The Evolution of Coda

M. SATYANARAYANAN
Carnegie Mellon University

Failure-resilient, scalable, and secure read-write access to shared information by mobile and static users over wireless and wired networks is a fundamental computing challenge. In this article, we describe how the Coda file system has evolved to meet this challenge through the development of mechanisms for server replication, disconnected operation, adaptive use of weak connectivity, isolation-only transactions, translucent caching, and opportunistic exploitation of hardware surrogates. For each mechanism, the article explains how usage experience with it led to the insights for another mechanism. It also shows how Coda has been influenced by the work of other researchers and by industry. The article closes with a discussion of the technical and nontechnical lessons that can be learned from the evolution of the system.

Categories and Subject Descriptors: D.4.3 [**Software**]: File Systems Management—*Distributed file systems*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; E.5 [**Data**]: Files—*Backup/recovery*; C.2.4 [**Computer Systems Organization**]: Distributed Systems—*Client/server*; C.2.1 [**Computer Systems Organization**]: Network Architecture and Design—*Wireless communication*; D.2.7 [**Software**]: Distribution, Maintenance, and Enhancement; D.2.11 [**Software**]: Software architecture; D.2.13 [**Software**]: Reusable Software

General Terms: Design, Experimentation, Performance, Reliability

Additional Key Words and Phrases: Adaptation, caching, conflict resolution, continuous data access, data staging, disaster recovery, disconnected operation, failure, high availability, hoarding, intermittent networks, isolation-only transactions, Linux, low-bandwidth networks, mobile computing, optimistic replica control, server replication, translucent cache management, UNIX, weakly connected operation, Windows

1. INTRODUCTION

Coda is best known today as a system that enables mobile file access through its support for disconnected and weakly connected operation. It may therefore come as a surprise that support for mobility was not one of the original goals of Coda. The very concept of mobile computing bordered on science fiction when Coda was first conceived in early 1987. Today, mobile information access is not only feasible but is beginning to be regarded as indispensable. Key elements of

Primary funding for this work was provided by the National Science Foundation (NSF) through grants CCR-8657907 and ECD-8907068, and by the Defense Advanced Research Projects Agency (DARPA), the U.S. Air Force (USAF), and the U.S. Navy (USN) under contracts F33615-87-C-1499, F33615-90-C-1465, F19628-93-C-0193, F19628-96-C-0061, and N66001-99-28918.
Author's address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213; email: satya@cs.cmu.edu.

Coda's design are being brought into mainstream commercial practice through their adoption in the IntelliMirror component of Windows 2000 [Gray 1997; Short 1997]. While much still remains to be done, both Coda and mobile computing have clearly come a long way.

How did this transformation occur? What were the key events in our work, as well as that of others, that made this shift possible? What is the level of consensus in the research community in this field? What are the open questions? Whither Coda?

This article answers such questions through a narrative account of the evolution of the Coda File System. It describes how the work of other researchers as well as hardware and software developments in industry influenced us. It also shows how our understanding of the core issues in mobile computing grew and matured with the development of Coda. The paper concludes with a discussion of the technical and nontechnical lessons that can be learned from the evolution of the system.

2. ORIGIN AND BACKGROUND

Coda began as an epilogue to the Andrew File System (AFS) [Howard et al. 1988; Satyanarayanan 1990]. As a principal architect and implementor of AFS, I had the opportunity to witness first-hand the power of a highly scalable, location transparent distributed file system to unite the large user community at the Carnegie Mellon campus. By providing an easy-to-use information sharing backbone, a community-wide namespace with effective access control, and upward compatibility with existing applications, AFS soon became a firmly entrenched component of our computing environment.

Unfortunately, AFS and similar systems are vulnerable to server and network failures. In taking advantage of the benefits of these systems, clients typically become dependent on files cached from servers. A server or network failure renders these files inaccessible, leaving clients crippled for the duration of the failure. In a large enough system, unplanned outages of servers and network segments are practically impossible to avoid. Together, these considerations augur ill for the growth of a distributed file system. Our extensive experience with AFS confirmed that these concerns were indeed justified—failures did occur, and they did cause significant inconvenience to our user community.

An obvious question follows: Can the virtues of AFS be preserved, while alleviating its vulnerability to failures? We initiated the Coda project in early 1987 with the goal of answering this question [Satyanarayanan et al. 1987]. At that time, high data availability was a feature supported only by an exotic (and hence expensive) breed of systems such as Tandem [Bartlett et al. 1988], Stratus [Harrison and Schmitt 1987], and Auragen [Borg et al. 1989] that relied extensively on proprietary hardware. In contrast, our goal was to build a system with high data availability relying solely on commodity hardware and widely accepted software standards.

