 70.62 with respect to the size of the subtree and the R^2 value is .994. For de-localization, the regression coefficient is 61.56 and the R^2 value is .998. Both latencies grow slightly faster than linear because they involve quadratic components in their operations such as scanning the cached object database to perform fid-translation. The graph also shows that the latency for localization grows a bit faster than that of de-localization. This is because localization needs to perform more internal operations such as checking for un-cached objects within a local subtree.

To obtain an estimate of latency for resolution localization and de-localization, we need to know the number of nodes in a local subtree and the number of local subtrees associated with a transaction to be resolved. The analysis in Section 9.3.2.4 on page 177 shows that a typical local subtree contains about 30 nodes. Our experience indicates that a non-certifiable transaction typically has two local subtrees. In this case, the resolution localization/de-localization latency should typically be between 4 and 5 seconds.

While 4 to 5 seconds may seem high, the cost must be considered within context of automatic conflict resolution. First, the automatic resolution for typical applications such as `make` usually involves a fair amount of computation, making such latency insignificant. More importantly, transparent conflict resolution relieves the user from spending possibly much more time to manually repair conflicts. Thus, a few seconds is a small price to pay.

9.2.4 Other Performance Issues

So far we have evaluated the primary factors reflecting the overall IOT performance for common applications in disconnected operation. In this section, we address some of the secondary performance issues.

Connected Transaction Execution We do not measure the performance of transaction execution in a connected environment for two main reasons. First, since IOT is intended to be used mainly on disconnected mobile client machines, connected transaction performance is not a particularly important metric. Second, by design, the performance overhead of connected transaction execution is basically the same as that of disconnected execution. The only difference is that connected transaction execution has the *write-back caching* effect for its mutation operations. Even on a fully connected client, all the mutations performed by a transaction are logged and committed (reintegrated) at the end. This is similar to the write-back caching policy for propagating updates from a client to the servers. As is well known, write-back caching offers superior performance to the *write-through caching* policy currently employed by Coda. Hence, it is possible for transactional execution of a particular application to take less time than its non-transactional execution on a connected client. To avoid misleading results resulting from this major difference, we decided not to evaluate connected transaction performance.

Global Concurrency Control For connected transaction execution, a potential performance cost is the need to automatically re-execute transactions when cross-client read/write sharing is detected. This is, of course, intrinsic to the OCC concurrency control scheme. We do not have enough usage information to offer a reasonable estimation of how likely OCC re-execution will be. Nor do we have sufficient data to design meaningful controlled experiments that can yield insightful results. There have been OCC performance studies in the literature [71, 6, 75], but they mostly assume a traditional database environment. To the best of our knowledge, there has been no actual transaction system in practical use employing OCC as its concurrency control algorithm. A credible study of the performance impact of OCC in a distributed file system environment needs to await adequate usage experience.

Two Phase Commitment The 2PC protocol for distributed transaction commitment will cause additional performance overhead. Since 2PC has not been fully implemented due to time constraints, we cannot measure its actual performance impact. In practice, however, 2PC has very little effect on transaction commitment performance because almost all of the transactions we experienced only update data in a single volume. In such a situation, the atomicity of transaction commitment is guaranteed by the current Coda's underlying reintegration process.

Local Concurrency Control IOT uses strict 2PL for local concurrency control among transactions (both IOT and IFT). The performance of a file access operation will be affected whenever it is in conflict with an ongoing IOT. We exclude 2PL from the evaluation for two main reasons. First, a disconnected client is typically operated by a single user and the likelihood of executing concurrent transactions performing conflicting accesses on shared data is very low. Second, the performance impact of 2PL depends on the data sharing pattern among concurrent applications. We do not have enough transaction usage experience to design meaningful experiments to measure such effect.

9.2.5 Summary

It is still premature to draw definitive conclusions about the overall IOT performance when the system has only been used by a few users. However, the experiments we have conducted provide substantial evidence to support the following characterizations.

The performance degradation for normal file access operations is small and barely noticeable for most disconnected activities. The performance overhead of running common applications as transactions is generally around 3%. When the I/O intensity of the applications increases, the performance overhead becomes higher, typically in the range of 10-20%. Long-running transactions tend to suffer less from IOT incurred overhead because it allows more IOT-generated internal disk writes to overlap with application computation or user think time.

Some of the performance overhead is due to certain specific implementation choices and could be improved by using better alternatives. In summary, there is sufficient evidence to believe that the IOT model can be realized at modest performance cost.

9.3 Resource Cost Measurement

The discussion in the previous section focused entirely on the client CPU overhead of using IOT. But the use of IOT also incurs other overheads, such as client memory, server CPU and network bandwidth. The current IOT implementation ensures that normal file access operations do not increase any system resource usage other than client CPU cycles. Hence, we only examine resource costs associated with transactional operations. The key questions we want to investigate are:

1. *Which system resources are subject to increased consumption by transaction execution?*
2. *What is the overhead of executing a common application as a transaction for each kind of affected resource?*

We classify system resources into two broad categories: *local system resources* and *global system resources*, and study the IOT impact on their usage separately.

9.3.1 Global System Resources

Global system resources refer to system resources outside of a Coda client such as network bandwidth, server CPU time and server disk space. Because the current IOT implementation requires no change to any server internal data structures, transaction execution does not cost any additional server disk space. Hence, our evaluation focuses on two main global resources: *server load* and network traffic. We use the term server load to refer to the total amount of server CPU and server I/O time spent on behalf of a particular system task associated with transaction operations. We studied the following two specific questions:

1. *How is server load affected by transaction-related system activities?*
2. *How is network traffic affected by transaction-related system activities?*

There are three kinds of transaction related activities that consume global system resources: transaction reintegration, transaction validation, and connected transaction execution. We first present the measurements of global system resource cost incurred by transaction reintegration, and then discuss the impact of transaction validation and connected transaction execution on global system resource usage.

9.3.1.1 Server Load for Reintegrating Disconnected Transactions

Methodology When there are no transaction executions, mutations performed in a disconnected operation session are reintegrated to the servers in one batch requiring a single reintegration operation on the corresponding servers. When there are disconnected transactions, the mutations will be reintegrated in different batches requiring multiple reintegration operations on the servers. Thus, the impact of reintegrating disconnected transactions on server load boils down to reintegrating the same set of mutations in one batch versus in multiple batches.

The server load for reintegrating a set of mutations depends on many factors such as the number, the type and the mixture of the involved mutation operations. Overall, the reintegration server load can be considered as consisting of two main factors: a fixed initial setup cost and the cost that is proportional to the number of mutation operations involved. When the number of mutations is small, the first factor dominates the reintegration server load. In contrast, when the number of mutations is large, the second factor dominates. Moreover, it grows at a faster than linear speed because it involves activities such as sorting.

Experiment Workload	**Reintegration Server Load**	
	One Run (millisecond)	Two Runs (millisecond)
Andrew Benchmark	11003 (617.3)	45429 (1276)
CFS-Build	912 (8.2)	1724.6 (20.6)

This table shows the total elapsed time for a dedicated server to perform reintegration for the disconnected mutations of one and two independent runs of the Andrew Benchmark and CFS-build task. The time values are in milliseconds and represent the mean over five runs. The numbers in parentheses are standard deviations.

Table 9.12: Impact of Disconnected Transactions on Reintegration Server Load

As a result, the impact of disconnected transactions on the total reintegration server load can go either way. Generally speaking, when the total number of mutations is small, disconnected transactions will increase the reintegration server load. On the other hand, when the total number of mutations is large, disconnected transactions can reduce the reintegration server load. We use two experiments to demonstrate this effect.

The first experiment compares the server load of reintegrating one and two independent runs of the Andrew Benchmark, which contains a large number of mutations. The second experiment compares the server load of reintegrating one and two independent runs of the CFS-build task, which compiles the Coda `cfs` tool and contains only a few mutations. Each experiment run consists of the execution of the workload (one or two independent runs of the Andrew Benchmark and CFS-build task) on a disconnected laptop client and the ensuing reintegration from the laptop client to a dedicated server. In order to eliminate possible interference from other clients, we use a separate network between the client and the server during reintegration and make sure that there are no other concurrent threads or RVM activities on both the client and the server during reintegration.

Results The results of the two experiments are shown in Table 9.12. Because the Andrew Benchmark contains a lot of mutations, the server elapsed time for reintegrating two disconnected benchmark runs together is much bigger than the sum of reintegrating the two runs one at a time. In contrast, the CFS-build task contains only a few mutations. Hence, reintegrating the two runs separately costs more server time than reintegrating them together. Suppose that there is a disconnected operation session containing two independent runs of the Andrew Benchmark, the reintegration server load will decrease when either of the two runs is executed as a transaction. Conversely, if the disconnected operation session contains two independent runs of the CFS-build task, using a transaction for either of the two runs will increase the reintegration server load.

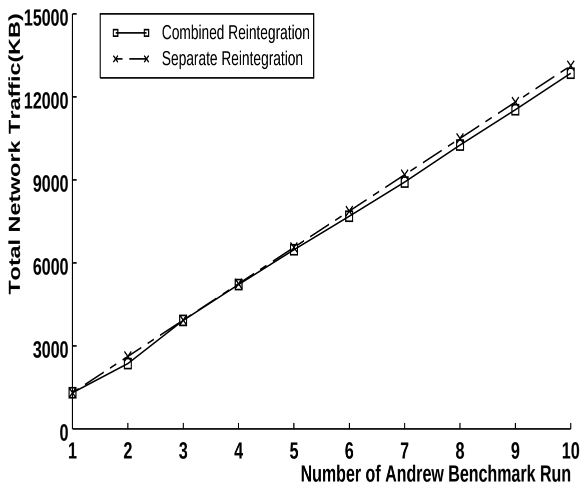
9.3.1.2 Network Traffic for Reintegrating Disconnected Transactions

Methodology If a disconnected operation session does not contain any transaction execution, all disconnected mutations are sent to the servers using one reintegration RPC. When disconnected transactions are involved, the same set of mutations will be broken up into several smaller reintegration RPC calls. This results in network traffic overhead because transmitting the same amount of data using multiple RPC calls consumes more packets than a single RPC call. Unlike server load, disconnected transactions always increase reintegration network traffic.

We use multiple independent runs of the Andrew Benchmark to measure the network traffic overhead by comparing reintegrating the multiple runs using a single RPC to that using one RPC per run. The experiment was conducted in the same environment as described in section 9.3.1.1.

Results The measured results displayed in Figure 9.6 indicate that there is only a slight overhead in reintegration network traffic for disconnected transactions containing a large number of mutations, such as the Andrew Benchmark. The overhead could be higher when the involved disconnected transactions contain only a few mutation operations.

Run Number	Reintegration Traffic(KB)
1	1313 (7.6)
2	2366 (75.2)
3	3923 (30.3)
4	5222 (44.5)
5	6476 (34.7)
6	7689 (19.3)
7	8925 (40.3)
8	10262 (38.8)
9	11537 (24.8)
10	12858 (37.7)



The table in this figure shows the measured network traffic for reintegrating disconnected mutations of multiple independent runs of the Andrew Benchmark. The metric used is KB and the values represent the mean over five runs. The number in parentheses are standard deviations. The two curves on the right plot the same data presented in the table and a linear projection based on the reintegration traffic of a single benchmark run.

Figure 9.6: Reintegration Traffic for Multiple Runs of Andrew Benchmark

9.3.1.3 The Impact of Transaction Validation

Transaction validation as currently designed is just comparing version vectors for the involved objects. We do not measure its effect on both server load and network traffic because it does not have any long term effect on these two global system resources. The main reason is that the internal mechanisms for transaction validation are overloaded with those for cache coherence maintenance, as discussed in section 8.6.1. In essence, the server workload and network traffic spent on behalf of validating a transaction will relieve the same amount of work that otherwise would have been carried out by client cache validation and callback maintenance, and vice versa.

9.3.1.4 The Impact of Connected Transaction Execution

Connected transaction execution has an impact on both the server load and the network traffic. There are two main factors: the write-back caching effect due to mutation logging and the 2PC protocol for distributed transaction commitment. Obviously, 2PC will increase both the server load and the network traffic. However, the write-back caching effect of mutation logging can influence both the server load and network traffic in either direction due to the fact that the current Coda implementation uses a write-through caching policy.

Connected transactional execution of applications containing a large number of mutation operations can reduce both the server load and network traffic compared to connected non-transactional execution. There are two main reasons. First, because mutations get batched at the client, there are opportunities to cancel redundant mutation operations as discussed in Chapter 4. Second, it consumes less network traffic and server load to transmit and perform a large number of mutations at once than to process them one at a time. On the other hand, both the server load and network traffic can be increased by connected transaction execution if the application contains only a few mutation operations because the initial overhead of reintegration will dominate the cost.

We decided not to evaluate the effect of connected transaction execution on server load and network traffic for the following reasons. First, the 2PC protocol has not been fully implemented yet. Thus, how it increases the server load and network traffic will not be known until the actual mechanisms are put in place. Second, a fair comparison on the server load and network traffic between connected transactional and non-transactional executions cannot be made until Coda implements a write-back caching policy.

