Experience with Disconnected Operation in a Mobile Computing Environment

M. Satyanarayanan, James J. Kistler, Lily B. Mummert,
Maria R. Ebling, Puneet Kumar, Qi Lu

June 1993
CMU-CS-93-168

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in
Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing, Cambridge, MA, August 1993

Abstract

In this paper we present qualitative and quantitative data on file access in a mobile computing environment. This information is based on actual usage experience with the Coda File System over a period of about two years. Our experience confirms the viability and effectiveness of *disconnected operation.* It also exposes certain deficiencies of the current implementation of Coda, and identifies new functionality that would enhance its usefulness for mobile computing. The paper concludes with a description of what we are doing to address these issues.

This work was supported by the Advanced Research Projects Agency (Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division(AFSC), U.S. Air Force, Wright-Patterson AFB under Contract F33615-90-C-1465, Arpa Order No. 7597), the National Science Foundation (Grant ECD 8907068), IBM, Digital Equipment Corporation, and Bellcore. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either express or implied, of the U.S. Government, IBM Corporation, Digital Equipment Corporation, or Bellcore.

Keywords: Disconnected operation, Coda File System, mobile computing, wireless communication, performance evaluation, hoarding, reintegration, file reference traces, weak connectivity, conflict resolution, transactions

1. Introduction

Portable computers are commonplace today. In conjunction with high- and low-bandwidth cordless networking technology, such computers will soon provide a pervasive hardware base for *mobile computing*. A key requirement of this new world of computing will be the ability to access critical data regardless of location. Data from shared file systems must be made available to programs running on mobile computers. But mobility poses serious impediments to meeting this requirement.

We begin this paper by describing how shared file access is complicated by the constraints of mobile computing. We then show how the design of the *Coda File System* addresses these constraints. The bulk of the paper focuses on our usage experience with Coda. We present qualitative and quantitative data that shed light on Coda's design choices. Based on our experience, we have identified a number of ways in which Coda could be improved. The paper concludes with a description of our current work along these dimensions.

2. Constraints of Mobile Computing

Access to shared data in a mobile environment is complicated by three fundamental constraints. These constraints are intrinsic to mobility, and are not just artifacts of current technology:

- *Mobile elements are resource-poor relative to static elements.* For a given cost and level of technology, mobile elements are slower and have less memory and disk space than static elements. Weight, power, and size constraints will always conspire to preserve this inequity.
- *Mobile elements are more prone to loss, destruction, and subversion than static elements.* A Wall Street stockbroker is more likely to be mugged on the streets of Manhattan and have his or her laptop stolen than to have the workstation in a locked office be physically subverted. Even if security isn't a problem, portable computers are more vulnerable to loss or damage.
- *Mobile elements must operate under a much broader range of networking conditions.* A desktop workstation can typically rely on LAN or WAN connectivity. A laptop in a hotel room may only have modem or ISDN connectivity. Outdoors, a laptop with a cellular modem may find itself in intermittent contact with its nearest cell.

These constraints violate many of the assumptions upon which today's distributed systems are based. Further, the ubiquity of portable computers will result in mobile computing systems that are much larger than the distributed systems of today. *Scalability* will thus be a continuing concern.

Ideally, mobility should be completely *transparent* to users. Transparency relieves users of the need to be constantly aware of the details of their computing environment, thus allowing them to focus on the real tasks at hand. The adaptation necessary to cope with the changing environment should be initiated by the system rather than by users. Of course, perfect transparency is an unattainable ideal. But that should not deter us from exploring techniques that enable us to come as close as possible to the ideal.

3. Overview of Coda File System

Coda, a descendant of the Andrew File System [4], offers continued access to data in the face of server and network failures. Earlier papers [7, 9, 14, 15, 16, 17] have described various aspects of Coda in depth. Here we only provide enough detail to make the rest of the paper comprehensible.

Coda is designed for an environment consisting of a large collection of untrusted Unix[1] clients and a much smaller number of trusted Unix file servers. The design is optimized for the access and sharing patterns typical of academic and research environments. It is specifically not intended for applications such as online transaction processing applications that exhibit highly concurrent, fine granularity update patterns.

Each Coda client has a local disk and can communicate with the servers over a high bandwidth network. Clients view Coda as a single, location-transparent shared Unix file system. The Coda namespace is mapped to individual

[1]Unix is a trademark of Unix System Laboratories.

file servers at the granularity of subtrees called *volumes.* At each client, a *cache manager (Venus)* dynamically obtains and caches data as well as volume mappings.

Coda uses two distinct, but complementary, mechanisms to achieve high availability. Both mechanisms rely on an *optimistic replica control* strategy. This offers the highest degree of availability, since data can be updated in any network partition. The system ensures detection and confinement of conflicting updates after their occurence, and provides mechanisms to help users recover from such conflicts.

3.1. Server Replication

The first high-availability mechanism, *server replication,* allows volumes to have read-write replicas at more than one server. The set of replication sites for a volume is its *volume storage group (VSG).* The subset of a VSG that is currently accessible is a client's *accessible VSG (AVSG).* The performance cost of server replication is kept low by callback-based caching [6] at clients, and through the use of parallel access protocols. Modifications at a Coda client are propagated in parallel to all AVSG sites, and eventually to missing VSG sites.

3.2. Disconnected Operation

Although server replication is an important part of Coda, it is the second high-availability mechanism, *disconnected operation,* that is a key enabling technology for mobile computing [8]. A client becomes disconnected with respect to a volume when no server in its VSG is accessible. An *involuntary disconnection* can occur in a mobile computing environment when there is a temporary impediment to communication. This can be caused by limitations such as short range, inability to operate underground and in steel-framed buildings, or line-of-sight constraints. A *voluntary disconnection* can occur when a user deliberately operates isolated from a network. This may happen because no networking capability is available at the location of a mobile computer, or to avoid use of the network for cost or power consumption reasons.

While disconnected, Venus services file system requests by relying solely on the contents of its cache. Since cache misses cannot be serviced or masked, they appear as failures to application programs and users. The persistence of changes made while disconnected is achieved via an operation log implemented on top of a transactional facility called *RVM* [16]. Venus implements a number of optimizations to reduce the size of the operation log.

To support disconnected operation, Venus operates in one of three states: *hoarding, emulation,* and *reintegration.* Venus is normally in the hoarding state, relying on server replication but always on the alert for possible disconnection. The hoarding state is so named because a key responsibility of Venus in this state is to ensure that critical objects are in the cache at the moment of disconnection. Upon disconnection, Venus enters the emulation state and remains there for the duration of disconnection. Upon reconnection, Venus enters the reintegration state, resynchronizes its cache with its AVSG, and then reverts to the hoarding state.

Venus combines implicit and explicit sources of information in its *priority-based cache management* algorithm. The implicit information consists of recent reference history, as in traditional caching algorithms. Explicit information takes the form of a per-client *hoard database (HDB),* whose entries are pathnames identifying objects of interest to the user at that client. A simple front-end program called `hoard` allows a user to update the HDB directly or via command scripts called *hoard profiles.*

Venus periodically reevaluates which objects merit retention in the cache via a process known as *hoard walking.* Hoard walking is necessary to meet user expectations about the relative importance of objects. When a cache meets these expectations, it is said to be in *equilibrium.*

4. Implementation Status

Disconnected operation in Coda was implemented over a period of two to three years. A version of disconnected operation with minimal functionality was demonstrated in October 1990. A more complete version was functional in early 1991 and began to be used regularly by members of the Coda group. By the end of 1991 almost all of the functionality had been implemented, and the user community had expanded to include several users outside the Coda group. Several of these new users had no connection to systems research whatsoever. Since mid-1992

implementation work has consisted mainly of performance tuning and bug-fixing. The current user community includes about 30 users, of whom about 20 use Coda on a regular basis. During 1992 the code was also made available to several sites outside of Carnegie Mellon University (CMU), and they are now using the system on a limited basis.