The name "Coda" was chosen to reflect its usage in classical music, as "something that serves to round out, conclude or summarize and that has interest of its own" [Webster's Ninth New Collegiate Dictionary 1988].[1] Our expectation was that Coda would have a research life of at most a couple of years. Little did we suspect that a decade and a half later the system would still be generating a considerable amount of interest and activity, and that its cumulative research contributions would overshadow those of AFS!

To bound the scope of our effort, we began by preserving the basic design choices of AFS in all areas other than high availability. Surprisingly, this architectural inheritance has proven to be robust and durable over a long evolution, and a shift in focus from high availability to mobility. The most salient aspects of Coda's AFS heritage [Howard et al. 1988; Satyanarayanan et al. 1990] are the following:

—Use of the *client-server model*, with a small number of trusted servers that are the custodians of data, and a much larger number of untrusted clients that use these data.

—A *location transparent file name space* that completely hides the identities of individual servers, and allows easy redistribution of data across new or existing servers.

—*Aggressive use of whole-file caching* by clients on their local disks, with a further level of block-level caching in their main memory buffer caches.

—The *organization of data into volumes* to facilitate system administration. Each volume is a partial subtree of the total name space, and its point of attachment in the name space is identical at all clients.

—An *access list model of protection*, combined with an authentication mechanism that is integrated with the remote procedure call package used for communication between clients and servers.

—A *user-level implementation strategy* that strives to minimize kernel modifications. The bulk of the implementation complexity at the client is encapsulated into a user-level cache manager called *Venus*.

—*Callback-based cache coherence*, whereby a server remembers what objects have been cached by a client, and notifies it when another client updates one of those objects. This eliminates the need for clients to validate cached objects before using them, thus improving performance and scalability.

3. SERVER REPLICATION: 1987–1991

3.1 Related Work

Replication is the fundamental technique for ensuring data availability during failures. Research on replication techniques had already been in progress for many years when we began work on Coda. As early as 1979, Gifford [1979] had described a replication technique known as *weighted voting*. Herlihy [1986] later developed a more general approach known as *quorum consensus*. Many

[1]It was later observed that "CODA" can also be viewed as an acronym for "constant data availability." This was, however, an afterthought and not the original reason for the choice of name.

other researchers such as Paris [1986], Long [1990], and El Abbadi [El Abbadi and Toueg 1989], were active in the field in the mid to late 1980s.

There were two characteristics common to most of this work: the research tended to concentrate on algorithm design, with little attention to practical issues such as scalability, usability, or system integration; and second, *pessimistic* replica control strategies dominated the research agenda. Such strategies preserve consistency of data even at the cost of denying access to replicas in some network partitions. The specific consistency guarantee is *one-copy serializability*: for any sequence of read and write operations on a replicated object, the sequence of values it assumes is identical to what it would assume if it were not replicated.

3.1.1 *Optimistic Replication.* In contrast to pessimistic strategies, *optimistic* replica control strategies trade consistency for availability. They permit reads and writes in any partition with a replica [Davidson et al. 1985]. Stale reads and conflicting writes are inherent risks in these strategies. The classic approach to coping with these risks is to ignore stale reads, and to detect and resolve conflicting writes after their occurrence.

We rejected pessimistic replica control because it placed unacceptable limits on data availability. The strengths and weaknesses of optimistic replication better matched our design goals. The dominant influence on our choice was the low degree of write-sharing typical of many user workloads. This implied that an optimistic strategy was likely to lead to relatively few update conflicts. An optimistic strategy was also consistent with our overall goal of providing the highest possible availability of data.

Once we had decided to use optimistic replication, we looked for examples of previous work from which we could benefit. By 1987 there was a substantial body of published research pertaining to optimistic replication in databases. However, little of that work directly applied to Coda. The atomic transaction concept, so central to cleanup after failure in a database, simply did not exist in a UNIX file system. The vector clock abstraction for reasoning about optimistic replication [Babaoglu and Marzullo 1993; Fidge 1988] had not yet been developed.

3.1.2 *LOCUS*. The only previous work directly relevant to Coda was LOCUS [Popek et al. 1981; Walker et al. 1983]. This distributed system, built at UCLA in the early 1980s, had pioneered the use of optimistic replication in a UNIX file system. The published research accomplishments of LOCUS, such as identification of version vectors [Parker et al. 1983] as an accurate and efficient technique for detecting update conficts, led us to believe that the hardest problems in optimistic replication had been solved.