9.3.2 Local System Resources

With the continuing trend of miniaturization of portable computers, a mobile client is likely to remain resource poor compared to its stationary counterpart. Hence, minimizing the consumption of resources local to a mobile client machine is critical to the viability of the IOT model. Two kinds of local resources are of primary concern: disk space and RVM space. Local resources are heavily used and sometimes in shortage during disconnected operation. Our evaluation concentrates on local resource cost of disconnected transactions. The key questions to be addressed are:

1. *What is the disk space and RVM space cost for executing a typical application as a transaction?*

2. *How long can a disconnected client support transaction operations in the two target application domains before exhausting local resources?*

Disconnected transaction execution consumes additional client disk space in two main categories: shadow cache files and the cache files of local objects used in conflict representation. RVM space is used for storing persistent transaction information such as readsets and writesets, local objects in conflict representation, the serialization graph, the wait-for graph and other miscellaneous items. We only measure the first two contributors to RVM usage because the rest of the items usually consume only a trivial amount of RVM space.

9.3.2.1 Disk Space Cost for Shadow Cache Files

Methodology A shadow cache file is created when its corresponding Coda object is to be updated and its present content has been accessed by at least one live transaction (other than the transaction that is doing the update). Reclamation of a shadow file occurs when all the live transactions that accessed the shadow content are committed or resolved.

For a disconnected operation session, the total amount of disk space used for maintaining shadow cache files is mainly decided by the amount of sequential read/write sharing among live transactions (both IOTs and IFTs) executed during the same disconnected operation session. Estimating disk usage for shadow cache files is difficult because of many complicating factors such as the number, the size and the distribution of transactions as well as the sizes of objects accessed by transactions.

Trace Identifier	Machine Name	Machine Type	Simulation Start	Records
Work-Day #1	brahms.coda.cs.cmu.edu	IBM RT-PC	25-Mar-91, 11:00	197,985
Work-Day #2	holst.coda.cs.cmu.edu	DECstation 3100	22-Feb-91, 09:15	354,105
Work-Day #3	ives.coda.cs.cmu.edu	DECstation 3100	05-Mar-91, 08:45	136,425
Work-Day #4	mozart.coda.cs.cmu.edu	DECstation 3100	11-Mar-91, 11:45	239,668
Work-Day #5	verdi.coda.cs.cmu.edu	DECstation 3100	21-Feb-91, 12:00	299,560
Full-Week #1	concord.nectar.cs.cmu.edu	Sun 4/330	26-Jul-91, 11:41	4,008,084
Full-Week #2	holst.coda.cs.cmu.edu	DECstation 3100	18-Aug-91, 23:31	2,303,306
Full-Week #3	ives.coda.cs.cmu.edu	DECstation 3100	03-May-91, 23:21	4,233,151
Full-Week #4	messiaen.coda.cs.cmu.edu	DECstation 3100	27-Sep-91, 00:15	1,634,789
Full-Week #5	purcell.coda.cs.cmu.edu	DECstation 3100	21-Aug-91, 14:47	2,193,320

The `Records` column refers to the number of trace records that are actually processed by the trace simulator during the simulated period, i.e., between simulation-start and simulation-start plus 12 or 168 hours.

Table 9.13: Information for the Work-Day and Full-Week Traces

To obtain a reliable measurement on how much shadow space is needed for a typical transaction and the accumulated shadow space cost over an extended period of disconnected transaction operations, we decided to simulate disconnected transaction executions using collected file reference traces. We choose the same ten traces that were used in a previous Coda performance study [26]. They were carefully screened to ensure that they contain active file references and the main application domains involved are software development and document processing. There are five "Work-Day" and five "Full-Week" traces that are 12 hours and 168 hours long respectively and cover a typical working day or week for the primary user of the workstation. Table 9.13 lists key information about each of the selected traces.

The traces are fed to a simulator that mimics the space allocation and de-allocation activities for disconnected transaction executions. The simulator takes as argument a list of pathnames for those applications that are to be simulated as transactions. It tracks all the `fork` and

Application	**Path Name**	**Occurrence**
`awk`	/bin/awk	8.72%
`cc`	/usr/cs/bin/cc	5.26%
`cp`	/bin/cp	2.12%
`cpp`	/usr/cs/lib/cpp	0.34%
`emacs`	/usr/cs/bin/emacs	1.10%
`find`	/usr/cs/bin/find	1.44%
`ld`	/usr/cs/bin/ld	0.04%
`make`	/usr/cs/bin/make	6.61%
`rcsci`	/usr/misc/bin/rcsci	0.07%
`rcsco`	/usr/misc/bin/rcsco	0.42%
`scribe`	/usr/misc/bin/scribe	0.34%
`sed`	/bin/sed	8.29%
`sh`	/bin/sh	64.30%
`vi`	/usr/ucb/vi	0.95%

This table displays the list of applications that are to be simulated as transactions. The "Occurrence" column shows the percentage of each application among the total number of transactions simulated. `awk` and `sed` are selected because both are frequently used script languages. `make`, `cc`, `cpp` and `ld` are commonly used in software development using the C programming language. `scribe` is the only typesetting tool found in the traces, whereas the more popular `latex` is notably absent from all trace records. Both `emacs` and `vi` are commonly used interactive editors at the time, while `rcsci` and `rcsco` are important tools for maintaining source code revisions. Finally, `sh` is chosen because important tasks are often carried out via a shell script.

Table 9.14: Simulated Transaction Applications

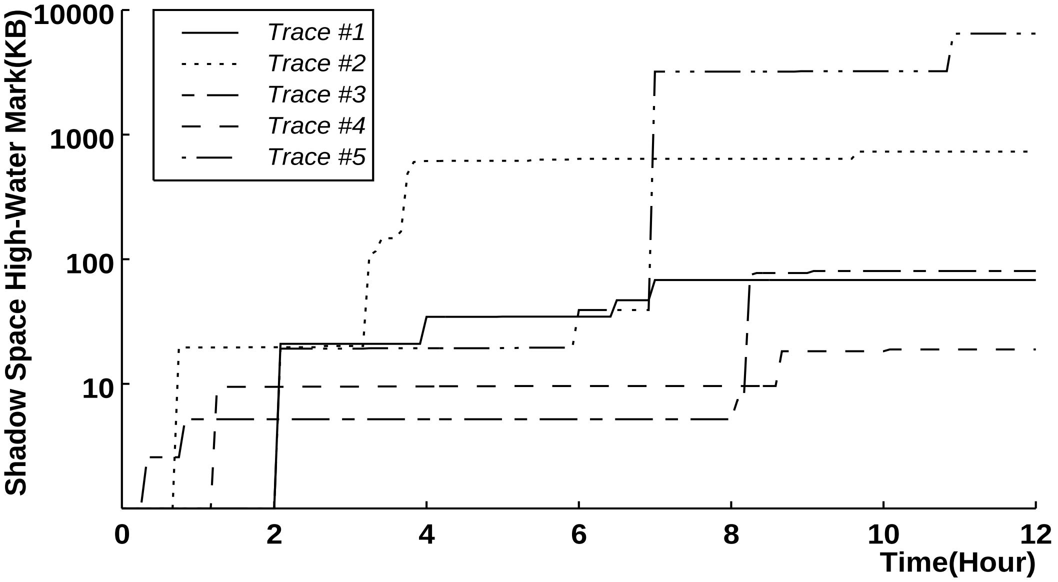
`execve` trace records that are performed on the selected applications and uses process group id to identify the file reference trace records that belong to the corresponding transactions. In order to accurately reflect important aspects of disconnected transaction execution such as transaction optimization, the simulator maintains a mini-database of all accessed objects and imitates key internal activities of the Venus cache manager and the transaction system. The simulator is written based on some existing trace analysis tools and contains over 5000 lines of C++ code, not including the linked libraries from the existing trace analysis tool package.

A key variable in this experiment is the selection of applications to be simulated as transactions. We manually screened all the *execve* trace records and chose fourteen applications (listed in Table 9.14) based on their importance in our target application domains and their frequency of occurrence in the traces. Other applications were not chosen either because we deemed them of lower importance or because they do not show up in the traces.

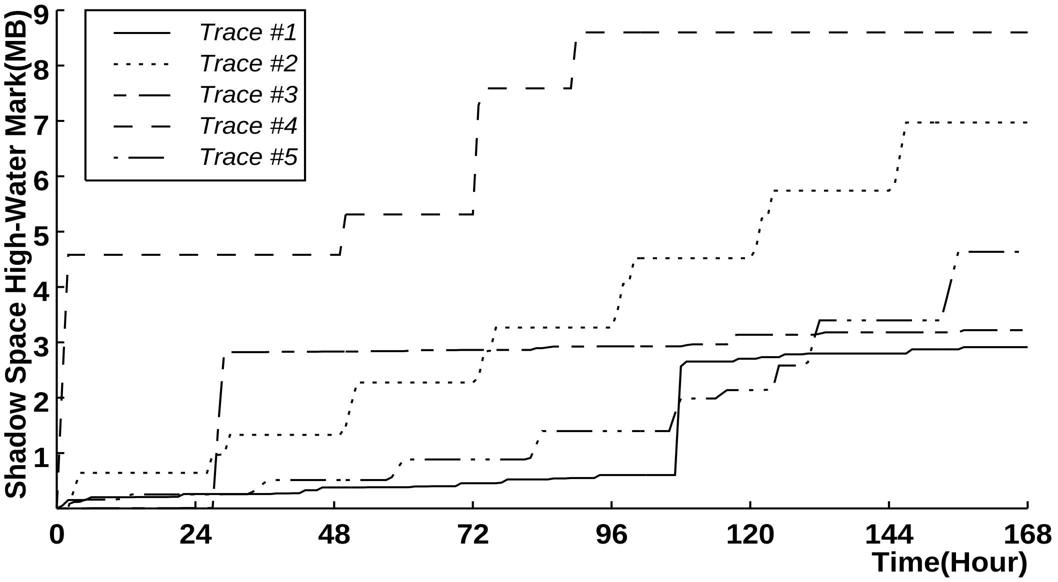
Results Because the trace simulation experiment produced a large quantity of informative results, we discuss them in three steps: (a) how much disk space is needed to maintain shadow cache files for an extended period of disconnected transaction operations; (b) how much shadow space is needed for each individual transaction; (c) what is the key factor in keeping the shadow space cost low.

First, Figure 9.7 shows the high-water mark of shadow space cost for the ten traces, where each trace is interpreted as an extended disconnected operation session. A majority of the Work-Day disconnected sessions require less than 100KB of shadow space. One of the traces results in a much higher cost at about 6.4MB. The reason for this anomaly is that there are several `make` transactions compiling the Coda Venus module that end up saving two shadow copies of the 3MB Venus binary. In case shadow space is exhausted, the two `make` transactions will lose their shadow Venus binary. Since binaries are easily regeneratedable from the relevant source files, discarding the binaries will not pose serious problems for a future resolution of these `make` transactions.

The shadow space cost in the five Full-Week sessions has much less variance, with the highest cost less than 9MB and the average cost about 5MB. Similar to the Work-Day sessions, many of the repeatedly shadowed objects are software target objects such as the library file `librvm.a` (about 700KB) and the executable file `rvmutil` (about 500KB) in trace #4. The curve corresponding to the second trace is shaped like a regularly increasing step function because of a shell script that is executed daily over-writing a large sup [65] log file.



(a) Work-Day Traces



(b) Full-Week Traces

The two graphs in this figure show the high-water marks of shadow space cost recorded by the simulator for the five Work-Day and five Full-Week traces listed in Table 9.13. Note that the y-axis in the Work-Day traces uses log scale to better represent the shadow space cost over time because one of the traces has much higher cost than the others.

Figure 9.7: High-Water Marks of Shadow Space Cost

Trace	Total Tran. Count	Live Transaction Count	Read Only Transaction Count	Cancelled Transaction Count	Total File Reference Count	Transactional File Reference Count
#1	49	13(26.5%)	17(34.7%)	19(38.8%)	197,985	20,915(10.6%)
#2	88	27(30.7%)	25(28.4%)	36(40.9%)	354,105	157,909(44.6%)
#3	50	14(28.0%)	19(38.0%)	17(34.0%)	136,425	12,321(9.0%)
#4	26	9(34.6%)	7(26.9%)	10(38.5%)	239,668	5,298(2.2%)
#5	23	8(34.8%)	4(17.4%)	11(47.8%)	299,560	104,315(34.8%)
Avg	47.2	14.2(30.1%)	14.4(30.5%)	18.6(39.4%)	245,548.6	60,151.6(24.5%)

(a) Work-Day Traces

Trace	Total Tran. Count	Live Transaction Count	Read Only Transaction Count	Cancelled Transaction Count	Total File Reference Count	Transactional File Reference Count
#1	1028	51(5.0%)	277(26.9%)	700(68.1%)	4,008,084	2,596,980(64.8%)
#2	781	86(11.0%)	182(23.3%)	513(65.7%)	2,303,306	1,267,155(55.0%)
#3	495	57(11.5%)	231(46.7%)	207(41.8%)	4,233,151	396,427(9.4%)
#4	142	40(28.2%)	21(14.8%)	81(57.0%)	1,634,789	442,855(27.1%)
#5	952	63(6.6%)	514(54.0%)	375(39.4%)	2,193,320	642,647(29.3%)
Avg	679.6	59.4(8.7%)	245(36.1%)	375.2(55.2%)	2,874,530	1,069,212.8(37.2%)

(b) Full-Week Traces

Table 9.15: Transaction and File Reference Statistics of Trace Simulation

This table shows statistics of transaction and file reference activities during trace simulation such as the number of transactions simulated during Week-Day and Full-Week, the number of transactions that are cancelled, the number of file references that are issued by transactions, and the total number of file references in each trace.