There are currently about 25 laptop and about 15 desktop clients in use. The laptops are mostly 386-based IBM PS2/L40's and the desktops are a mix of DECStation 5000/200's, Sun Sparcstations, and IBM RTs. We expect to be adding about 20 newer 486-based laptops in the near future. We currently have three DECstation 5000/200's with 2GB of disk storage in use as production servers, volumes being triply replicated across them. Additional servers are used for debugging and stress-testing pre-release versions of the system.

The production servers currently hold about 150 volumes. Roughly 25% of the volumes are *user* volumes, meaning that they are assigned to specific users who have sole administrative authority over them. Users are free, of course, to extend access rights to others by changing access-control lists on specific objects in the volume. Approximately 65% of the volumes are *project* volumes, for which administrative rights are assigned collectively to the members of a group. Most of the project volumes are used by the Coda project itself, although there are three or four other groups which have some project volumes. The other 10% of the volumes are *system* volumes, which contain program binaries, libraries, header files, and the like.

To limit our logistical and manpower commitments, we use Coda in slightly different ways on our desktop and laptop clients. On desktop clients, Coda is currently used only for user and project data. The system portions of their namespaces are in AFS, and maintenance of these namespaces is by the CMU facilities staff. Disconnected operation on these machines is therefore restricted to cases in which AFS servers are accessible but Coda servers are not. Such cases can arise when Coda servers have crashed or are down for maintenance, or when a network partitioning has separated a client from the Coda servers but not from AFS servers.

Our mobile clients do not use AFS at all and are therefore completely dependent on Coda. The system portions of the name space for this machine type are maintained by us in Coda. To minimize this maintenance effort, we initially supported only a minimal subset of the system software and have grown the size of the supported subset only in response to user requests. This strategy has worked out very well in practice, resulting in a highly usable mobile computing environment. Indeed, there are many more people wishing to use Coda laptops than we can accommodate with hardware or support services.

Porting Coda to a new machine type is relatively straightforward. Most of the code is outside the kernel. The only in-kernel code, a VFS driver [17], is small and entirely machine independent. Porting simply involves recompiling the Coda client and server code, and ensuring that the kernel works on the specific piece of hardware.

5. Qualitative Evaluation

The nature of our testbed environment has meant that we have more experience with voluntary than with involuntary disconnected operation. The most common disconnection scenario has been a user detaching his or her laptop and taking it home to work in the evening or over the weekend. We have also had cases where users have taken their laptops out of town, on business trips and on vacations, and operated disconnected for a week or more.

Although the dependence of our desktop workstations on AFS has limited our experience with involuntary disconnections, it has by no means eliminated it. Particularly during the early stages of development, the Coda servers were quite brittle and subject to fairly frequent crashes. When the crash involved corruption of server meta-data (alas, a common occurence) repairing the problem could take hours or even days. Hence, there were many opportunities for clients to involuntarily operate disconnected from user and project data.

We present our observations of hoarding, server emulation, and reintegration in the next three sections. This is followed by a section with observations that apply to the architecture as a whole.

5.1. Hoarding

In our experience, hoarding has substantially improved the usefulness of disconnected operation. Disconnected cache misses have occurred, of course, and at times they were quite painful, but there is no doubt that both the number and the severity of those misses were dramatically reduced by hoarding. Moreover, this was realized without undue burden on users and without degradation of connected mode performance.

Our experience has confirmed one of the main premises of hoarding: that implicit and explicit sources of reference information are both important for avoiding disconnected cache misses, and that a simple function of hoard and reference priorities can effectively extract and combine the information content of both sources. It also confirms that the cache manager must actively respond to local and remote disequilibrating events if the cache state is to meet user expectations about availability. In the rest of this section we examine specific aspects of hoarding in more detail.

5.1.1. Hoard Profiles

The aggregation of hints into profiles is a natural step. If profiles had not been proposed and support for them had not been built into the `hoard` tool, it's certain that users would have come up with their own ad-hoc profile formulations and support mechanisms. No one, not even the least system-savvy of our users, has had trouble understanding the concept of a profile or making modifications to pre-existing profiles on their own. And, although there has been occasional direct manipulation of the HDB via the `hoard` tool, the vast majority of user/HDB interactions have been via profiles.

Most users employ about 5-10 profiles at any one time. Typically, this includes one profile representing the user's ''personal'' data: the contents of his or her root directory, notes and mail directories, etc. Several others cover the applications most commonly run by the user: the window system, editors and text formatters, compilers and development tools, and so forth. A third class of profile typically covers data sets: source code collections, publication and correspondence directories, collections of lecture notes, and so on. A user might keep a dozen or more profiles of this type, but only activate a few at a time (i.e., submit only a subset of them to the local Venus). The number of entries in most profiles is about 5-30, with very few exceeding 50. Figure 1 gives examples of typical hoard profiles.

```
# Personal files
a /coda/usr/satya 100:d+
a /coda/usr/satya/papers/mobile93 1000:d+

# System files
a /usr/bin 100:d+
a /usr/etc 100:d+
a /usr/include 100:d+
a /usr/lib 100:d+
a /usr/local/gnu d+
a /usr/local/rcs d+
a /usr/ucb d+
```

(a)

```
# X11 files
# (from X11 maintainer)
a /usr/X11/bin/X
a /usr/X11/bin/Xvga
a /usr/X11/bin/mwm
a /usr/X11/bin/startx
a /usr/X11/bin/xclock
a /usr/X11/bin/xinit
a /usr/X11/bin/xterm
a /usr/X11/include/X11/bitmaps c+
a /usr/X11/lib/app-defaults d+
a /usr/X11/lib/fonts/misc c+
a /usr/X11/lib/system.mwmrc
```

(b)

These are typical hoard profiles in actual use by some of our users. The 'a' at the beginning of a line indicates an add-entry command. Other commands are delete an entry, clear all entries, and list entries. The numbers following some pathnames specify hoard priorities (default 10). The 'c+' and 'd+' notations indicate meta-expansion, as explained in Section 5.1.3.

Figure 1: Sample Hoard Profiles

Contrary to our expectations, there has been little direct sharing of profiles. Most of the sharing that has occurred has been indirect; that is, a user making his or her own copy of a profile and then changing it slightly. There appear to be several explanations for this:

- early users of the system were not conscientious about placing application profiles in public areas of the namespace.
- our users are, for the most part, quite sophisticated. They are used to customizing their environments via files such as `.login` and `.Xdefaults` (and, indeed, many cannot resist the temptation to constantly do so).

- most of our users are working independently or on well-partitioned aspects of a few projects. Hence, there is not much incentive to share hoard profiles.

We expect that the degree of direct profile sharing will increase as our user community grows, and as less sophisticated users begin to use Coda.