Unfortunately, this belief proved to be misplaced. As our design progressed, we realized that the very different architectural assumptions of Coda precluded direct use of solutions from LOCUS. For example, LOCUS was based on a peer-to-peer architecture whereas Coda was based on a client-server architecture. LOCUS assumed that the system and network were small and homogeneous.

Hence, it was reasonable to assume that nodes could rapidly converge on a consistent network topology after failure and recovery events. In contrast, Coda was being designed for a much larger and more heterogeneous environment where timely consensus on network topology could not be assumed. The system had to operate correctly even when different parts of the system had very different notions of what was up and what was down. Scalability, in terms of number of clients that can be effectively supported by a server, was a major design consideration in Coda but not in LOCUS. Finally, LOCUS did not fully address the problem of conflict detection and resolution.

In the end, the design of Coda borrowed little from LOCUS. The only real similarity was at a high level of abstraction: Coda and LOCUS both used optimistic replication. At the next level of detail, they diverged completely.

3.2 Replica Control

The unit of replication in Coda is a volume. A replicated volume consists of several physical volume replicas that are managed as one logical entity by the system. Individual replicas are not normally visible to users or application programs. All they see is an ordinary file system mounted at `/coda`. This illusion is created by the Venus cache manager on the client. Venus intercepts all file references to `/coda`, discovers which servers hold replicas of a volume, fetches file and directory data as necessary from those servers, and manages cache copies of this data using files in the local file system as containers. Logically, Venus is part of the client operating system. However, as discussed later in Section 9, the actual implementation of Venus resides outside the kernel.

The set of servers that contain replicas of a volume constitutes its *volume storage group* (VSG). For every volume from which it has cached data, Venus keeps track of the subset of the VSG that is currently accessible. This subset is called the *accessible volume storage group* (AVSG). Depending on their network connectivity, different clients may have different AVSGs for the same volume at a given instant. Venus performs periodic probes, typically once every 10 minutes, to detect shrinking or enlargement of AVSGs. Shrinking may also be detected sooner, as a side effect of attempting a server operation.

Coda integrates server replication with caching using a variant of the *read-one*, *write-all* strategy. As Figures 1 and 2 show, this can be characterized as *read-one-data*, *read-all-status*, *write-all*.

3.2.1 *Read Protocol.* In the common case of a cache hit on data known to be valid because of an outstanding callback, Venus avoids contacting the servers altogether. To service a cache miss, Venus obtains data from one randomly selected member of its AVSG known as the preferred server (PS); it also collects version information from the other servers in the AVSG. Venus uses this information to verify that accessible replicas are identical. If the replicas are in conflict, the system call that triggered the cache miss is aborted. If some replicas are stale, they are brought up to date and the fetch operation is restarted. A callback is established with each AVSG member as a side effect of fetching or

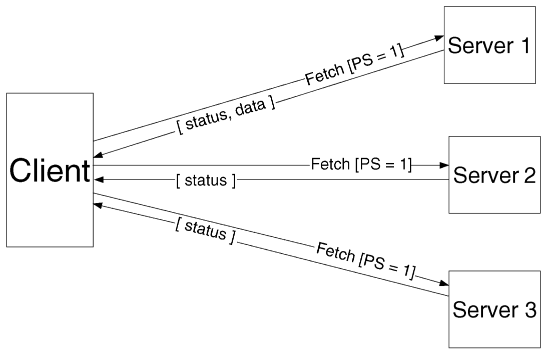

Fig. 1. Servicing a cache miss: the events that follow from a cache miss at the client. Both data and status are fetched from Server 1, which is the preferred server. Only status is fetched from Servers 2 and 3. The calls to all three servers occur in parallel.

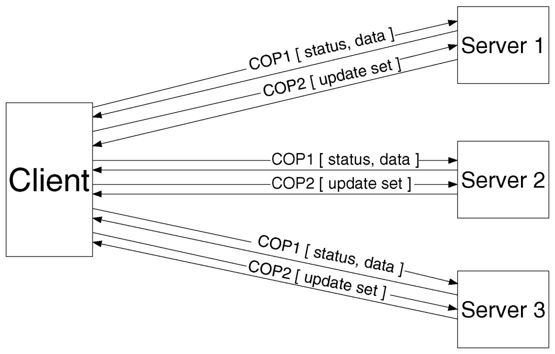

Fig. 2. The two phases of the Coda update protocol. In the first phase, COP1, the three servers are sent new status and data in parallel. In the later asynchronous phase, COP2, the update set is sent to these servers. COP2 also occurs in parallel and may be piggybacked on the next COP1.

validating data. This does not require any communication beyond that shown in Figure 1.