The shadow space cost is small considering the length of disconnection, the rapidly growing disk capacity on portable computers, and particularly the amount of disconnected transaction activity shown in Table 9.15. On average, 47 transactions are executed on each Work-Day and about a quarter of the file references are performed by transactions. During a Full-Week, an average number of 680 transactions are executed and about 37% of the file access operations are issued by transactions.

Second, Table 9.16 contains detailed transaction information about each selected application during the trace simulation. It displays important statistics such as the total number of transactional executions for each selected application, the number of transactions that are cancelled, the average size of the readset and writeset, as well as detailed resource cost information. The average shadow space cost associated with each live transaction is very small (less than a couple of hundred bytes) for applications such as `awk`, `cpp`, `ld`, `rcsci`, `rcsco`, `scribe`, `sed` and `vi`. Other applications such as `emacs` use more shadow space but average only around a dozen KB.

The `make` transactions are the biggest shadow space consumers, which average around 718KB per transaction during Work-Day and 295KB per transaction during Full-Week. The other big shadow space consumer is `sh`, which uses an average of 284KB per transaction during Work-Day and 111KB per transaction during Full-Week. These two applications tend to access large objects and have read/write sharing with subsequent transactions. However, large shadow files are often associated with objects that can be automatically re-generated such as library and executable files. Thus, the consequence is rather acceptable even if the limit on shadow space is exhausted and some transactions must lose their shadow cache files.

Third, the modest shadow space usage for disconnected transactions is primarily due to the effectiveness of transaction cancellation. In order to find out the exact impact of transaction cancellation on shadow space cost reduction, we perform the same trace simulation experiment to measure the shadow space cost without transaction cancellation, and the results are displayed in Figure 9.8. Without transaction cancellation, the highest shadow space cost for Full-Week is close to 115MB and the average cost is over 40MB. Compared to the data shown in Figure 9.7, the reduction in shadow space cost is about one order of magnitude. Another important issue is that the longer the disconnected operation duration, the more effective the transaction cancellation is. From Table 9.15, an average of 39.4% of transactions are cancelled during Work-Day and an average of 55.2% of transactions are cancelled during Full-Week.

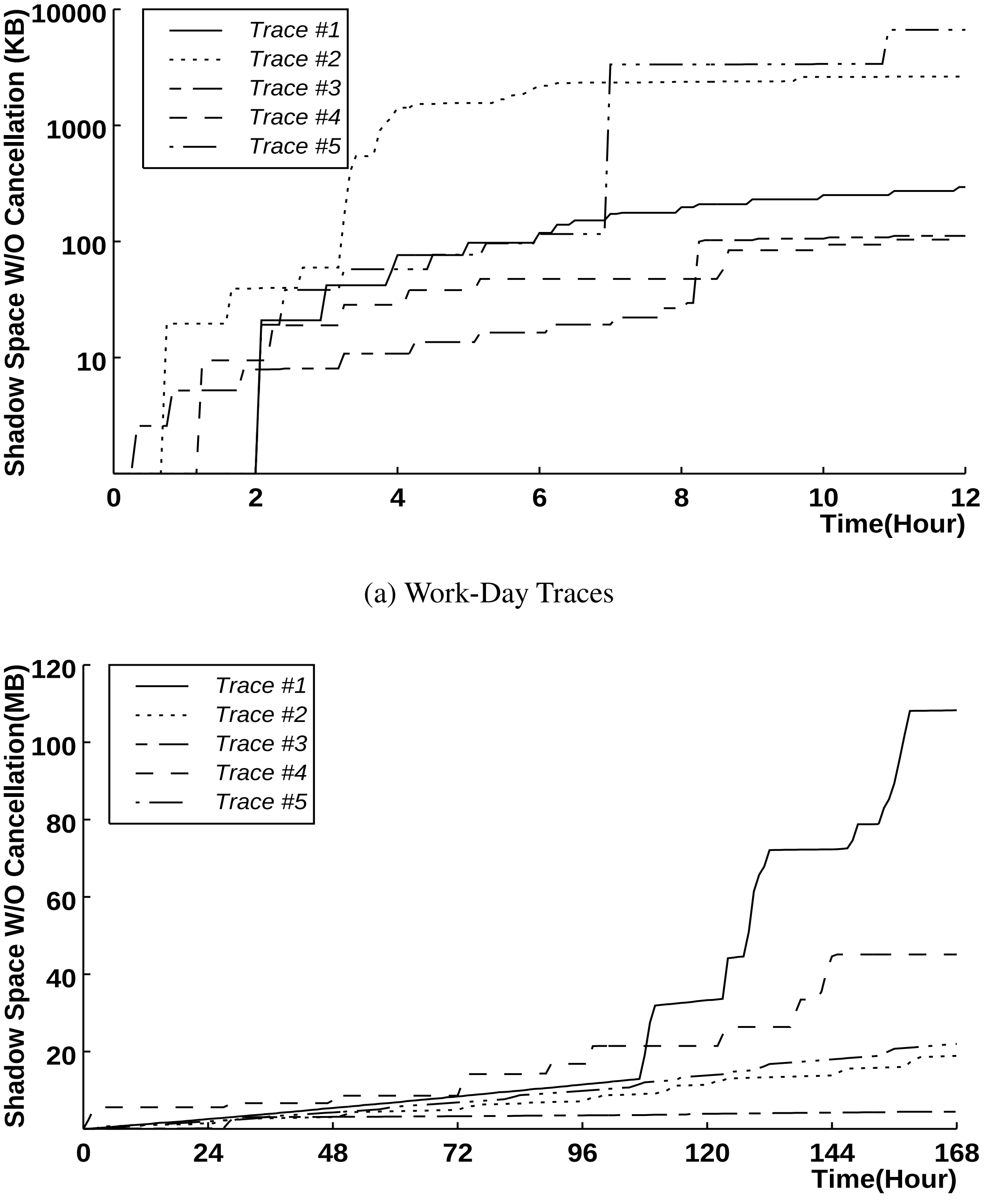
App. Name	Total Trans. Count	Live Trans. Count	Read Only Trans. Count	Cancelled Trans. Count	Average Read Set Size	Average Write Set Size	Average RVM Space Cost(B)	Average Shadow Space Cost(B)
awk	6	0	4	2	1.0	0.3	225.0	0.0
cc	11	7	1	3	10.4	2.4	603.4	17.7
cp	27	19	3	5	22.0	11.1	1,562.4	211.9
cpp	10	0	10	0	16.7	0.0	805.9	0.0
emacs	14	13	1	0	22.9	2.6	1,042.4	2,089.8
find	3	0	3	0	1239.3	0.0	46,043.3	0.0
ld	1	0	0	1	7.0	1.0	447.0	0.0
make	28	10	4	14	78.9	4.8	3,121.0	718,190.7
rcsci	1	1	0	0	13.0	7.0	785.0	0.0
rcsco	5	3	2	0	7.6	3.4	540.0	24.0
scribe	9	4	2	3	10.6	2.1	614.3	8.8
sed	16	0	16	0	0.0	0.0	190.3	0.0
sh	99	12	26	61	14.5	1.5	730.0	284,348.2
vi	6	2	0	4	4.5	2.5	362.7	0.0

(a) Work-Day Traces

App. Name	Total Trans. Count	Live Trans. Count	Read Only Trans. Count	Cancelled Trans. Count	Average Read Set Size	Average Write Set Size	Average RVM Space Cost(B)	Average Shadow Space Cost(B)
awk	311	0	309	2	0.02	0.01	188.8	0.0
cc	180	15	6	159	18.1	2.7	862.7	51,992.3
cp	50	33	10	7	30.5	15.3	2,126.3	28,770.2
cpp	2	0	0	2	51.5	1.0	2,093.5	0.0
emacs	26	22	1	3	35.6	4.0	1,543.3	11,054.0
find	49	2	46	1	62.5	0.3	2,512.5	98,906.0
ld	0	0	0	0	0	0	0	0
make	212	40	14	158	108.7	13.6	4,726.4	294,566.5
rcsci	1	1	0	0	13.0	7.0	779.0	0.0
rcsco	10	9	0	1	9.0	4.6	614.4	384.0
scribe	3	2	0	1	13.7	3.0	745.7	17.5
sed	285	1	279	5	0.1	0.02	193.1	0.0
sh	2241	165	561	1515	50.9	2.5	2,108.1	111,366.7
vi	28	6	0	22	5.5	2.4	390.2	99.7

(b) Full-Week Traces

Table 9.16: Transaction Application Statistics Of Trace Simulation



(a) Work-Day Traces

(b) Full-Week Traces

The two graphs in this figure show the high-water marks of recorded shadow space cost when the simulator does not perform transaction cancellation for the 10 traces listed in Table 9.13. Similar to Figure 9.7, the y-axis in the Work-Day traces uses log scale.

Figure 9.8: Shadow Space Cost Without Transaction Cancellation

9.3.2.2 RVM Space Cost for Persistent Transaction Data Structures

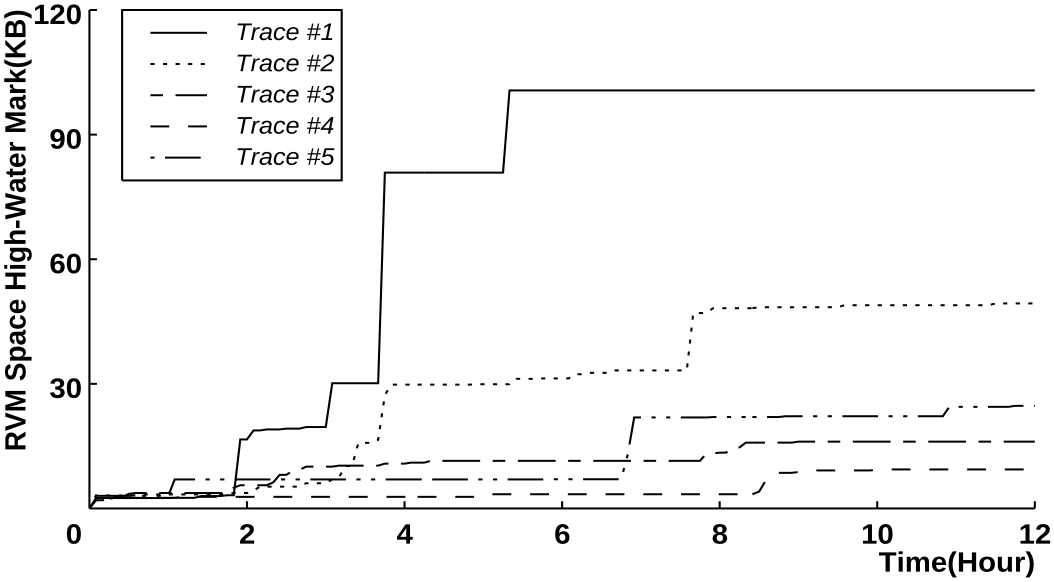
Methodology For each transaction, key information such as the readset and writeset, the list of accessed volumes, the conflict resolution option, and the execution environment needs to be stored in RVM. These persistent transaction data structures are maintained only when the transaction is alive and are reclaimed as soon as the transaction is committed or resolved. Two main factors decide the total amount of RVM cost for persistent transaction data structures. The first factor is the duration of a disconnected operation session: longer sessions typically result in more live transactions. The second factor is the transaction usage pattern such as the number and size of live transactions.

We employ the same trace simulation experiment described in the previous Section (9.3.2.1) to measure the accumulated RVM cost over a sustained period of disconnected transaction operations. We also use controlled experiments to measure the RVM cost for individual transactions. The experiment workloads include software build and document build tasks of small, medium and large sizes. In addition, we execute three different *synrgen micro models* [12] as interactive transactions containing repeated editing and compiling activities.