5.1.2. Multi-Level Hoard Priorities

The earliest Coda design had only a single level of hoard priority; an object was either ‘‘sticky’’ or it was not. Sticky objects were expected to be in the cache at all times. Although the sticky approach would have been simpler to implement and easier for users to understand, we are certain that it would have been much less pleasant to use and far less effective in avoiding misses than our multi-level priority scheme.

We believe that a sticky scheme would have induced the following, undesirable types of hoarding behavior:

- a tendency to be conservative in specifying hints, to avoid pinning vast amounts of low-utility data.
- a proliferation of hoard profiles for the same task or data set into, for example, ‘‘small,’’ ‘‘medium,’’ and ‘‘large’’ variants.
- micro-management of the hoard database, to account for the facts that profiles would be smaller and more numerous and that the penalty for poor specification would be higher.

The net effect of all this is that much more time and effort would have been demanded by hoarding in a sticky scheme than is the case now. This would have reduced the ability of users to hoard effectively, resulting in more frequent disconnected misses. Overall, the utility of disconnected operation would have been sharply reduced.

An argument besides simplicity which is sometimes used in favor of the sticky approach is that ‘‘you know for sure that a sticky object will be in the cache when you disconnect, whereas with priorities you only have increased probability that a hoarded object will be there.’’ That statement is simply not true. Consider a trivial example in which ten objects have been designated sticky and they occupy 90% of the total cache space. Now suppose that all ten are doubled in size by a user at another workstation. How can the local cache manager ensure that all sticky objects are cached? Clearly it cannot. The best it can do is re-fetch an arbitrary subset of the ten, leaving the rest uncached.

A negative aspect of our current priority scheme is that the range of hoard priorities is too large. Users are unable to classify objects into anywhere near 1000 equivalence classes, as the current system allows. In fact, they are often confused by such a wide range of choice. Examination of many private and a few shared profiles revealed that, while most contained at least two levels of priority, few contained more than three or four. Moreover, it was also apparent that no user employs more than six or seven distinct levels across all profiles. We therefore believe that future versions of the system should offer a priority range of about 1-10 instead of the current 1-1000. Such a change would reduce the uncertainty felt by some users as well as aid in the standardization of priorities across profiles.

5.1.3. Meta-Expansion

To reduce the verbosity of hoard profiles and to simplify their maintenance, Coda supports *meta-expansion* of HDB entries. If the letter ’c’ (or ’d’) follows a pathname in a hoard profile, the command also applies to immediate children (or all descendants). A ’+’ following the ’c’ or ’d’ indicates that the command applies to all future as well as present children or descendents.

Meta-expansion has proven to be an indispensible feature of hoarding. Virtually all hoard profiles use it to some degree, and some use it exclusively. There are also many cases in which a profile would not even have been created had meta-expansion not been available. The effort in identifying the relevant individual names and maintaining the profile over time would simply have been too great. Indeed, it is quite possible that hoarding would never have reached a threshold level of acceptance if meta-expansion had not been an option.

A somewhat unexpected benefit of meta-expansion is that it allows profiles to be constructed incrementally. That is, a usable profile can almost always be had right away by including a single line of the form ‘‘`add <rootname> d+`,’’ where `<rootname>` is the directory heading the application or data set of interest. Typically, it is also wise

to specify a low priority so that things don't get out of hand if the sub-tree turns out to be very large. Later, as experience with the application or data set increases, the profile can be refined by removing the ''root expansion'' entry and replacing it with entries expanding its children. Children then known to be uninteresting can be omitted, and variations in priority can be incorporated. This process can be repeated indefinitely, with more and more hoarding effort resulting in better and better approximations of the user's preferences.

5.1.4. Reference Spying

In many cases a user is not aware of the specific files accessed by an application. To facilitate construction of hoard profiles in such situations, Coda provides a `spy` program. This program can record all file references observed by Venus between a pair of start and stop events indicated by a user. Of course, different runtime behavior of the application can result in other files being accessed.

The `spy` program has been quite useful in deriving and tuning profiles. For example, it identified the reason why the X window system would sometimes hang when started from a disconnected workstation. It turns out that X font files are often stored in compressed format, with the X server expected to uncompress them as they are used. If the `uncompress` binary is not available when this occurs then the server will hang. Before `spy` was available, mysterious events such as this would happen in disconnected mode with annoying frequency. Since `spy`'s introduction we have been able to correct such problems on their first occurrence or, in many cases, avoid them altogether.

5.1.5. Periodic Hoard Walking

Background equilibration of the cache is an essential feature of hoarding. Without it there would be inadequate protection against involuntary disconnection. Even when voluntary disconnections are the primary type in an environment, periodic equilibration is still vital from a usability standpoint. First, it guards against a user who inadvertently forgets to demand a hoard walk before disconnecting. Second, it prevents a huge latency hit if and when a walk is demanded. This is very important because voluntary disconnections are often initiated when time is critical---for example, before leaving for the airport or when one is already late for dinner. Psychologically, users find it comforting that their machine is always ''mostly current'' with the state of the world, and that it can be made ''completely current'' with very little delay. Indeed, after a short break-in period with the system, users take for granted the fact that they'll be able to operate effectively if either voluntary or involuntary disconnection should occur.

5.1.6. Demand Hoard Walking

Foreground cache equilibration exists solely as an insurance mechanism for voluntary disconnections. The most common scenario for demand walking concerns a user who has been computing at their desktop workstation and is about to detach their laptop and take it home to continue work in the evening. In order to make sure that the latest versions of objects are cached, the user must force a hoard walk. An easy way to do this is to put the line ''`hoard walk`'' in one's `.logout` file. Most users, however, seem to like the reassurance of issuing the command manually, and internalize it as part of their standard shutdown procedure. In any case, the requirement for demand walking before voluntary disconnection cannot be eliminated since the background walk period cannot be set too close to 0. This bit of non-transparency has not been a source of complaint from our users, but it could conceivably be a problem for a less sophisticated user community.

5.2. Server Emulation

Our qualitative evaluation of server emulation centers on two issues: transparency and cache misses.

5.2.1. Transparency

Server emulation by Venus has been quite successful in making disconnected operation transparent to users. Many involuntary disconnections have not been noticed at all, and for those that have the usual indication has been only a pause of a few seconds in the user's foreground task at reintegration time. Even with voluntary disconnections, which by definition involve explicit manual actions, the smoothness of the transition has generally caused the user's awareness of disconnection to fade quickly.

The high degree of transparency is directly attributable to our use of a single client agent to support both connected and disconnected operation. If, like FACE [1], we had used a design with separate agents and local data stores for connected and disconnected operation, then every transition between the two modes would have been visible to users. Such transitions would have entailed the substitution of different versions of the same logical objects, severely hurting transparency.

5.2.2. Cache Misses

Many disconnected sessions experienced by our users, including many sessions of extended duration, involved no cache misses whatsoever. We attribute this to two primary factors. First, as noted in the preceding subsection, hoarding has been a generally effective technique for our user population. Second, most of our disconnections were of the voluntary variety, and users typically embarked on those sessions with well-formed notions of the tasks they wanted to work on. For example, they took their laptop home with the intent of editing a particular paper or working on a particular software module; they did not normally disconnect with the thought of choosing among dozens of distinct tasks.