3.2.2 *Update Protocol.* When a file is closed after modification it is transferred in parallel to all members of the AVSG, as shown in Figure 2. Operations that update directories, such as creating a new directory or removing a file, are also written through to all AVSG members. Coda checks for replica divergence before and after update operations. The protocol for an update consists of two phases, COP1 and COP2, where COP stands for "Coda optimistic protocol." COP1 performs the semantic part of the operation, such as transferring file contents, making a directory entry, or changing an access list. COP2 distributes a data structure called the *update set* that summarizes Venus's knowledge of which servers successfully performed the earlier COP1 operation. In the common case of no failures occurring during the protocol, the state of all AVSG members moves forward in lock step.

This update protocol differs from a classical two-phase commit in that it avoids blocking. If a server or network failure occurs during the protocol, the client times out and conservatively assumes that servers from which it has yet to receive a reply were not updated. Similarly, if a server times out on a COP2, it assumes that no other server received the earlier COP1. In both cases, system invariants guarantee that the worst outcome is a false conflict that will be detected when connectivity is restored; under no circumstances are updates ever lost.

Two protocol optimizations are used to improve performance. First, latency is reduced by Venus returning control to the user after completion of COP1 and performing the COP2 asynchronously. Second, network and server CPU load are reduced by piggybacking the asynchronous COP2 messages on subsequent COP1 calls to the same VSG. As a further optimization, Venus contacts AVSG members using a parallel remote procedure call mechanism [Satyanarayanan and Siegel 1990]. The original version of this mechanism was capable of exploiting support for multicast; measurements showed that this support reduced network load but did not lower latency. To restore this functionality, the implementation needs to be upgraded to the current Multicast IP standard.

3.2.3 *Consistency Model.* Coda's consistency model can be informally described using the concept of *accessible universe*. In the absence of failures, the accessible universe is the entire collection of servers and clients in the system. In the presence of failures, the accessible universe of a client is the subset of servers to which it is connected, and the subset of clients to which those servers are connected.

At any instant of time, an `open` on a file at a client sees the result of the most recent `close` on that file in the client's accessible universe. A `close` after updates results in immediate visibility of those updates to any future `open` in the accessible universe, and eventual visibility in the entire universe. In other words, Coda offers one-copy serializability at `open-close` granularity in the accessible universe.

In the limiting case of an isolated client, its accessible universe shrinks to just itself. The above consistency model holds even in this degenerate case because of support for disconnected operation, as discussed in the next section. A formal specification of this unified consistency model for server replication and disconnected operation is presented in an earlier paper [Satyanarayanan et al. 1990].

3.2.4 *Fault-Tolerance.* The correctness of the update protocol requires each AVSG member to ensure the atomicity and permanence of local updates to metadata by COP1. Coda initially used Camelot [Eppinger et al. 1991] to provide transactional support for critical server data structures. Unsatisfactory experience with Camelot led us to reimplement the small subset of its functionality that was relevant to Coda in a lightweight programming library called RVM. Since RVM has been described elsewhere [Satyanarayanan et al. 1993b], we just give a brief overview here.

RVM implements the abstraction of *recoverable virtual memory* [Eppinger 1989]. Updates to data structures in RVM-backed regions of a server's address space are guaranteed to be atomic and permanent. The programming interface is simple: `begin_transaction` initiates a transaction; `set_range` informs RVM that a certain area of virtual memory is about to be modified; `end_transaction` commits a transaction; and `abort_transaction` aborts it. By default, a successful commit guarantees permanence. To reduce commit latency, an application can use a no-flush or "lazy" transaction. Such an application must explicitly force RVM's write-ahead log from time to time to ensure permanence.

Although designed specifically for server replication, RVM has been successfully used for many other purposes on both clients and servers. Its simple design and minimal demands on the operating system have eased the task of porting Coda across a wide range of platforms. RVM has also proved surprisingly versatile in non-Coda applications. Examples include: concurrent language-based garbage collection [O'Toole et al. 1993], transactional distributed shared memory [Feeley et al. 1994], persistent Java [Jordan 1996], and persistent-memory transactional support [Lowell and Chen 1997].

4. DISCONNECTED OPERATION: 1988–1993

Even with optimistic replica control, server replication cannot help if all servers in a VSG crash or if a network failure isolates a client. This concern nagged us from the very beginning of Coda's design, and grew in significance as its implementation progressed. We often experienced correlated server crashes due to software bugs. We also encountered network isolation caused by overloaded or faulty routers or bridges. These observations, together with the publication of Gray's [1986] paper identifying operator error as the leading cause of failures in high availability systems, convinced us that server replication alone was not going to be adequate as a mechanism for high availability in Coda. We had to find some way to reduce the dependence of a client on the network and servers.