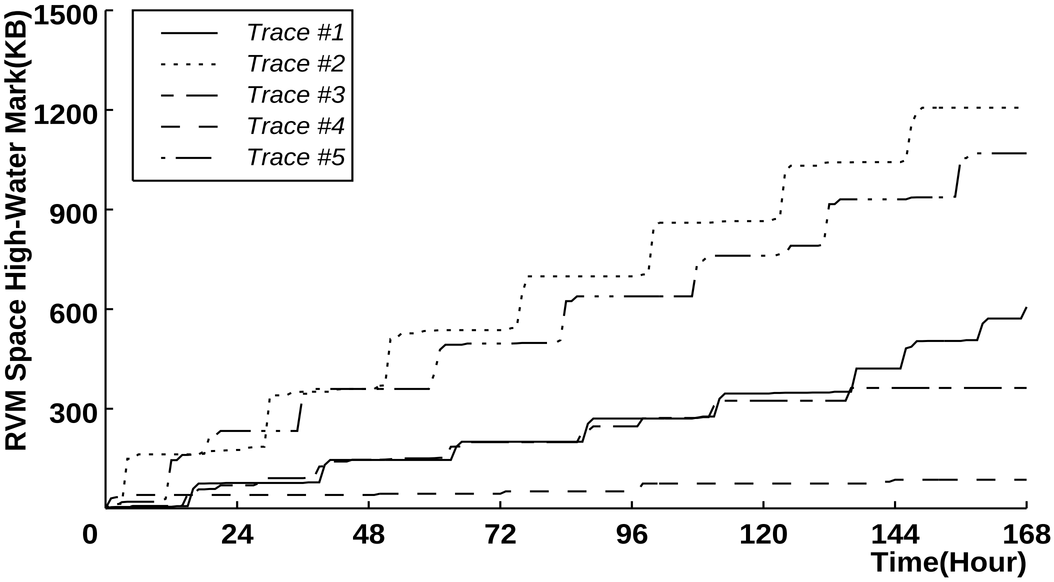
	RVM Space Cost(Byte)				
Application	Total	Readset	Writeset	Volume-set	Other
Latex Dissertation(252 pages)	3162	2340	612	96	114
Latex Proposal(54 pages)	1052	756	108	80	108
Latex Short Paper(6 pages)	830	540	108	80	102
Build Coda Venus	13278	8280	4794	96	108
Build Coda Server	7493	6012	1247	128	106
Build Repair Tool	2146	1584	340	112	110
Synrgen Codahacker	6071	2700	3205	48	118
Synrgen Programmer	5933	2664	3103	48	118
Synrgen Synrgenhacker	5641	2664	2808	48	121

This table shows the measured RVM space cost breakdown for executing common applications as transactions. The `Readset`, `Writeset` and `Volume-set` columns display the RVM cost for storing the transaction readset, writeset and the list of accessed volumes. The `Other` column contains the RVM cost for storing the conflict resolution option, process group id, and pathname of the working directory, etc. Note that the total cost here does not include the RVM space used to store the environment variable list, which is stored in a shared environment variable database and its average RVM space cost is about 2.4KB.

Table 9.17: RVM Cost for Common Transactions



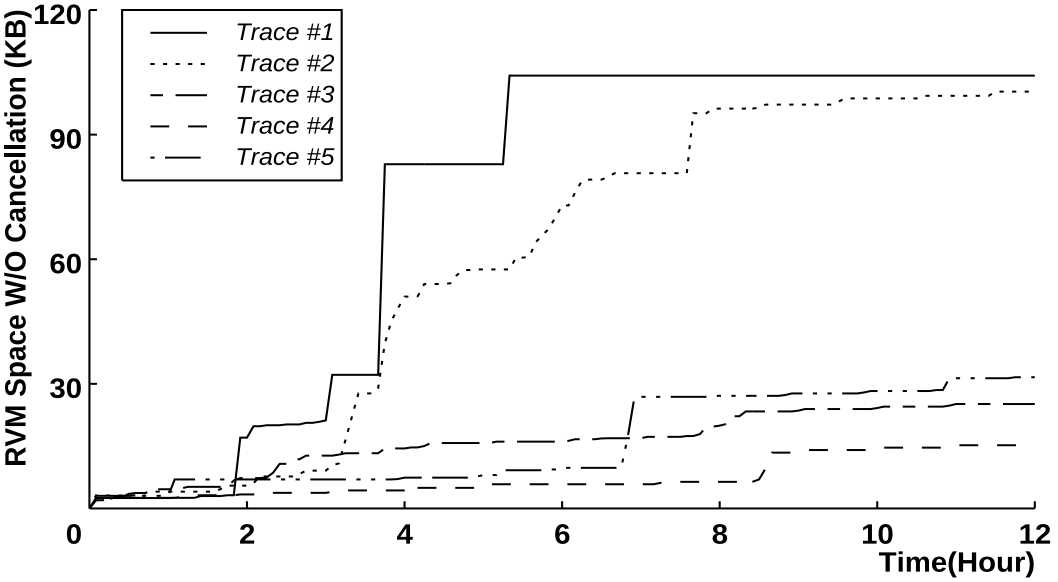
(a) Work-Day Traces



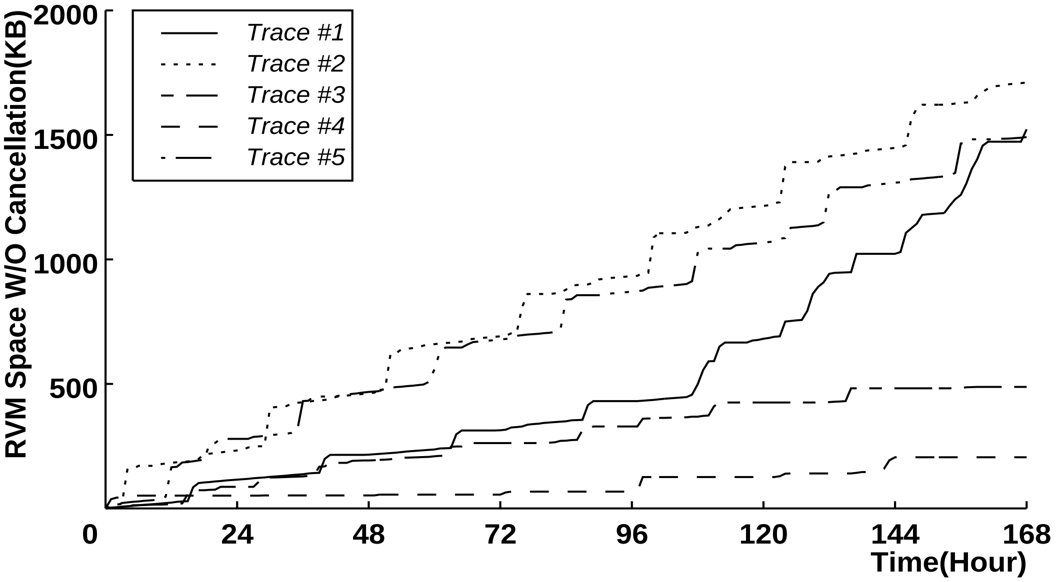
(b) Full-Week Traces

The two graphs in this figure show the high-water marks of RVM space cost recorded by the simulator for the five Work-Day and five Full-Week traces listed in Table 9.13. The simulator assumes that the environment variable list for individual transactions is stored in a shared environment database occupying 3KB RVM space.

Figure 9.9: High-Water Marks of RVM Space Cost



(a) Work-Day Traces



(b) Full-Week Traces

The two graphs in this figure show the high-water marks of recorded RVM space cost when the simulator does not perform transaction cancellation for the 10 traces listed in Table 9.13.

Figure 9.10: RVM Space Cost Without Transaction Cancellation

Results Similar to the discussion of shadow space cost, we present the measurement results on RVM space cost in three steps: (a) how much RVM space is needed for an extended period of disconnected transaction operation; (b) how much RVM space is needed for each individual transaction; (c) what the key factor is in keeping the RVM space cost low.

First, Figure 9.9 displays the high-water marks of RVM space cost recorded during trace simulation. The simulator assumes that the total RVM space cost for maintaining the shared environment variable database is 3KB, which is typical under normal circumstances. For supporting disconnected transaction operations in a Work-Day, the highest RVM space cost is about 100KB and the average RVM space cost is less than 50KB. For the longer duration of a Full-Week, the highest RVM space cost is about 1.2MB and the average cost is about half a MB. Once again, considering the length of the disconnected duration and the amount of transaction activity shown in Table 9.15, such RVM space costs are acceptable.

Second, we discuss the measurement results on RVM space cost for individual transactions from both trace simulation and controlled experiments. Note that the RVM space cost shown in Table 9.16 does not include the environment variable list since we store them in a shared environment variable database as described in Chapter 8. For most applications, the RVM space cost per transaction is quite small, from a couple hundred bytes to several KB. The only exception is the `find` transactions in the Work-Day traces where they read 1239.3 objects on average and cost about 46KB RVM space per transaction. Table 9.17 lists the RVM space cost for running typical applications as transactions ranging from 1KB to 13KB.

Third, transaction cancellation again plays an important role in keeping the accumulated RVM space cost low over an extended period of disconnected transaction operations, as shown in Figure 9.10.

9.3.2.3 Disk Space Cost for Conflict Representation

Methodology When a subtree is localized to represent the client state of an object in conflict, all the local objects inside the subtree still retain their container cache file. This results in additional disk space usage. Two factors decide the amount of disk space used by a local subtree: the number of files in the subtree and the sizes of those files. We employ a combination of empirical data from AFS and trace analysis to first estimate the number of files in a typical local subtree, and then estimate the corresponding disk space cost for maintaining that subtree.

Our first step in estimating the number of files in a typical local subtree is to use the result of a previous study on the distribution of various physical characteristics of AFS reported by Maria Ebling in [12] shown in Table 9.18. Although the study was done over a large number of AFS volumes, the strong similarity in file system structures and usage environments between Coda and AFS makes the extrapolation to Coda acceptable. Our second step is to obtain an estimation of the height of a typical local subtree using trace analysis and then deduce the

	Volume Type				
Physical Characteristic	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
File Size (KB)	10.3 (65.0)	24.0 (145.7)	16.4 (72.6)	2.6 (7.0)	19.1 (118.0)
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 9.18: File System Object Distribution

number of files in the subtree. The basic idea is that every local subtree corresponds to a conflict, and every conflict involves a partitioned mutation operation. If we know the typical location of an object that is updated by a mutation operation in the file name space, we can find out the corresponding height of the subtree rooted at that location. That subtree will be localized if the mutation results in a conflict.

We built an analysis tool that reads trace records and computes the distribution of the height of the subtrees rooted at the locations touched by directory mutation operations which occurred in the input trace. We restrict the trace analysis to only directory mutations because file mutations can only result in the trivial case of single node subtrees.

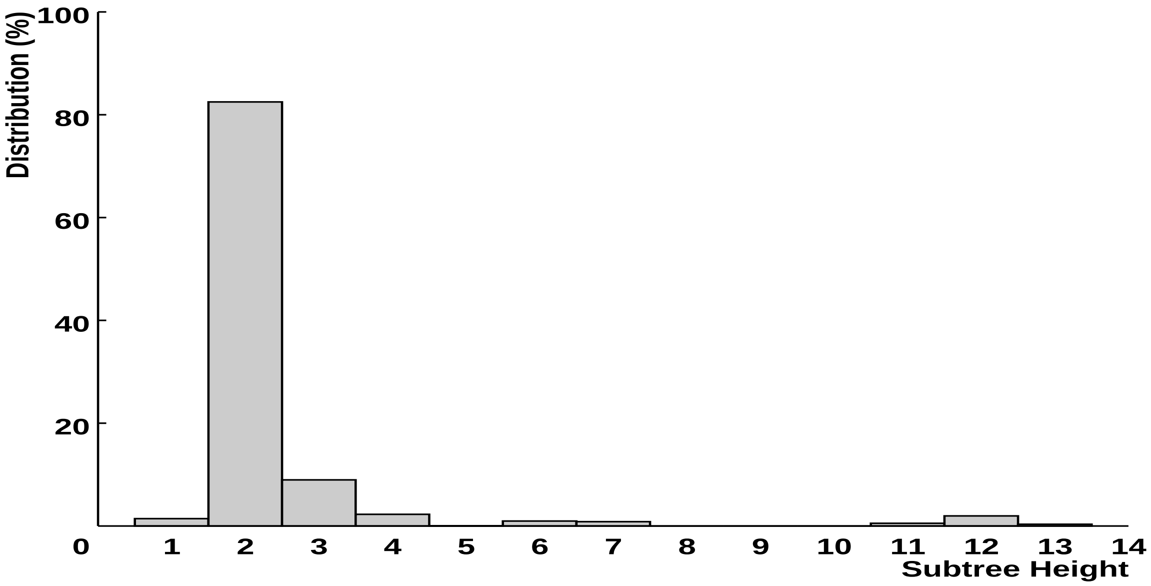
Results and Analysis We performed trace analysis on four of the five Full-Week traces listed in Table 9.13. As indicated by the results shown in Figure 9.11, most directory mutations are performed at the bottom levels of the file name space. A typical local subtree has only two levels, i.e., its height is two.

To obtain the relationship between the height of a subtree and the number of files in that subtree, all we need to do is to solve the following recurrences:

$$P(1) = 0 \tag{9.1}$$

$$P(H) = F + D * P(H-1) \tag{9.2}$$

Subtree	Subtree Height Distribution Over Directory Mutation(%)							
Height	Create	Link	Unlink	Mkdir	Rmdir	Symlink	Rename	Total
1	0.00	0.00	1.45	0.00	0.00	0.00	0.00	1.46
2	36.53	1.05	37.14	0.15	0.00	1.94	5.67	82.50
3	2.49	0.13	4.16	0.07	0.05	1.26	0.82	8.99
4	0.77	0.17	1.03	0.01	0.00	0.14	0.16	2.28
5	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.04
6	0.06	0.00	0.06	0.00	0.00	0.00	0.88	0.99
7	0.00	0.00	0.00	0.00	0.00	0.00	0.86	0.86
8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
11	0.13	0.13	0.27	0.00	0.00	0.00	0.00	0.53
12	0.67	0.32	0.98	0.00	0.00	0.00	0.00	1.98
13	0.03	0.03	0.31	0.00	0.00	0.00	0.00	0.36
total	40.69	1.83	45.40	0.24	0.07	3.36	8.41	100.00



The table in this figure shows the distribution of subtree height over different kinds of directory mutation operations. The result is obtained by using the trace analysis tool on four Full-Week traces listed in Table 9.13. Trace #3 is omitted because of a technical difficulty pertaining to a particular record in the trace. The graph in this figure plots the same distribution data displayed in the table.