When disconnected misses did occur, they often were not fatal to the session. In most such cases the user was able to switch to another task for which the required objects were cached. Indeed, it was often possible for a user to ''fall-back'' on different tasks two or three times before they gave up and terminated the session. Although this is a result we expected, it was still quite a relief to observe it in practice. It confirmed our belief that hoarding need not be 100% effective in order for the system to be useful.

On a cache miss, the default behavior of Venus is to return an error code. A user may optionally request Venus to block processes until cache misses can be serviced. In our experience, users have made no real use of the blocking option for handling disconnected misses. We conjecture that this is due to the fact that all of our involuntary disconnections have occurred in the context of networks with high mean-time-to-repair (MTTR). We expect blocking will be a valuable and commonly used option in in networks with low MTTRs.

5.3. Reintegration

Our qualitative evaluation of reintegration centers on two issues: performance and failures.

5.3.1. Performance

The latency of reintegration has not been a limiting factor in our experience. Most reintegrations have taken less than a minute to complete, with the majority having been in the range of 5-20 seconds. Moreover, many reintegrations have been triggered by background Venus activity rather than new user requests, so the perceived latency has often been nil.

Something which we have not experienced but consider a potential problem is the phenomenon of a *reintegration storm*. Such a storm could arise when many clients try to reintegrate with the same server at about the same time. This could occur, for instance, following recovery of a server or repair of a major network artery. The result could be serious overloading of the server and greatly increased reintegration times. We believe that we have not observed this phenomenon yet because our client population is too small and because most of our disconnections have been voluntary rather than the result of failures. We do, however, have two ideas on how the problem should be addressed:

- Have a server return a ''busy'' result once it reaches a threshold level of reintegration activity. Clients could back-off different amounts of time according to whether their reintegration was triggered by foreground or background activity, then retry. The back-off amounts in the foreground case would be relatively short and those in the background relatively long.
- Break operation logs into independent parts and reintegrate the parts separately. Of course, only the parts corresponding to foreground triggering should be reintegrated immediately; reintegration of the other parts should be delayed until the storm is over.

5.3.2. Detected Failures

Failed reintegrations have been very rare in our experience with Coda. The majority of failures that have occurred have been due to bugs in the implementation rather than update conflicts. We believe that this mostly reflects the low degree of write-sharing intrinsic to our environment. There is no doubt, however, that it also reflects certain behavioral adjustments on the part of our users. The most significant such adjustments were the tendencies to favor indirect over direct forms of sharing, and to avoid synchronization actions when one was disconnected. So, for example, if two users were working on the same paper or software module, they would be much more likely to each make their own copy and work on it than they would to make incremental updates to the original object. Moreover, the ''installation'' of a changed copy would likely be delayed until a user was certain he or she was connected. Of course, this basic pattern of sharing is the dominant one found in any Unix environment. The observation here is that it appeared to be even more common among our users than is otherwise the case.

Although detected failures have been rare, recovering from those that have occurred has been irksome. If reintegration fails, Venus writes out the operation log and related container files to a local file called a *closure.* A tool is provided for the user to inspect the contents of a closure, to compare it to the state at the AVSG, and to replay it selectively or in its entirety.

Our approach of forming closures and storing them at clients has several problems:

- there may not be enough free space at the client to store the closure. This is particularly true in the case of laptops, on which disk space is already precious.
- the recovery process is tied to a particular client. This can be annoying if a user ever uses more than one machine.
- interpreting closures and recovering data from them requires at least an intermediate level of system expertise. Moreover, even for expert users it can be difficult to determine exactly why some reintegrations failed.

The first two limitations could be addressed by migrating closures to servers rather than keeping them at clients. That strategy was, in fact, part of the original design for disconnected operation, and it continues to look like a worthwhile option.

We believe that the third problem can be addressed through a combination of techniques that reduce the number of failures that must be handled manually, and simplify the handling of those that remain. We discuss our current work in this area in Section 7.3.

5.4. Other Observations

5.4.1. Optimistic Replication

The decision to use optimistic rather than pessimistic replica control was undoubtedly the most fundamental one in the design of Coda. Having used the system for more than two years now, we remain convinced that the decision was the correct one for our type of environment.

Any pessimistic protocol must, in one way or another, allocate the rights to access objects when disconnected to particular clients. This allocation involves an unpleasant compromise between availability and ease of use. On the one hand, eliminating user involvement increases the system's responsibility, thereby lowering the sophistication of the allocation decisions. Bad allocation decisions translate directly into lowered availability; a disconnected client either does not have a copy of a critical object, or has a copy that it cannot use because of insufficient rights. On the other hand, the more involved users are in the allocation process, the less transparent the system becomes.

An optimistic replication approach avoids the need to make a priori allocation decisions altogether. Our users have never been faced with the situation in which they are disconnected and have an object cached, but they cannot access it because of insufficient replica control rights. Similarly, they have never had to formally ''grab control'' of an object in anticipation of disconnection, nor have they had to ''wrest control'' from another client that had held rights they didn't really need. The absence of these situations has been a powerful factor in making the system effective and pleasant to use.

Of course, there is an advantage of pessimistic over optimistic replica control, which is that reintegration failures cannot occur. Our experience indicates that, in a Unix file system environment, this advantage is not worth much because there simply are very few failed reintegrations. The amount and nature of sharing in the workload make reintegration failures unlikely, and users adopt work habits that reduce their likelihood even further. In effect, the necessary degree of cross-partition synchronization is achieved voluntarily, rather than being enforced by a pessimistic algorithm.

Herlihy [3] once gave the following motivation for optimistic concurrency control, which applies equally well to optimistic replica control:

> ...[optimistic replica control] is based on the premise that it is more effective to apologize than to ask permission.

In our environment, the cases in which one would wrongfully be told ‘‘no’’ when asking permission vastly outnumber those in which a ‘‘no’’ would be justified. Hence, we have found it far better to suffer the occasional indignity of making an apology than the frequent penalty of a wrongful denial.

5.4.2. Security

There have been no detected violations of security in our use of Coda, and we believe that there have been no undetected violations either. The friendliness of our testbed environment is undoubtedly one important explanation for this. However, we believe that the Coda implementation would do well security-wise even under more hostile conditions.

The basis for this belief is the faithful emulation of the AFS security model. Coda servers demand to see a user’s credentials on every client request, including reintegration. Credentials can be stolen, but this requires subversion of a client or a network-based attack. Network attacks can be thwarted through the use of (optional) message encryption, and the danger of stolen credentials is limited by associating fixed lifetimes with them. Access-control lists further limit the damage due to credential theft by confining it to areas of the namespace legitimately accessible to the subverted principal. Disconnected operation provides no back-doors that can be used to circumvent these controls.

AFS has provided good security at large-scale and under circumstances that are traditionally somewhat hostile. Indeed, we know of no other distributed file system in widespread use that provides better security with a comparable level of functionality. This strongly suggests that security would not be a factor limiting Coda’s deployment beyond our testbed environment.

5.4.3. Public Workstations

Some computing environments include a number of public workstation clusters. Although it was never a primary goal to support disconnected operation in that domain, it was something that we hoped would be possible and which influenced Coda’s early design to some degree.

Our experience with disconnected operation has convinced us that it is simply not well suited to public access conditions. One problem is that of security. Without disconnected operation, it is the case that when a user leaves a public workstation his or her data is all safely at servers and he or she is totally independent of that workstation. This allows careful users to flush their authentication tokens and their sensitive data from the cache when they depart, and to similarly ‘‘scrub’’ the workstation clean when they arrive. But with disconnected operation, scrubbing is not necessarily an option. The departing user cannot scrub if he or she has dirty objects in the cache, waiting to be reintegrated. The need to leave valid authentication tokens with the cache manager is particularly worrying, as that exposes the user to arbitrary damage. And even if damage does not arise due to security breach, the departing user still must worry that a future user will scrub the machine and thereby lose his or her pending updates.

The other major factor that makes disconnected operation unsuited to public workstations is the latency associated with hoarding. Loading a cache with one’s full ‘‘hoardable set’’ can take many minutes. Although this is done in the background, it can still slow a client machine down considerably. Moreover, if a user only intends to use a machine briefly, as is often the case with public machines, then the effort of hoarding is likely to be a waste. It is only when the cost of hoarding can be amortized over a long usage period that it becomes a worthwhile exercise.

6. Quantitative Evaluation

An earlier paper [7] presented measurements that shed light on key aspects of disconnected operation in Coda. Perhaps the most valuable of those measurements was the compelling evidence that optimistic replication in a Coda-like environment would indeed lead to very few write-write conflicts. That evidence was based on a year-long study of a 400-user AFS cell at CMU. The data showed that cross-user write-sharing was extremely rare. Over 99% of all file and directory modifications were by the previous writer, and the chances of two different users modifying the same object less than a day apart was at most 0.72%. If certain system administration files which skewed the data were excluded, the absence of write-sharing was even more striking: more than 99.7% of all mutations were by the previous writer, and the chances of two different users modifying the same object within a week were less than 0.3%.

In the following sections we present new measurements of Coda. These measurements either address questions not considered in our earlier paper, or provide more detailed and up-to-date data on issues previously addressed. The questions we address here are:

- How large a local disk does one need?
- How noticeable is reintegration?
- How important are optimizations to the operation log?

6.1. Methodology

To estimate disk space requirements we relied on simulations driven by an extensive set of file reference traces that we had collected [12]. Our analysis was comprehensive, taking into account the effect of all references in a trace whether they were to Coda, AFS or the local file system. The traces were carefully selected for sustained high levels of activity from over 1700 samples. We chose 10 workstation traces, 5 representing 12-hour workdays and the other 5 representing week-long activity. Table 1 identifies these traces and presents a summary of the key characteristics of each.

Trace Identifier	Machine Name	Machine Type	Simulation Start	Trace Records
Work-Day #1	brahms.coda.cs.cmu.edu	IBM RT-PC	25-Mar-91, 11:00	195289
Work-Day #2	holst.coda.cs.cmu.edu	DECstation 3100	22-Feb-91, 09:15	348589
Work-Day #3	ives.coda.cs.cmu.edu	DECstation 3100	05-Mar-91, 08:45	134497
Work-Day #4	mozart.coda.cs.cmu.edu	DECstation 3100	11-Mar-91, 11:45	238626
Work-Day #5	verdi.coda.cs.cmu.edu	DECstation 3100	21-Feb-91, 12:00	294211
Full-Week #1	concord.nectar.cs.cmu.edu	Sun 4/330	26-Jul-91, 11:41	3948544
Full-Week #2	holst.coda.cs.cmu.edu	DECstation 3100	18-Aug-91,23:21	3492335
Full-Week #3	ives.coda.cs.cmu.edu	DECstation 3100	03-May-91,12:15	4129775
Full-Week #4	messiaen.coda.cs.cmu.edu	DECstation 3100	27-Sep-91, 00:15	1613911
Full-Week #5	purcell.coda.cs.cmu.edu	DECstation 3100	21-Aug-91,14:47	2173191

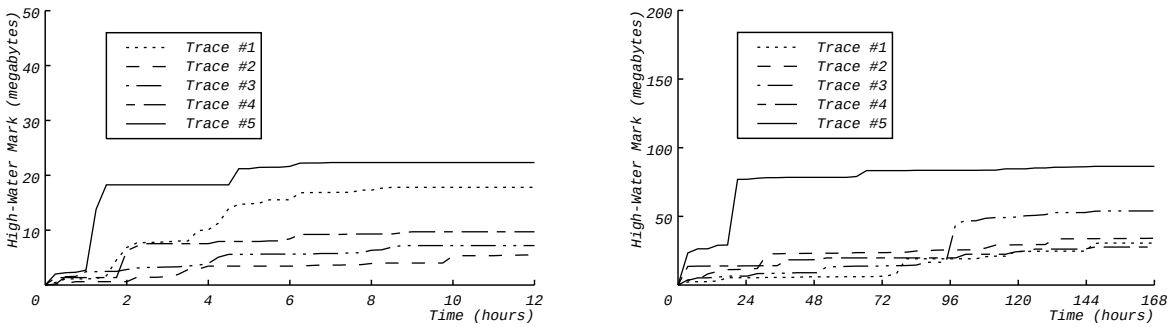
Table 1: Vital Statistics for the Work-Day and Full-Week Traces

To address the question of reintegration latency, we performed a well-defined set of activities while disconnected and timed the duration of the reintegration phase after each. One of these activities was the running of the Andrew benchmark [4]. Another was the compilation of the then-current version of Venus. A third class of activities corresponded to the set of traces in Table 1. We effectively ‘‘inverted’’ these traces and generated a command script from each. When executed, each of these scripts produced a trace isomorphic to the one it was generated from. This gave us a controlled and repeatable way of emulating real user activity.

We combined these two techniques to assess the value of log optimizations. Using our trace suite, we compared disk usage with and without optimizations enabled. We also measured the impact of log optimizations on reintegration latency for the set of activites described in the previous paragraph.

6.2. Disk Space Requirements

Figure 2 shows the *high-water mark* of cache space usage for the Work-Day and Full-Week traces as a function of time. The high-water mark is simply the maximum cache space in use at the current and all previous points in the simulation. The high-water mark therefore never declines, although the current cache space in use may (due to the deletion of objects).



(a) Work-Day Traces (b) Full-Week Traces

This graph presents the high-water marks of cache usage for each trace in Table 1. Note that the vertical axis on the graphs for Work-Day and Full-Week traces are different.

Figure 2: High-Water Marks of Cache Space Usage

These curves indicate that cache space usage tends to grow rapidly at the start, but tapers off quite soon. For example, most of the Work-Day traces had reached 80% of their 12-hour high-water marks within a few hours of their start. Similarly, all but one of the Full-Week traces had reached a substantial fraction of their 7-day high-water marks by the end of the second day. Note that it was not the case that the workstations simply became idle after the first parts of the traces; the traces were carefully selected to ensure that users were active right through to the end of the simulated periods.

These results are encouraging from the point of view of disconnected operation. The most expansive of the Work-Day traces peaked out below 25 MB, with the median of the traces peaking at around 10 MB. For the Full-Week traces, the maximum level reached was under 100 MB and the median was under 50 MB. This suggests that today's typical desktop workstation, with a disk of 100 MB to 1 GB, should be able to support many disconnections of a week or more in duration. Even the 60-200MB disk capacity of many laptops today is adequate for extended periods of disconnected operation. These observations corroborate our first-hand experience in using Coda laptops.