Figure 9.11: Subtree Height Distribution

where $P(H)$ is the number of files in a subtree of height H, F is the number of files per directory and D is the number of directories per directory. The solution to the above equations is:

$$P(H) \;=\; F * (D^{H-1} - 1)/(D - 1)$$

Using the numbers from Table 9.18, we have:

$$P(H) \;=\; 15.7 * (3.2^{H-1} - 1)/2.2$$

Therefore, a typical local subtree of height 2 will have 16 files in it. Since the average file size is 19.1KB from Table 9.18, a typical subtree costs about 306KB, which is reasonably small.

9.3.2.4 RVM Space Cost for Conflict Representation

Methodology Local objects in conflict representation still need to be maintained in the cached object database, which must be stored in RVM. The total amount of RVM space cost for a local subtree depends on the number of objects in the subtree because each of them occupies roughly the same amount of RVM space. We use the same methodology used in section 9.3.2.3 to measure the RVM space cost for local subtrees.

Results and Analysis The relationship between the number of nodes and the height of a subtree is the solution of the following recurrences.

$$N(1) \;=\; 1 \tag{9.3}$$
$$N(H) \;=\; (1 + F + L + S) + D * N(H-1) \tag{9.4}$$

where N is the number of nodes; H is the subtree height; F is the number of files per directory; L is the number of links per directory; S is the number of symbolic links per directory and D is the number of directories per directory. The solution is the following formula:

$$N(H) \;=\; (D^{H} + (F + L + S) * D^{H-1} - F - L - S - 1)/(D - 1)$$

Using the numbers in Table 9.18, we have:

$$N(H) \quad = \quad (3.2^H + 25.4 * 3.2^{H-1} - 26.4)/2.2$$

According to this formula, a typical subtree with a height of 2 contains 30 nodes. Under the current Venus implementation, each node costs about 392 bytes of RVM space. Hence, a typical local subtree will cost around 11.7KB of RVM space. It is slightly higher than that of a typical transaction in Table 9.17, but still small.

9.3.3 Summary

Global Resource Cost Our evaluation based on both quantitative measurements and qualitative analysis indicates that transaction executions have limited impact on the overall usage of global system resources. By design, transaction validation has no long term effect on global system resource usage. Due to the write-through caching policy employed by the current Coda implementation, both the server load and network traffic can be increased or decreased as a result of connected transaction execution depending on the specific circumstances. There can be a slight increase in server load and network traffic caused by reintegrating disconnected transactions.

Local Resource Cost Trace driven simulation and controlled experiments offer convincing evidence that the local system resource cost associated with disconnected transaction usage for common applications is modest. Both disk space and RVM space costs are small relative to the normal capacity of today's portable computers. This is in large part due to the effectiveness of transaction cancellation.

Local resource cost for conflict representation is also affordable in typical situations and our anecdotal experience conforms with our evaluation results. In addition, conflicts are rare and often short lived (quickly resolved afterwards), thus conflict representation incurred resource costs are unlikely to pose serious problems in practice.

9.4 A Preliminary Usability Assessment

While the previous sections demonstrate the feasibility of the IOT model on the cost side, key usability issues remain unanswered. For example:

1. *How easy it is to program, specify and invoke a transaction? How easy it is to program an automatic resolver?*

2. *How often can conflicts be transparently and automatically resolved?*

3. *How effective is the transaction repair tool in helping users to manually resolve conflicts?*

Questions such as these cannot be answered until there is substantial actual system usage from a sizable user community. What is presented here is the best we can offer at the present time: a preliminary assessment of a few usability issues based on our limited usage experience.

9.4.1 Interactive Transaction Invocation

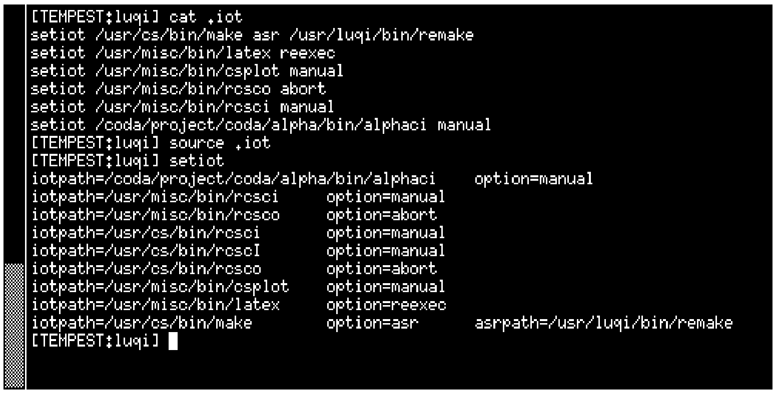


Figure 9.12: Examples for Transaction Specification
The window image in this figure shows the content of an IOT profile and the effect of sourcing it to specify applications such as `rcsci`, `latex` and `make` as transactions with various conflict resolution options.

Our experience indicates that the interactive interface of the IOT C-shell serves its intended purposes well. Specifying an application to be treated as a transaction as well as its resolution option is straightforward. Any existing Unix application can be invoked as a transaction with ease. This is largely due to the simplicity of the `setiot` and `unsetiot` commands, and their resemblance to the commonly used `setenv` and `unsetenv` shell commands. For example, transactions can be specified by setting a profile and sourcing it at the login time. Figure 9.12 shows the content of such an IOT profile and its effect on transaction specification.

We have experimented executing most of the common applications in our environment as transactions. Two areas of our implementation need further improvement. First, we should extend the current interface to better accommodate the traditional C-Shell idioms. Specifically, shell commands connected by pipelines or involving input/output re-directions need to be properly included in the scope of transaction executions. Second, we need to extend the C-Shell wild-card mechanism so that a group of applications can be specified as transactions together. For example, we may want to specify that all applications under `/usr/cs/bin` are to be treated as transactions by using the command of `setiot /usr/cs/bin/*`.

9.4.2 Programming A Transaction

We have used the IOT programming interface to put a transaction wrapper on several common applications such as `rcsco`, `make` and `latex`. Our experience confirms much of what was expected from the interface: straightforward programming and well-structured code adaptation. The most sophisticated use of the IOT programming interface so far has been in developing the IOT C-Shell. The `begin_iot` and `end_iot` calls are used to bracket the execution of any applications specified by *setiot* to be treated as transactions. There is some extra code dealing with failures and abnormal conditions. Overall, IOT related code is cleanly integrated with the rest of the C Shell source code.

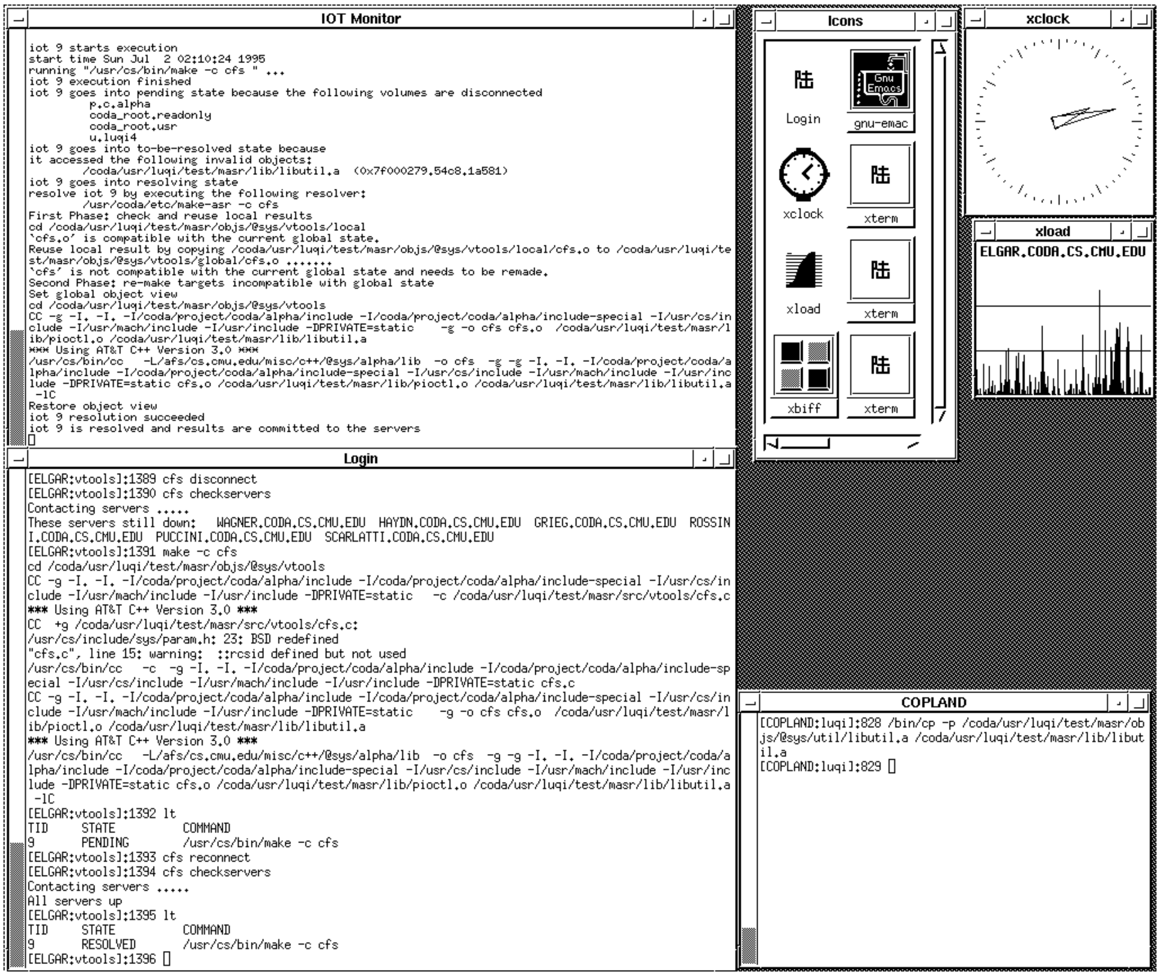
9.4.3 Resolver Development

Compared to programming a transaction, the programming of an application-specific resolver can be a demanding task. We conducted a number of case studies to write resolvers for some non-trivial and commonly-used Unix applications. Our purpose was to evaluate how well the task of programming resolvers was supported. In this section, we first present two resolver examples, and then summarize our resolver programming experience with some general observations.

9.4.3.1 Case Studies

A Smart Resolver for Make We developed a resolver for `make` based on the source code of the CMU Mach `make` to demonstrate that our application-specific resolver paradigm can indeed work for practical and sophisticated Unix applications. The main objective of this exercise is to investigate how application semantics can be employed to resolve conflicts more efficiently.

When a `make` transaction fails to certify, it means that some of the objects accessed by the transaction have been updated on the servers. As discussed in Chapter 3, an automatic re-execution of the same transaction under the original environment is guaranteed to restore data consistency as required by the semantics of `make`. But, re-execution is often wasteful, because it throws away local results even when most of them can be reused. For example,



The window image in this figure is taken from the console of *elgar*, a desktop Coda client. The `Login` window displays the main actions on elgar: disconnecting it from the servers, executing a `make` transaction that compiles a `cfs.o` file and links it with a library `libutil.a` and other libraries to create the `cfs` binary file. Before elgar reconnects to the servers, another connected client `COPLAND` installs a new version of library `libutil.a`, as shown by the *COPLAND* telnet window. Upon reconnection, the transaction system on elgar fails to certify the `make` transaction and automatically invokes the *smart-make* resolver. The `IOT Monitor` window shows the resolution actions where the local result of `cfs.o` is reused and the new version of `libutil.a` is linked to create an up-to-date version of `cfs`.

Figure 9.13: An Example of a Resolver for Make

suppose a disconnected `make` transaction compiles dozens of object files and builds a Venus binary. However, one of the libraries linked in was updated on the servers during the disconnection. Upon reconnection, we can reuse those locally compiled object files and re-link them with the new version of the library to produce an up-to-date Venus binary. This can avoid the unnecessary recompilations of the objects that would have been performed by automatic re-execution.