6.3. Reintegration Latency

Reintegration latency is a function of the update activity at a client while disconnected. In our use of the system, most one-day disconnections have resulted in reintegration times of a minute or less, and a few longer disconnections have taken a few minutes. Table 2 reports the latency, number of log records, and amount of data *back-fetched* for each of our reintegration experiments. Back-fetching refers to the transfer of data from client to server representing disconnected file store operations. Reintegration occurs in three subphases: a *prelude,* an *interlude,* and *postlude.* Latency is reported separately for the subphases as well as in total. On average, these subphases contributed 10%, 80% and 10% respectively to the total latency.

These results confirm our subjective experience that reintegration after a typical one-day disconnection is hardly perceptible. The Andrew benchmark, Venus make, and four of the five Work-Day trace-replay experiments all reintegrated in under 40 seconds. The other Work-Day trace-replay experiment took only slightly more than a minute to reintegrate.

The reintegration times for the week-long traces are also consistent with our qualitative observations. Four of the five week-long trace-replay experiments reintegrated in under five minutes, with three completing in three minutes or less. The other trace-replay experiment is an outlier, requiring about 20 minutes to reintegrate.

Task	Log Record Total	Back-Fetch Total	Latency			
			Prelude	Interlude	Postlude	Total
Andrew Benchmark	203	1.2	1.8	7.5	.8	10 (1)
Venus Make	146	10.3	1.4	36.2	.4	38 (1)
Work-Day #1 Replay	1422	4.9	6.5	54.7	10.7	72 (5)
Work-Day #2 Replay	316	.9	1.9	9.8	1.7	14 (1)
Work-Day #3 Replay	212	.8	1.0	6.2	.9	8 (0)
Work-Day #4 Replay	873	1.3	2.9	23.2	5.9	32 (3)
Work-Day #5 Replay	99	4.0	.9	20.5	.5	22 (2)
Full-Week #1 Replay	1802	15.9	15.2	138.8	21.9	176 (3)
Full-Week #2 Replay	1664	17.5	16.2	129.1	15.0	160 (2)
Full-Week #3 Replay	7199	23.7	152.6	881.3	183.0	1217(12)
Full-Week #4 Replay	1159	15.1	5.1	77.4	7.0	90 (1)
Full-Week #5 Replay	2676	35.8	28.2	212.8	31.7	273 (9)

This data was obtained with a DECstation 5000/200 client and server. The Back-Fetch figures are in megabytes. Latency figures are in seconds. Each latency number is the mean of three trials. The numbers in parentheses in the ''Latency Total'' column are standard deviations. Standard deviations for the individual phases are omitted for space reasons.

Table 2: Reintegration Latency

In tracking down the reason for this anomaly, we discovered a significant shortcoming of our implementation. We found, much to our surprise, that the time for reintegration bore a non-linear relationship to the size of the operation log and the number of bytes back-fetched. Specifically, the regression coefficients were .026 for the number of log records, .0000186 for its square, and 2.535 for the number of megabytes back-fetched. The quality of fit was excellent, with an R^2 value of 0.999.

The first coefficient implies a direct overhead per log record of 26 milliseconds. This seems about right, given that many records will require at least one disk access at the server during the interlude phase. The third coefficient implies a rate of about 400 KB/s for bulk data transfer. This too seems about right, given that the maximum disk-to-disk transfer rate between 2 DECstation 5000/200s on an Ethernet that we've observed is 476 kilobytes/second. The source of the quadratic term turned out to be a naive sorting algorithm that was used on the servers to avoid deadlocks during replay. For disconnected sessions of less than a week, the linear terms dominate the quadratic term. This explains why we have never observed long reintegration times in normal use of Coda. But around a week, the quadratic term begins to dominate. Clearly some implementation changes will be necessary to make reintegration linear. We do not see these as being conceptually difficult, but they will require a fair amount of code modification.

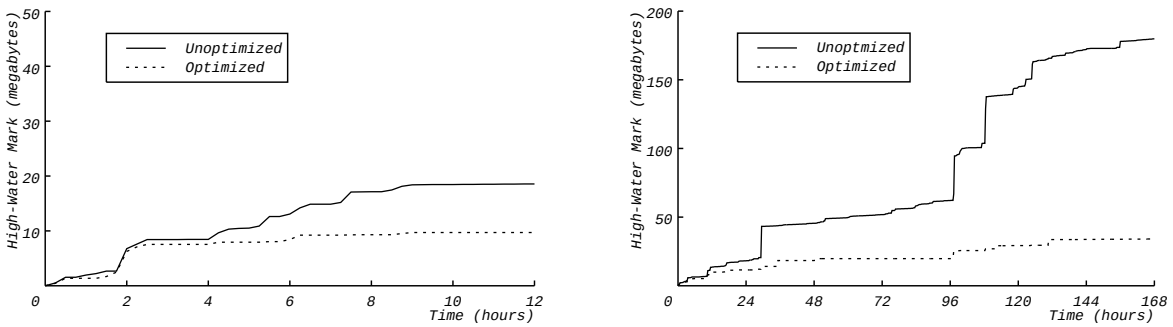
It is worth making two additional points about reintegration latency here. First, because reintegration is often triggered by a daemon rather than a user request, perceived latency is often nil. That is, reintegrations often occur entirely in the background and do not delay user computation at all. Second, the trace-replay experiments reflect activity that was originally performed in a number of volumes. For the Work-Day traces 5-10 volumes were typically involved, and for the Full-Week traces the number was typically 10-15. For logistical reasons, the replay experiments were each performed within a single Coda volume. Hence, there was only one reintegration for each experiment. Following an actual disconnected execution of the trace activity, though, there would have been a number of smaller reintegrations instead of one large one. If the reintegrated volumes were spread over different servers, a significant amount of parallelism could have been realized. The total latency might therefore have been much smaller, perhaps by a factor of three or four.

6.4. Value of Log Optimizations

Venus uses a number of optimizations to reduce the length of the operation log. A small log conserves disk space, a critical resource during periods of disconnection. It also improves reintegration performance by reducing latency and server load. Details of these optimizations can be found elsewhere [7, 8].

In order to understand how much space these optimizations save in practice, our Venus simulator was augmented to

report cache usage statistics with the optimizations turned off as well as on. Figure 3 compares the median high-water marks of space usage for our trace suite with and without optimizations.



(a) Work-Day Traces (b) Full-Week Traces

Each curve above represents the median values of the high-water marks of space usage for the five corresponding traces. Note that the vertical axis on the two graphs are different.

Figure 3: Optimized versus Unoptimized Cache Space High-Water Marks

The differences between the curves in each case are substantial. After an initial period in which the two curves increase more or less together, the unoptimized curves continue to increase while the optimized curves taper off. For the Work-Day traces, the unoptimized total has grown to nearly twice that of the optimized case by the 12-hour mark. The trend continues unabated with the Full-Week traces, with the unoptimized total being more than 5 times that of the optimized case at the end of the week. This equates to a difference of more than 145 megabytes. The slopes of the two lines indicate that the difference would increase even further over periods of greater length.

Table 3 shows that the differences for certain individual traces are even more striking. That table lists the unoptimized and optimized totals for each trace at its termination. In addition, each total is broken down into its two constituents: cache container space and RVM space. Cache container space refers to the space used by the local files that are used to hold the images of the current versions of Coda files.

The greatest savings tend to be realized in cache container space, although the RVM space savings can also be substantial. The far right column shows the ratio of unoptimized to optimized total space usage. The maximum ratio for the Work-Day traces is 3.1, indicating that more than three times the amount of space would have been needed without the optimizations. The maximum ratio for the Full-Week traces is an astonishing 28.9, which corresponds to a difference of more than 850 megabytes.

Trace	Container Space		RVM Space		Total Space		
	Unopt	Opt	Unopt	Opt	Unopt	Opt	Ratio
Work-Day #1	34.6	15.2	2.9	2.7	37.5	17.8	2.1
Work-Day #2	14.9	4.0	2.3	1.5	17.2	5.5	3.1
Work-Day #3	7.9	5.8	1.6	1.4	9.5	7.2	1.3
Work-Day #4	16.7	8.2	1.9	1.5	18.6	9.7	1.9
Work-Day #5	59.2	21.3	1.2	1.1	60.4	22.3	2.7
Full-Week #1	872.6	25.9	11.7	4.8	884.3	30.6	28.9
Full-Week #2	90.7	28.3	13.3	5.9	104.0	34.2	3.0
Full-Week #3	119.9	45.0	46.2	9.1	165.9	54.0	3.1
Full-Week #4	222.1	23.9	5.5	3.9	227.5	27.7	8.2
Full-Week #5	170.8	79.0	9.1	7.7	179.8	86.5	2.1

The figures in the "Unopt" and "Opt" columns are in megabytes

Table 3: Optimized versus Unoptimized Space Usage

These results confirm that log optimizations are critical for managing space at a disconnected client. But they are also important for keeping reintegration latency low. To confirm this, we used the regression results and measured values of unoptimized log records and data back-fetched from the experiments reported in Section 6.3. Using this information we estimated how long reintegration would have taken, had log optimizations not been done. Table 4

presents our results.

Task	Log Record Total		Back-Fetch Total		Latency		
	Unopt	Opt	Unopt	Opt	Unopt	Opt	Ratio
Andrew Benchmark	211	203	1.5	1.2	10	10	1.0
Venus Make	156	146	19.5	10.3	54	38	1.4
Work-Day #1 Replay	2422	1422	19.1	4.9	221	72	3.1
Work-Day #2 Replay	4093	316	10.9	.9	446	14	31.9
Work-Day #3 Replay	842	212	2.1	.8	41	8	5.1
Work-Day #4 Replay	2439	873	8.5	1.3	196	32	6.1
Work-Day #5 Replay	545	99	40.9	4.0	123	22	5.6
Full-Week #1 Replay	33923	1802	846.9	15.9	24433	176	138.8
Full-Week #2 Replay	36855	1664	62.4	17.5	26381	160	164.9
Full-Week #3 Replay	175392	7199	75.0	23.7	576930	1217	474.1
Full-Week #4 Replay	8519	1159	199.1	15.1	2076	90	23.1
Full-Week #5 Replay	8873	2676	92.7	35.8	1930	273	7.1

Back-fetch figures are in megabytes, and latencies in seconds. The reported latencies are the means of three trials. Standard deviations are omitted for brevity.

Table 4: Optimized versus Unoptimized Reintegration Latency

The time savings due to optimizations are enormous. The figures indicate that without the optimizations, reintegration of the trace-replay experiments would have averaged 10 times longer than actually occurred for the Work-Day set, and 160 times longer for the Full-Week set. Reintegrating the unoptimized replay of Full-Week trace #3 would have taken more than 6 days, or nearly as long as the period of disconnection! Obviously, much of the extra time is due to the fact that the unoptimized log record totals are well into the range at which the quadratic steps of our implementation dominate. Although the savings will not be as great when our code is made more efficient, it will not be inconsequential by any means. Even if the quadratic term is ignored, the ratios of unoptimized to optimized latency are still pronounced: on average, 4.5 for Work-Day traces and 7.6 for the Full-Week traces.

7. Work in Progress

Coda is a system under active development. In the following sections we describe work currently under way to enhance the functionality of Coda as well as to alleviate some of its current shortcomings.

7.1. Exploiting Weak Connectivity

Although disconnected operation in Coda has proven to be effective for using distributed file systems from mobile computers, it has several limitations:

- cache misses are not transparent. A user may be able to work in spite of some cache misses, but certain critical misses may frustrate these efforts.
- longer disconnections increase the likelihood of resource exhaustion on the client from the growing operation log and new data in the cache.
- longer disconnections also increase the probability of conflicts requiring manual intervention upon reconnection.

Wireless technologies such as cellular phone and even traditional dialup lines present an opportunity to alleviate some of the shortcomings of disconnected operation. These *weak connections* are slower than LANs, and some of the wireless technologies have the additional properties of intermittence and non-trivial cost. The characteristics of these networks differ substantially from those of LANs, on which many distributed file systems are based.

We are exploring techniques to exploit weak connectivity in a number of ways:

- Coda clients will manage use of the network intelligently. By using the network in a preemptive, prioritized fashion, the system will be able to promptly service critical cache misses. It will propagate mutations back to the servers in the background to prevent conflicts that would arise at reintegration

time and to reclaim local resources. It will allow users to weaken consistency on an object-specific basis to save bandwidth.

- Coda clients will minimize bandwidth requirements by using techniques such as batching and compression. Techniques in this class demand more server computation per request, so the state of the server will play a role in the use of these techniques.
- Coda clients will dynamically detect and adapt to changes in network performance. This will be especially important when connectivity is intermittent.

An important consideration in the use of weak connections is the issue of callback maintenance. Callback-based cache consistency schemes were designed to minimize client-server communication, but with an underlying assumption that the network is fast and reliable. After a network failure all callbacks are invalid. In an intermittent low-bandwidth network, the cost of revalidation may be substantial and may nullify the performance benefits of callback-based caching.

To address this issue we have introduced the concept of *large granularity callbacks.* A large granularity trades off precision of invalidation for speed of validation after connectivity changes. Venus will choose the granularity on a per-volume basis, adapting to the current networking conditions as well as the observed rate of callback breaks due to mutations elsewhere in the system. Further details on this approach can be found in a recent paper [11].

7.2. Hoarding Improvements

We are in the process of developing tools and techniques to reduce the burden of hoarding on users, and to assist them in accurately assessing which files to hoard. A key problem we are addressing in this context is the choice of proper metrics for evaluating the quality of hoarding.

Today, the only metric of caching quality is the *miss ratio.* The underlying assumption of this metric is that all cache misses are equivalent (that is, all cache misses exact roughly the same penalty from the user). This assumption is valid in the absense of disconnections and weak connections because the performance penalty resulting from a cache miss is small and independent of file length. This assumption is not valid during disconnected operation and may not be valid for weakly-connected operation, depending on the strength of the connection. The cache miss ratio further assumes that the timing of cache misses is irrelevant. But the user may react differently to a cache miss occurring within the first few minutes of disconnection than to one occurring near the end of the disconnection.

We are extending the analysis of hoarding tools and techniques using new metrics such as:

- the time until the first cache miss occurs.
- the time until a critical cache miss occurs.
- the time until the cumulative effect of multiple cache misses exceeds a threshold.
- the time until connection transparency is lost.
- the percentage of the cache actually referenced when disconnected or weakly-connected, as a measure of overly-generous hoarding.
- the change in connected-mode miss ratio due to hoarding.