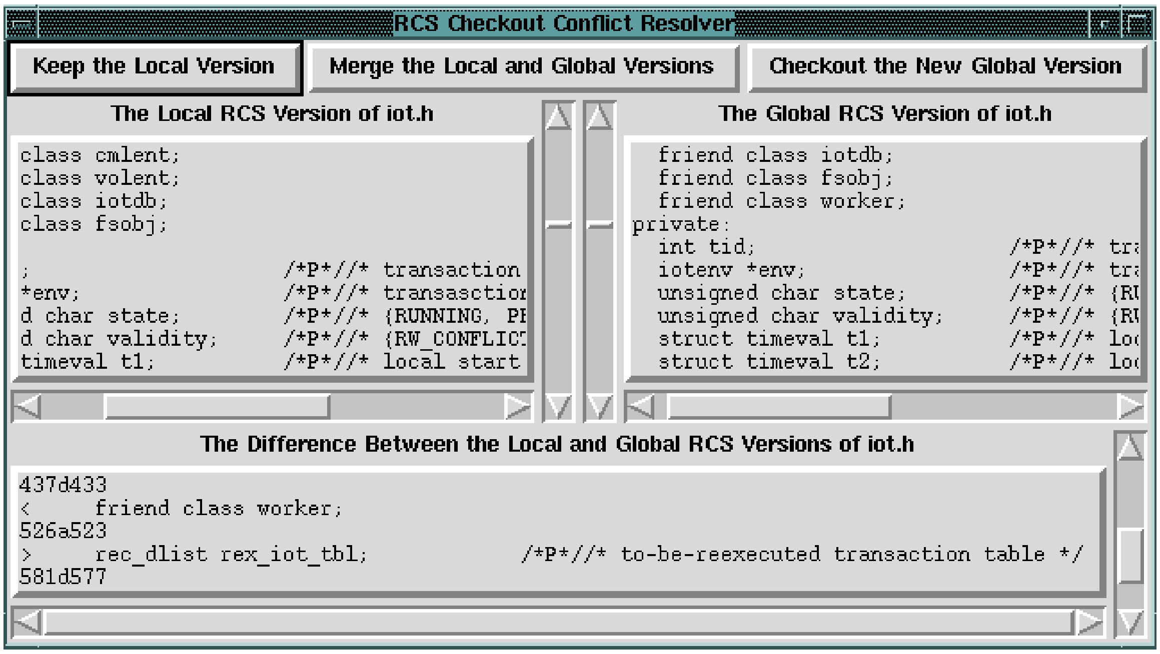
We developed a resolver for `make` called `smart-make`. Its first part is programmed based on both the original `make` source code and the resolution object view provided by IOT's dual-replica conflict representation. It extends the basic `make` dependency checking with the inspection of the local and global replicas for objects that are in conflict. This enables it to identify those local objects that were updated by the original `make` transaction and are still compatible (as required by the `make` dependency semantics) with the current global state. For each `make` target, the resolver first obtains its local replica, and then tracks the dependencies defined in the corresponding `makefile(s)` to find out all the objects that the target depends on. If any of them has different local and global replicas, the target is re-made. Otherwise, the local replica of the target can be reused and is copied onto the corresponding global replica.

The second part of `smart-make` uses the ASR programming library routines to set the global object view so that only the global state is visible to the resolver. Note that the visible global state after the first part may contain the reusable local results that just have been copied into the relevant global replicas. The resolver simply calls internal `make` functions to re-make the remaining targets and restore the original object view. When the resolver exits, the IOT system will take over and automatically commit the new results to the servers and discard all the previous local results. A complete example that illustrates how the `smart-make` program can efficiently resolve a `make` transaction is shown in Figure 9.13.

An Interactive Resolver for RCS Checkout For some Unix applications, there are often different alternatives to resolve a read/write conflict and the best strategy for an application-specific resolver is to interact with the user. Because such interactive resolvers need to display messages to the users and read input from them, they have to manage input/output as explained in Chapter 6. Figure 9.14 illustrates an interactive resolver we have developed for the RCS checkout command (`rcsco`). The resolver was developed using `tcl/tk` and written as a `wish` script.

9.4.3.2 General Observations

Programming a resolver can be quite difficult, particularly for complicated applications. Working knowledge of the source code of the original application is a must. We spent a significant amount of time in studying the `make` source code during the development of the `smart-make`



The window image in this figure demonstrates an interactive resolver for `rcsco`. The target transaction being resolved is an `rcsco` command that checked out an old version of `iot.h` during disconnection, which was later updated on the servers. The resolver is automatically invoked after the read/write conflict is detected. It displays the content of both the local and global versions of `iot.h` and the difference between them. It also provides the user three resolution options by simply clicking the corresponding button.

Figure 9.14: An Example of A Resolver for RCS Checkout

resolver. In general, the original author(s) or the site maintainer(s) of an application are the best candidates for writing the corresponding resolver.

The IOT resolution model is not easily understood at first sight. It usually requires some concrete examples for a novice IOT programmer to grasp the essence of the model and be able to code resolvers. The difficulty usually lies in figuring out what is detected by the IOT mechanism and what the file level conflict means to the application at a semantic level. However, once the user is able to code a simple resolver, it is not as difficult to apply the same principles to more complicated cases.

IOT's incremental resolution model indeed simplifies the programming of resolvers. It localizes the effect of conflicts and allows the resolver-writers to concentrate only on two sides, the current global state and the local actions of the transaction being resolved. The multiple view capability is also a plus, enabling the resolver to take advantage of the original application capabilities as demonstrated in the `smart-make` example.

Support for resolver programming could be further improved. Among other things, finding out what has been accessed (and updated) by the transaction being resolved can be difficult for some resolvers. A better facility such as iterators capable of enumerating all the elements in the transaction's readset and writeset will be helpful. More support is needed to help resolvers to find and compare local and global replicas of a given object.

9.5 Further Evaluation

So far, we have evaluated the performance overhead, resource cost and some usability issues of the IOT mechanism based on controlled experiments and case studies. Many other system usability issues are still unaddressed pending further accumulation of usage experience. Moreover, some of the previous quantitative measurement results need to be further strengthened or adjusted with more usage data. Here we describe the ongoing work and a brief plan of collecting IOT usage data and further evaluating the system when sufficient experience is obtained.

9.5.1 Data Collection

We have implemented a data collection mechanism for IOT usage that has been operational in the production release of Coda for half a year. It is based on the data collection machinery called `mond` implemented by Brian Noble [49]. Each running IOT-Venus sends two kinds of data to a `mond` server which puts the data into a log file. The `mond` server periodically ships the log file to a distributed relational database so that the collected data can be post-processed using SQL.

The first kind of IOT data is about a particular transaction: its application name, conflict resolution option, resource usage, and state transition history, etc. When a transaction completes its lifecycle, such a record is sent from the IOT-Venus to the `mond` server. The second kind of data is periodically transmitted from the IOT-Venus to the `mond` server. This data contains statistics such as maximum and average readset/writeset size and various information about localized subtrees and transaction repair sessions.

Other IOT usage data is also collected and persistently stored at the client. This includes information such as the high-water mark of shadow space cost and statistics about the serialization and wait-from graphs.

9.5.2 User Survey

Many usability issues are highly subjective. A reliable evaluation requires a scientific survey of IOT users with substantial experience. Such a survey will have to await the emergence of a sizable IOT user community. The survey will be used to examine those aspects of the IOT model and implementation with which users or programmers directly interact. It will need to ask the users to grade, characterize and comment on their perceptions about issues such as the effectiveness of the IOT support for resolver programming and manual conflict resolution. Combined with automatically collected data, this will enable a comprehensive usability study and shed more light on IOT.

Chapter 10

Related Work

To the best our knowledge, IOT is the first transaction model that is designed for the sole purpose of providing consistency support for partitioned file access operations, and specifically addresses the unique constraints and needs of mobile computing such as the resource limitation on a mobile client. Although the design and implementation of IOT builds upon existing techniques from areas such as transaction processing and optimistic replication, no other system offers the combination of properties that IOT possesses.

The first section of this chapter surveys important transaction systems and models that have had strong impact on the IOT model. The second section compares the conflict detection and resolution mechanisms between IOT and a few other optimistically replicated systems. Finally, several commercial products employing optimistic replication techniques are reviewed.

10.1 Transaction Models and Systems

10.1.1 General Purpose Transaction Systems

The basic transaction processing techniques were originally developed in the database community and have been successfully applied in commercial systems. Because of its ability to relieve programmers from complicated tasks of handling concurrent accesses to shared data and various failures in distributed systems, a number of research systems have been developed to explore transactions as a basic system construct as well as a general programming paradigm for building reliable distributed applications.

The Camelot system [13] provides a set of libraries for managing key aspects of transaction processing such as locking and logging so that applications can use them to construct distributed applications that have the ACID properties. While systems such as Argus [37, 36] and

Avalon [23] emphasize programming language support for transaction operations, the TABS system [66] supported transactional accesses to user defined abstract data types. A common objective behind these general purpose transaction systems was to facilitate the management of system resources and services that possess some or all of the ACID properties. The IOT model inherits this spirit, except that it specializes in isolation enforcement for partitioned file services.

10.1.2 Transaction Support for File Systems

QuickSilver QuickSilver [69, 64] is a distributed operating system that uses transactions as the basic tool to organize all system resource management tasks. Its distributed Unix file system provides failure atomicity, durability and a relaxed isolation guarantee. It shares IOT's pursuit of upward compatibility through an easy-to-use transaction interface that allow existing Unix applications to be executed as transactions without change. In addition, both consistency models fine tuned their guarantees for read-only transactions to minimize their performance impact. However, QuickSilver's transaction service does not support partitioned file access operations.

Locus The Locus distributed operating system [74] provides a general purpose nested transaction facility. Its main purpose is to support the construction of reliable distributed applications that are able to deal with complex system failures in a multi-machine environment. Locus uses a shadow page technique to support atomic file updates on all files and guarantees global atomicity via a two-phase commit protocol across all sites involved in a transaction. It employs both implicit and explicit record-level locking to perform concurrency control. Although Locus is one of the first to adopt optimistic replication, its transaction service is not intended to guard against data inconsistencies resulted from partitioned sharing.

10.1.3 Optimistic Concurrency Control

An important transaction processing model that has significant impact on IOT is Kung and Robinson's optimistic concurrency control model [33]. The OCC model and disconnected operation share the basic design philosophy of performing computations in a local scope and verifying their global validity later. The entire IOT execution model is designed based on OCC. OCC is directly adopted for cross-client concurrency control for connected transactions. The optimistic isolation enforcement for disconnected transactions is also OCC-like: consisting two main steps of local execution and global validation. The major differences are that the validation is delayed until the client is reconnected to the corresponding servers, and that there are more options to deal with an invalidated transaction than OCC's designated choice of automatic re-execution. In addition, the original goal of OCC was to reduce the performance overhead of concurrency control instead of optimistic replication.

10.1.4 Special Transaction Models

In the voluminous research literature about transaction processing techniques, there are a few specialized transaction models that share important common ground with IOT.

Saga: A Model of Long-Lived Transactions The Saga model [16] is an attempt to apply transaction models to long running applications. The key idea is to circumvent the strict serializability requirements by exposing the result of long running transactions and employing compensating actions to bring the database system into a consistent state. The strong similarity between Sagas and IOTs is their common belief that it is usually acceptable to expose tentative transaction results to other transactions as long as their validity can be established later. They also assume that data consistency can be effectively restored by using application provided compensating actions. However, Saga and IOT use different transaction execution models and consistency validation schemes. In addition, the Saga model does not deal with mobility issues and has not been implemented in an actual system.

Utilizing Transaction Semantics Garcia-Molina proposed to utilize application semantic knowledge for transaction processing in distributed database systems [15]. This approach shares IOT's design philosophy of integrating application semantics to serve useful purposes. However, it aims to obtain performance gains in concurrency control rather than better consistency maintenance. The model in [15] divides each individual transaction into steps and acquires application semantic information in the form of transaction types, compatibility sets and countersteps. Using such information, the transaction manager can produce more efficient schedules that are not necessarily serializable but still preserve data consistency.

10.2 Optimistically Replicated Systems

10.2.1 The Coda File System

In addition to disconnected operation, the Coda file system also employs optimistic server replication to enhance data availability. It uses the version vector technique to automatically detect partitioned write/write conflicts among server replicas, and separate mechanisms to resolve conflicts for directories and files. Note that Coda's earlier focus was on write/write conflicts and IOT is the first attempt to address the issue of partitioned read/write conflicts.

Log-Based Directory Resolution Coda uses logging to record partitioned directory mutation operations performed on server replicas [29]. Because directories are used as meta data for organizing file system structures, the semantics of directory mutation operations are known to the file system. Such semantics can be employed to automatically resolve many write/write conflicts such as a partitioned pair of `mkdir foo` and `mkdir bar`, as long as the names `foo` and `bar` are not used in the parent directory. The resolution protocol gathers all the mutation logs from the diverged replicas of a directory; deduces a compensation mutation log for each replica; replays compensation mutations at the corresponding replicas so that they become identical again. Actual experience has shown that most conflicts on directories can be automatically resolved and the log-based approach is very effective because it usually consumes only a small amount of server space.

Application-Specific File Resolution For write/write conflicts on the server replicas of file objects, Coda employs a rule-based framework for the users to supply application-specific resolvers to be automatically invoked [32]. Each resolver is associated with an individual file and the object/resolver binding is specified in a special rule file. The automatic resolver invocation is performed lazily, i.e., it is triggered by an attempt to access an inconsistent object.

This object-based ASR differs from IOT's ASR mechanism in resolver binding and resolver's knowledge of the relevant conflicts. Either approach has its pros and cons. The object-based ASR is more suitable for applications involving files whose consistency can be decided by its content alone without knowing the relevant partitioned computations that caused the conflict (e.g., the calendar program). IOT's ASR knows more information about the conflict and can be applied to more common applications.

Representation of Server/Server Conflicts IOT's dual-replica conflict representation scheme is based on Coda's original representation for write/write conflicts among server replicas. It uses the same in-place approach and each inconsistent object is dynamically converted into a directory containing all the accessible server replicas. In addition, dangling symbolic links are also used to prevent an inconsistent object from being accessed [29].

Original Representation of Local/Global Conflicts Coda's original representation for local/global conflicts adopted a different strategy than the one used for conflicts among server replicas. The local replica of an inconsistent object was represented outside of the Coda namespace. All the data in the local replicas were contained in a *closure file* stored on the client's local disk [26]. The global replica of the inconsistent object was visible at its original location. This approach has the advantage of simple implementation, but it is very difficult for automatic resolvers to access the local replica. The closure file is still retained in the current system to serve as a backup copy for local transaction results.

10.2.2 The Ficus File System

The Ficus distributed file system is a descendant of Locus and performs optimistic replication within a peer-to-peer system architecture [22, 21].

Conflict Resolution Ficus only detects and resolves partitioned write/write conflicts. There is no support for detecting and resolving read/write conflicts. In contrast to Coda's log-based approach, Ficus employs an inferential method to deduce and resolve conflicts on directories. Although it avoids the space cost for logging partitioned mutations, the resolution process becomes much more complicated and a two phase distributed protocol is needed to garbage collect removed objects, which negatively affects the overall system scalability. Similar to Coda, Ficus transparently invokes an application specific resolver when a file is detected to be in conflict [56]. It adopts a slightly different method for binding files with their corresponding resolvers. Resolvers are executed on the servers, making it more susceptible for security attacks. In addition, there is no transaction encapsulation for resolver execution.

Conflict Representation Ficus uses a different strategy for conflict representation. It puts the diverged replicas of an inconsistent object into a special per-volume directory called an *orphanage* and notifies the users about the conflict by sending an email. The limitations of this approach have already been discussed in Chapter 5.

10.2.3 The Bayou System

Bayou is a recent optimistically replicated storage system designed for a mobile computing environment that includes portable computers with less than ideal network connectivity [68].

Application-Specific Consistency Validation A unique feature of Bayou is its reliance on application-specific semantics for detecting and resolving write/write conflicts. For every update operation, the user must supply a routine called *dependency check* and it will be automatically executed on the servers so that application-specific knowledge can be employed to validate whether the current update is in conflict with the previous server state or not. Recall that IOT's ASR mechanism can also be employed to perform application-specific consistency validation. However, Bayou carries the design philosophy of utilizing application semantics to the extreme. Application-specific consistency validation is performed for every single update operation. In contrast, Coda's approach is to rely on the inexpensive version comparison for the common cases and to use the more expensive application-specific consistency validation only when the first approach fails.

Application-Specific Conflict Resolution In Bayou, the user must also provide a *merge procedure* for every update operation so that it can be automatically invoked to resolve conflict when the dependency check fails. This is similar to IOT's ASR mechanism because the application provided resolvers are associated with actions instead of objects. However, an action in Bayou is only one individual update operation, while it is the entire execution of an application in IOT. Because the current Bayou design requires full replication of an entire database on each host of the system in order to maintain full consistency, it is impractical for applications involving large or multiple databases.

10.2.4 Davidson's Optimistic Transaction Model

The optimistic transaction model proposed by Davidson established the theoretical foundation for transaction processing in optimistically replicated database systems [9]. The main purpose of the model is to guarantee that the effect of partitioned transaction executions is equivalent to a serial execution of the same set of transactions in a connected environment. The key technique is to establish a global data structure called the *precedence graph* based on recorded transaction histories. If the graph has cycles, it means that the partitioned transactions are not globally serializable and some of them must be backed out to restore data consistency.

This model provides the basis on which the IOT consistency model is built, although the current implementation only supports the more restrictive serialization criterion of global certification. However, there are some important differences between Davidson's model and IOT. First, the base semantics that Davidson's model is trying to protect from partitioned sharing is the traditional serializability-based isolation model instead of the shared memory Unix model. Second, Davidson's model is designed for a distributed database environment and pays no attention to constraints of mobility. Third, its only way of restoring consistency is through transaction back-out. Finally, there has been actual implementation of this model so far.

10.3 Commercial Products

Optimistic replication techniques have already found their way into several existing commercial software products to improve data availability.

10.3.1 Lotus Notes

Lotus Notes [25] is one of the first commercial products to embrace optimistic replication. `Notes` is based on a shared document database system that is designed to support a group of people working on shared documents in a personal computer network where the database servers are rarely connected. Group communication is accomplished primarily through adding documents to a shared database. Optimistic replication is valuable because typical `notes` databases such as an address book and software project bug reports are not heavily updated once the documents are placed in the database. In addition, group members sharing the documents do not need to see up-to-date data all the times. The replication algorithm of `notes` uses a one-way *pull model* [11] and guarantees eventual consistency of the documents in all replicas (i.e., changes made to one copy eventually migrate to all). Because of its intended purposes and operating environment, `notes` only promises to detect partitioned write/write conflicts and provides no support for dealing with partitioned read/write conflicts.

10.3.2 Oracle Server Replication

A recent release of Oracle database servers employs optimistic read-only and *symmetric* (read-write) replication to improve data availability [51]. A complete copy or a *snapshot* (partial copy) of a database table can be replicated at different sites. Read-only replicas are automatically refreshed to reflect the new updates based on the intervals specified by the users. Updates to read-write replicas are first performed at the local site and then propagated to the other sites via the *deferred transaction* mechanism either periodically (at the intervals specified by the users) or at specific points in time when connectivity is available or the communication costs are cheap. Similar to the incremental propagation scheme of the IOT model, deferred transactions are re-applied one-by-one at the remote site. The difference is that the consistency validation for deferred transactions relies on value certification and it is only performed for objects in the transaction's writeset. If a partitioned write/write conflict is detected, either a pre-defined conflict resolution procedure or a user specified resolver is automatically executed, with manual resolution as the last resort. Although Oracle's optimistic replication mechanisms adopt many measures similar to Coda, it does not have the capability to detect and resolve partitioned read/write conflicts, which could be quite important for database applications. In addition, the design and implementation do not pay attention to the constraints of mobility.

Chapter 11

Conclusion

This dissertation has described IOT, an explicit transactional extension to the Unix file system for safeguarding data consistency in mobile file access. The central idea is optimistic enforcement of serializability-based isolation requirements for partitioned transaction execution, aided by flexible conflict resolution mechanisms and integration of application-specific semantics in not only conflict resolution but also consistency validation. Adopting OCC as the underlying implementation framework is critical to the successful realization of the IOT model, where the key insight is recognizing that the disk cache of a mobile client can serve as the private workspace for optimistic transaction processing under unpredictably changing system connectivities.

The design, implementation, experimentation and evaluation of a working IOT extension to the Coda file systems enables us to reach the following conclusions.

- The IOT model is a feasible way of addressing the data inconsistency problems caused by partitioned read/write conflicts in mobile file access.
- Its practicality is demonstrated by its ability to maintain upward Unix compatibility and the ample evidence indicating that the support of IOT only incurs modest performance overhead and low resource costs.
- Initial evidence from controlled experiments shows the effectiveness of IOT's conflict resolution mechanisms and the conflict representation scheme in supporting application-specific resolvers for common Unix applications. It also demonstrates the ease of use of the IOT interfaces for transaction programming, specification and invocation.
- Due to the lack of usage experience, definitive conclusion on the usability of IOT needs to be deferred until substantial usage data is accumulated.

The actual realization of the IOT model is much more complicated than anticipated. This can be attributed to the inherent difficulty of maintaining and propagating tentative transaction results under unpredictable connectivities in a mobile environment. The design of incremental transaction propagation and in-place conflict representation strives to simplify the process of connectivity transitions and transaction resolutions. But it overloads the client with multiple duties of maintaining the results of uncommitted local transactions, reflecting the changing global server state, and representing resolution object views to resolvers. Therefore, the design trade-off for a clean model of client/server state synchronization and transaction resolution is not only additional computation and resource costs but also significantly increased complexity and its associated system development and maintenance costs. Such design knowledge could not have been obtained without the complete process of engineering an actual IOT implementation in the Coda file system, and is one of the most important findings of this research.

11.1 Contributions

To the best of our knowledge, this research is the first attempt to develop a practical file system facility to address the data inconsistency problems caused by partitioned read/write conflicts; and IOT is the first transaction model designed solely for the purpose of improving data consistency in mobile file access. Centered around those two aspects: the specific contributions of this thesis research can be classified into the following four areas:

1. *Conceptual Analysis*
   - A study of data inconsistencies caused by partitioned read/write conflicts within the context of a shared-memory consistency model.
   - An analysis of the role of a serializability-based consistency model in detecting data inconsistencies resulting from partitioned read/write sharing.
2. *System Design*
   - The design of a new abstraction, the IOT model, that balances three distinct criteria: guarding against data inconsistencies in partitioned file access, maintaining upward Unix compatibility, and incurring only modest performance and resource costs.
   - The smooth integration of application-specific knowledge into the IOT model for both purposes of conflict resolution and consistency validation.
   - A concise conflict representation scheme that provides resolvers with convenient access to information relevant to resolving an invalidated transaction.

- An incremental transaction propagation model that simplifies the tasks of resolver programming and manual conflict resolution.

3. *Implementation*

- A successful combination of optimistic concurrency control across clients with strict local two phase locking for transaction processing under various system connectivities.
- A safe and robust resolver invocation mechanism that supports both automatic transaction re-execution and automatic execution of application-specific resolvers.
- An interactive transaction interface in the form of a special C-Shell that enables convenient transaction specification and invocation.
- A model of identifying and cancelling redundant transactions during disconnected operation.

4. *Experiment and Evaluation*

- Empirical measurements based on controlled experiments, trace simulation and trace analysis that confirm IOT's modest performance and resource costs.
- The development of application-specific resolvers for commonly used Unix applications.
- The demonstration of the effectiveness of transaction cancellation in reducing client space cost for long-lasting disconnected operation sessions.

11.2 Future Work

For implementation expedience, a few minor features logically belonging to the current IOT model have not yet been fully supported. A number of implementation enhancements have been suggested in previous chapters. For example, Chapter 8 outlined a strategy for achieving full atomicity for transaction validation and commitment. This section describes additional implementation extensions that will make the current IOT service in Coda more complete. In addition, two areas worth further investigation are discussed: generalizing the IOT model to other system environments and providing support for the development of application-specific resolvers.

11.2.1 Implementation Extensions

Providing the G1SR Consistency Guarantee Recall that the IOT consistency model described in Chapter 3 contains two basic consistency criteria for validating disconnected transactions, namely G1SR and GC. The current IOT implementation in Coda only provides the GC consistency guarantee. However, for tasks involving tight sharing and frequent concurrent accesses to shared data from different clients such as the traditional database applications, the G1SR consistency guarantee may be more suitable for safeguarding their data integrity.

The specific design issues of implementing G1SR have been presented in an earlier document [39]. The key is using Davidson's optimistic database transaction model [9] where the servers maintain a transaction history and build a global precedence graph when pending transactions need to be propagated. Theoretically, the testing of G1SR can be achieved by simply checking whether the precedence graph is acyclic or not. In practice, however, there are many challenging design issues that require further investigation. These include reducing server space cost for maintaining a global transaction history, performing automatic resolution actions, and handling server and partition failures during the G1SR validation process.

Supporting Resolver I/O The resolution of transactions containing interactive I/O operations often requires communicating with the users through interactive I/O. The current IOT implementation requires the resolvers to manage their own standard I/O environment. Providing a library of window-based I/O operations would significantly simplify the writing of interactive resolvers.

11.2.2 Model Generalization

The current IOT model is designed to support mobile file access in a networked Unix workstation environment. It also does not deal with server replication for the sake of simplicity. However, the basic principles behind IOT can be applied to a more general setting.

Server Replication As currently designed, the IOT model only supports disconnected operation (i.e., optimistic second class replication). However, the IOT model can be generalized to provide consistency support for optimistic server replication (i.e., first class replication). There are two alternatives to extending the current transaction propagation model to deal with multiple servers that are potentially partitioned. The first approach is to superimpose a pessimistic replication scheme on server replicas before transaction propagation. In other words, the transaction system will propagate the result of a tentative transaction to the servers only when a majority of the replicas of the involved objects are accessible. This method inherits much of the original IOT model and greatly simplifies the process of transaction validation and resolution. The second approach is to aggressively propagate transactions to any replicas that are currently accessible. This would require a complicated transaction propagation protocol such as the anti-entropy model used in the Bayou system [68] to guarantee eventual mutual consistency among replicas. Although this approach offers better data availability, it adds considerable complexity to the transaction model and makes automatic conflict resolution more difficult.

Database Environment The IOT model can be applied to guard against data inconsistencies caused by partitioned sharing in optimistically replicated database systems. The basic principle of imposing serializability-based requirements for partitioned transaction executions naturally addresses the consistency needs of mobile access in database systems for the following reasons. First, transactions are already an inherent part of the data access model of typical database systems. Therefore, there is no need for API extension and maintaining upward compatibility. Second, serializability-based requirements are commonly adopted as the consistency model for interleaved transaction executions in database systems. Therefore, they can be uniformly employed for both executing connected transactions and validating disconnected transactions.