We plan to use these metrics to evaluate the relative value of different kinds of hoarding assistance. For example, under what circumstances does one tool prove better than another? Some of our experiments will be performed on-line as users work. Others will be performed off-line using post-mortem analysis of file reference traces. The tools we plan to build will address a variety of needs pertinent to hoarding. Examples include tools to support task-based hoarding and a graphical interface to accept hoard information and provide feedback regarding the cache contents.

7.3. Application-Specific Conflict Resolution

Our experience with Coda has established the importance of optimistic replication for mobile computing. But optimistic replication brings with it the need to detect and resolve concurrent updates in multiple partitions. Today Coda provides for transparent resolution of directory updates. We are extending our work to support transparent resolution on arbitrary files. Since the operating system does not possess any semantic knowledge of file contents, it is necessary to obtain assistance from applications.

The key is to provide an application-independent invocation mechanism that allows pre-installed, *application-specific resolvers (ASRs)* to be transparently invoked and executed when a conflict is detected. As a practical example of this approach, consider a calendar management application. The ASR in this case might merge appointment database copies by selecting all non-conflicting appointments and, for those time slots with conflicts, choosing to retain one arbitrarily and sending mail to the rejected party(s).

We have recently described such an interface for supporting ASRs [10]. Our design addresses the following issues:

- An application-independent interface for transparently invoking ASRs.
- An inheritance mechanism to allow convenient rule-based specification of ASRs based on attributes such as file extension or position in the naming hierarchy.
- A fault tolerance mechanism that encapsulates ASR execution.

Even in situations where manual intervention is unavoidable, ASR technology may be used for partial automation. Consider, for example, the case of two users who have both edited a document or program source file. An ‘‘interactive ASR’’ could be employed in this case which pops up side-by-side windows containing the two versions and highlights the sections which differ. The user could then quickly perform the merge by cutting and pasting. Similarly, a more useful version of the calendar management ASR might begin with a view of the appointment schedule merged with respect to all non-conflicting time slots, then prompt the user to choose between the alternatives for each slot that conflicts.

Another class of ASRs that may be valuable involves automatic re-execution of rejected computations by Venus. This is precisely the approach advocated by Davidson in her seminal work on optimistic replication in databases [2], and it will be feasible to use in Coda once the transactional extensions described in Section 7.4 are completed. Automatic re-execution would be appropriate in many cases involving the `make` program, for example.

7.4. Transactional Extensions for Mobile Computing

With the increasing frequency and scale of data sharing activities made possible by distributed Unix file systems such as AFS, Coda, and NFS [13], there is a growing need for effective consistency support for concurrent file accesses. The problem is especially acute in the case of mobile computing, because extended periods of disconnected or weakly-connected operation may increase the probability of read-write inconsistencies in shared data.

Consider, for example, a CEO using a disconnected laptop to work on a report for an upcoming shareholder’s meeting. Before disconnection she cached a spreadsheet with the most recent budget figures available. She writes her report based on the numbers in that spreadsheet. During her absence, new budget figures become available and the server’s copy of the spreadsheet is updated. When the CEO returns and reintegrates, she needs to discover that her report is based on stale budget data. Note that this is not a write-write conflict, since no one else has updated her report. Rather it is a read-write conflict, between the spreadsheet and the report. No Unix system today has the ability to detect and deal with such problems.

We are exploring techniques for extending the Unix interface with transactions to provide this functionality. A key attribute of our effort is upward compatibility with the Unix paradigm. Direct transplantation of traditional database transactions into Unix is inappropriate. The significant differences in user environment, transaction duration and object size between Unix file systems and database systems requires transactional mechanisms to be specially tailored. These considerations are central to our design of a new kind of transaction, called *isolation-only transaction,* whose use will improve the consistency properties of Unix file access in partitioned networking

environments.

A distinct but related area of investigation is to explore the effects of out-of-band communication on mobile computing. For example, a disconnected user may receive information via a fax or phone call that he incorporates into the documents he is working on. What system support can we provide him to demarcate work done before that out-of-band communication? This will become important if he later needs to extricate himself from a write-write or read-write conflict.

8. Conclusion

In this paper, we have focused on disconnected operation almost to the exclusion of server replication. This is primarily because disconnected operation is the newer concept, and because it is so central to solving the problems that arise in mobile computing. However, the importance of server replication should not be underestimated. Server replication is important because it reduces the frequency and duration of disconnected operation. Thus server replication and disconnected operation are properly viewed as complementary mechanisms for high availability.

Since our original description of disconnected operation in Coda [7] there has been considerable interest in incorporating this idea into other systems. One example is the work by Huston and Honeyman [5] in implementing disconnected operation in AFS. These efforts, together with our own substantial experience with disconnected operation in Coda, are evidence of the soundness of the underlying concept and the feasibility of its effective implementation.

None of the shortcomings exposed in over two years of serious use of disconnected operation in Coda are fatal. Rather, they all point to desirable ways in which the system should evolve. We are actively refining the system along these dimensions, and have every reason to believe that these refinements will render Coda an even more usable and effective platform for mobile computing.

Acknowledgments

We wish to thank all the members of the Coda project, past and present, for their contributions to this work. David Steere, Brian Noble, Hank Mashburn, and Josh Raiff deserve special mention. We also wish to thank our brave and tolerant user community for their willingness to use an experimental system.

References

[1] Cova, L.L.
Resource Management in Federated Computing Environments.
PhD thesis, Department of Computer Science, Princeton University, October, 1990.

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

[3] Herlihy, M.
Optimistic Concurrency Control for Abstract Data Types.
In *Proceedings of the Fifth Annual Symposium on Principles of Distributed Computing*. August, 1986.

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

[5] Huston, L., Honeyman, P.
Disconnected Operation for AFS.
In *Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing*. Cambridge, MA, August, 1993.

[6] Kazar, M.L. .
Synchronization and Caching Issues in the Andrew File System.
In *Winter Usenix Conference Proceedings, Dallas, TX*. 1988.

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

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

[9] Kumar, P., Satyanarayanan, M.
Log-Based Directory Resolution in the Coda File System.
In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*. San Diego, CA, January, 1993.

[10] Kumar, P., Satyanarayanan, M.
Supporting Application-Specific Resolution in an Optimistically Replicated File System.
June, 1993.
submitted to 4th IEEE Workshop on Workstation Operating Systems, Napa, CA, October 1993.

[11] Mummert, L.B., Satyanarayanan, M.
File Cache Consistency in a Weakly Connected Environment.
June, 1993.
submitted to 4th IEEE Workshop on Workstation Operating Systems, Napa, CA, October 1993.

[12] Mummert, L.B.
Efficient Long-Term File Reference Tracing.
1993.
in preparation.

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

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

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

[16] Satyanarayanan, M., Mashburn, H.H., Kumar, P., Steere, D.C., Kistler, J.J.
Lightweight Recoverable Virtual Memory.
Technical Report CMU-CS-93-143, School of Computer Science, Carnegie Mellon University, March, 1993.

[17] Steere, D.C., Kistler, J.J., Satyanarayanan, M.
Efficient User-Level Cache File Management on the Sun Vnode Interface.
In *Summer Usenix Conference Proceedings, Anaheim*. June, 1990.