Many important aspects of the IOT model are likely to remain effective in a database environment, such as the conflict resolution options of automatic transaction re-execution and automatic invocation of application-specific resolvers. However, some design decisions and implementation strategies need to be adapted in order to apply the IOT model to database systems. For example, the consistency validation criterion for disconnected transactions needs to be changed from GC to G1SR because most database systems use 1SR as their consistency model. In addition, typical database applications involve frequent data sharing among different users performing concurrent accesses to shared data, making G1SR more suitable than GC. As another example, OCC may no longer be an appropriate concurrency control algorithm for connected transaction executions because heavy cross-client data sharing will result in frequent transaction re-executions required by OCC, which leads to excessive performance overhead.

Non-Unix File Systems The basic IOT model should be applicable to other non-Unix file systems such as those used in Windows 95, Windows NT and DOS. Due to the differences in system usage environments and application paradigms, important aspects of the IOT model such as consistency validation criterion and conflict resolution options may need to be changed accordingly. In addition, specific mechanisms such as OCC and 2PL used for concurrency control may also need to be modified to adjust to the new environment. However, the basic principle of optimistic enforcement of serializability-based isolation requirements for partitioned file access operations can remain intact. In addition, the hierarchical file system structure would allow the current conflict representation scheme to be largely retained.

11.2.3 Resolver Development

Application-specific resolution plays a key role in the IOT model. However, there is a lack of support for the programming of application-specific resolvers. Although we have successfully developed experimental resolvers for commonly used Unix applications such as `make`, our approach has been ad-hoc. Generally speaking, the research on how to best support resolver development is still in its infancy. Fundamental issues such as the semantic model and logical framework of resolvers are yet to be understood. Basic mechanisms such as secure and reliable resolver invocation and supporting resolver I/O require further investigation. We discuss the following two specific areas of future research on resolver development.

Programming Methodology There is a need to gain considerably more experience in developing resolvers for a large number of applications spanning a wide variety of application domains. Only when sufficient empirical experience is accumulated, will we be able to classify common resolver architectures and relate them to application characteristics. A programming methodology for resolvers will contain a set of basic guidelines for writing a resolver based on application characteristics. Because conflict resolution requirements for applications in different domains can be very different, it is likely that such a programming methodology will be domain-dependent. This means that applications need to be classified into domains and resolver development for a particular application needs to follow a set of domain-specific guidelines.

Development Tool Resolver programming is fundamentally different from normal application programming. First, a resolver is invoked only under special conditions. Second, the specific missions of a resolver depend very much on the dynamic system state on the client and accessible servers. Third, a resolver must be able to handle a wide range of situations involving partitioned data sharing. As a result, testing resolvers is much more difficult than testing applications. To facilitate resolver development, it would help to provide a development tool that assists the users to test resolvers. For example, the tool could allow developers to supply specifications about the target application and its resolver, and automatically generate test cases for the resolver. Of course, the development tool must also establish the proper system environment for invoking the resolver, and present the resolver execution outcome for inspection.

11.3 Final Remarks

This dissertation has shown that it is practical to use a transactional extension to the Unix file system for improving the consistency of accessing shared data on a disconnected client in a mobile environment. Two basic ideas made it possible: imposing serializability-based isolation requirements for partitioned transaction execution and adopting OCC as the underlying transaction processing framework under changing system connectivities. In addition, the integration of application-specific semantics via pre-programmed resolvers proves to be invaluable in consistency maintenance for mobile file access.

Enabling isolated components of a distributed computing system to operate autonomously will become increasingly desirable due to growing system size and component mobility. The successful introduction of disconnected file service four years ago was a milestone in this trend. This dissertation represents another significant step because it addresses a major limitation of disconnected file service.

Bibliography

[1] 4.3 BERKELEY SOFTWARE DISTRIBUTION. *UNIX Programmer's Reference Manual*, 1986.

[2] AGRAWAL, D. The performance of protocols based on locks with ordered sharing. *IEEE Transactions on Knowledge and Data Engineering 6*, 5 (1994).

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

[4] BARGHOUTI, N., AND KAISER, G. Concurrency control in advanced database applications. *ACM Computing Surveys 23*, 3 (1991).

[5] BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] DAN, A., TOWSLEY, D., AND KOHLER, W. Modeling the effects of data and resource contention on the performance of optimistic concurrency control protocols. In *Proceedings of the 4th International Conference on Data Engineering* (1988).

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

[8] DAVIDSON, S. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems 9*, 3 (September 1984).

[9] DAVIDSON, S., GARCIA-MOLINA, H., AND SKEEN, D. Consistency in partitioned networks. *ACM Computing Surveys 17*, 3 (September 1985).

[10] DELLAFERA, A., EICHIN, M., FRENCH, R., JEDLINSKY, D., KOHL, J., AND SOMMERFELD, W. The Zephyr notification service. In *Proceedings of the 1988 USENIX Winter Conference* (1988).

[11] DEMERS, A., GREENE, D., HAUSE, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D., AND TERRY, D. Epidemic algorithms for replicated database maintenance. *ACM Operating Systems Review* (January 1988).

[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. *Guide to the Camelot Distributed Transaction Facility including the Avalon Language*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[14] ESWARAN, K., GRAY, J., LORIE, R., AND TRAIGER, I. The notions of consistency and predicate locks in a distributed database system. *Communications of the ACM 19*, 11 (1976).

[15] GARCIA-MOLINA, H. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems 8*, 2 (1983).

[16] GARCIA-MOLINA, H., AND SALEM, K. Sagas. In *Proceedings of ACM SIGMOD Conference* (May 1987).

[17] GARCIA-MOLINA, H., AND WIEDERHOLD, G. Read-only transactions in a distributed database. *ACM Transactions on Database Systems 7*, 2 (June 1982).

[18] GIFFORD, D. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles* (August 1979).

[19] GRAY, J. Notes on database operating systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[20] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1993.

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

[22] 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).

[23] HERLIHY, M. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems 4*, 1 (February 1986).

[24] 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).

[25] KAWELL, L., BECKHARDT, S., HALVORSEN, T., AND OZZIE, R. Replicated document management in a group communication system. In *Groupware: Software for Computer-Supported Cooperative Work*. IEEE Computer Society Press, 1992.

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

[27] KISTLER, J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[28] KLEIMAN, S. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Summer Usenix Conference Proceedings* (June 1986).

[29] KUMAR, P. *Mitigating the Effects of Optimistic Replication in a Distributed File System.* PhD thesis, Carnegie Mellon University, Pittsburgh, December 1994.

[30] KUMAR, P., AND SATYANARAYANAN, M. Log-based directory resolution in the Coda file system. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems* (January 1993).

[31] 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 Systems* (Napa, CA, October 1993).

[32] KUMAR, P., AND SATYANARAYANAN, M. Flexible and safe resolution of file conflicts. In *Proceedings of the Winter Usenix Conference* (New Orlean, LA, January 1995).

[33] KUNG, H., AND ROBINSON, J. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6*, 2 (1981).

[34] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM 21*, 7 (July 1978).

[35] LEVY, E., AND SILBERSCHATZ, A. Distributed file systems: Concepts and examples. *ACM Computing Surveys 22*, 4 (1990).

[36] LISKOV, B., DAY, M., HERLIHY, M., JOHNSON, P., LEAVENS, G., SCHEIFLER, R., AND WEIHL, W. Argus reference manual. Tech. Rep. Technical Report-400, MIT Laboratory for Computer Science, November 1987.

[37] LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems 5* (July 1983).

[38] LORIE, R. Physical integrity in a large segmented database. *ACM Transactions on Database Systems 2*, 2 (1977).

[39] LU, Q. Isolation-only transactions in distributed Unix file systems. Thesis proposal, Carnegie Mellon University School of Computer Science, May 1993.

[40] LU, Q., AND SATYANARAYANAN, M. Isolation-only transactions for mobile computing. *ACM Operating Systems Review* (April 1994).

[41] LU, Q., AND SATYANARAYANAN, M. Improving data consistency in mobile computing using isolation-only transactions. In *Proceedings of the 5th Hot Topics in Operating Systems* (Orcas Island, WA, May 1995).

[42] MARTIN, B., AND PEDERSEN, C. Long-lived concurrent activities. Tech. Rep. HPL-90-178, HP Laboratories, 1990.

[43] MINOURA, T., AND WIEDERHOLD, G. Resilient extended true-copy token scheme for a distributed database system. In *IEEE Transactions on Software Engineering* (May 1982).

[44] MORRIS, J. H., SATYANARAYANAN, M., CONNER, M., HOWARD, J., ROSENTHAL, D., AND SMITH, F. Andrew: A distributed personal computing environment. *Communications of the ACM 29*, 3 (March 1986).

[45] MUMMERT, L., EBLING, M., AND SATYANARAYANAN, M. Exploiting weak connectivity in mobile file access. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995).

[46] MUMMERT, L., AND SATYANARAYANAN, M. Long term distributed file reference tracing : Implementation and experience. Tech. Rep. CMU-CS-94-213, Carnegie Mellon University School of Computer Science, 1994.

[47] MURANAGA, T., LU, Q., AND SATYANARAYANAN, M. Supporting cooperative work in a mobile distributed file system using isolation-only transactions. Carnegie Mellon University School of Computer Science, manuscript in preparation, 1996.

[48] NELSON, M., WELCH, B., AND OUSTERHOUT, J. Caching in the Sprite network file system. *ACM Transactions on Computer Systems 6*, 1 (1987).

[49] NOBLE, B., AND SATYANARAYANAN, M. An empirical study of a highly available file system. In *Proceedings for the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, TN, May 1994).

[50] NOVELL CORPORATION. *NetWare User Manual*, 1993.

[51] ORACLE CORPORATION. *Oracle7 Server Distributed Systems*, 1995.

[52] OUSTERHOUT, J. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[53] 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).

[54] PAUSCH, R. *Adding Input and Output to the Transaction Model*. PhD thesis, Carnegie Mellon University School of Computer Science, 1988.

[55] RAHM, E., AND THOMASIAN, A. Distributed optimistic concurrency control for high performance transaction processing. In *PARBASE-90 International Conference on Database, Parallel Architectures and Their Applications* (1990).

[56] 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).

[57] ROSENTHAL, D. Evolving the Vnode interface. In *Proceedings of the Summer Usenix Conference* (June 1990).

[58] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and implementation of the Sun network file system. In *Summer Usenix Conference Proceedings* (June 1985).

[59] SATYANARAYANAN, M. Scalable, secure, and highly available distributed file access. *Computer 23*, 5 (May 1990).

[60] SATYANARAYANAN, M., EBLING, M., AND RAIFF, J. *Coda File System: User and System Administrator's Manual*. Carnegie Mellon University School of Computer Science, December 1995.

[61] 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).

[62] SATYANARAYANAN, M., KISTLER, J., MUMMERT, L., EBLING, M., KUMAR, P., AND LU, Q. Experience with disconnected operation in a mobile environment. In *Proceedings of USENIX Symposium on Mobile Location-Independent Computing* (Cambridge, Massachusetts, August 1993).

[63] 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).

[64] SCHMUCK, F., AND WYLLIE, J. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Monterey, CA, October 1991).

[65] SHAFER, S. *The SUP Software Upgrade Protocol User Manual*. Carnegie Mellon University School of Computer Science, August 1990.

[66] SPECTOR, A., DANIELS, D., DUCHAMP, D., EPPINGER, J., AND PAUSCH, R. Distributed transactions for reliable systems. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA, Decemeber 1985).

[67] STEERE, D., KISTLER, J., AND SATYANARAYANAN, M. Efficient user-level file cache management on the Sun Vnode interface. In *Proceedings of the Summer Usenix Conference* (June 1990).

[68] TERRY, D., THEIMER, M., PETERSEN, K., DEMERS, A., SPREITZER, M., AND HAUSER, C. Managing update conflicts in a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, December 1995).

[69] THEIMER, M., CABRERA, F., AND WYLLIE, J. Quicksilver: Support for access to data in large, geographically dispersed systems. In *9th International Conference on Distributed Computing Systems* (1989).

[70] THOMASIAN, A., AND RAHM, E. A new distributed optimistic concurrency control method and a comparison of its performance with two-phase locking. Tech. Rep. IBMC 15073, IBM Watson Research Center, 1989.

[71] THOMASIAN, A., AND RYU, I. Analysis of some optimistic concurrency control schemes based on certification. In *Proceedings of the 1985 SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1985).

[72] TRAGER, I., GRAY, J., GALTIERI, C., AND LINDSAY, B. Transactions and consistency in distributed database systems. *ACM Transactions on Database Systems 7*, 3 (1982).

[73] TYGAR, D., AND YEE, B. Strongbox: A system for self securing programs. In *CMU Computer Science: 25th Anniversary Commemorative*. Addison-Wesley, 1991.

[74] 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).

[75] YU, P., AND DIAS, D. Notes on modeling optimistic concurrency control schemes. Tech. Rep. IBMC 14825, IBM Watson Research Center, 1989.