Early in 1988, it dawned on us that the caching we were already implementing for performance reasons could also be exploited to improve availability. Successful cache management by Venus implies that the current working set of files is already present on the client, and can be used to service client read requests without contacting servers. For updates, Venus would have to suppress propagation to servers and instead remember these updates for future propagation. This promised to be a simple approach to improving availability in precisely those circumstances where server replication was ineffective. We named this mode of use *disconnected operation* because it is a temporary deviation from normal operation by a client of a shared data repository.

The absence of communication with servers during disconnected operation meant that some reads might be stale and that some updates might be in conflict with updates elsewhere in the system. But we had already accepted these risks by using optimistic replica control for server replication. Although they might occur more frequently because of disconnected operation, these risks did not represent a fundamentally new complication in our design.

4.1 Relevance to Mobile Computing

By coincidence, our formulation of the concept of disconnected operation happened at about the time portable computers were starting to be used by the general public. These early portables had many limitations: they were large and bulky, or they had miniscule CPU, memory, and disk capacity compared to desktops. Yet, we found it instructive to observe how the owner of such a portable managed data from a shared file system like AFS or NFS. In preparation for a trip, she would identify files of interest and download them from the shared file system into the local name space. Upon her return, she would copy modified files back into the shared file system. As we observed this usage pattern, we realized that the user was effectively performing manual caching, with writeback upon reconnection—in other words, she was using a manual implementation of disconnected operation.

We realized that system support for disconnected operation could substantially simplify the use of portable clients. Users would not have to use a different name space while isolated, nor would they have to manually propagate changes upon reconnection. Furthermore, detection of update conflicts and possibly their resolution could be automated. Thus portable machines are a champion application for disconnected operation. It is from this point that the association between Coda and mobile computing began. This association soon enriched Coda's original focus on high availability and forced us to think seriously about the broader implications of mobility for information access.

The use of portable machines also gave us another insight. The fact that people are able to operate for extended periods in isolation suggests that they are competent at predicting their future file access needs. This, in turn, suggests that it is reasonable to seek user assistance in augmenting the cache management policy for disconnected operation. Hence Coda's cache management mechanism relies on a traditional LRU policy by default, but is able to incorporate user advice when available.

In the Coda model of disconnected operation, *involuntary* disconnections caused by failures are no different from *voluntary* disconnections caused by deliberate unplugging of a portable computer. A single mechanism is able to cope with both types of disconnection. Of course, user expectations and the extent of user cooperation may be different in the two cases. Both forms of disconnection are relevant to mobile computing based on wireless networks. Involuntary disconnections can occur due to signal propagation limitations arising from terrain or moving obstacles, or from limited coverage. Voluntary disconnections can occur because a user turns off his transceiver to conserve battery life or, in a military application, to preserve radio silence.

4.2 Relationship to Server Replication

As the numerous benefits of disconnected operation became apparent, we began to question the value of server replication. If disconnected operation is feasible, why is server replication needed at all? Why pay for additional hardware and suffer a more complex update protocol if high availability can be achieved through disconnected operation? Wrestling with these questions clarified our thinking and helped guide further development of Coda. We realized that server replication and disconnected operation were complementary mechanisms, each of whose strengths could alleviate the weaknesses of the other. These strengths and weaknesses are direct consequences of the client-server model.

Systems like Coda make very different assumptions about clients and servers. Clients are like appliances: they can be turned off at will and may be unattended for long periods of time. They have limited disk storage capacity, their software and hardware may be tampered with, and their owners may not be diligent about backing up the local disks. Servers are typically like public utilities: they have much greater disk capacity, they are physically secure, and they are monitored and administered by professional staff.

It is therefore appropriate to distinguish between *first-class replicas* on servers, and *second-class replicas* (i.e., cache copies) on clients. First-class replicas are of higher quality: they are more persistent, widely known, secure, available, complete, and accurate. Second-class replicas, in contrast, are inferior along all these dimensions. The effort needed to preserve the quality of first-class replicas typically results in greater performance degradation than with second-class replicas. From the viewpoint of system scalability, first-class replication is more expensive in hardware cost and performance degradation. However, it increases the chances of unanticipated data accesses being serviced even in the presence of failures.

Only by periodic revalidation with respect to a first-class replica can a second-class replica be useful. The function of a cache coherence protocol is to combine the performance and scalability advantages of a second-class replica with the quality of a first class replica. As discussed in Section 3.2.3, a single consistency model unites server replication and disconnected operation in Coda.

When disconnected, the quality of the second-class replica may be degraded because the first-class replica upon which it is contingent is inaccessible. The longer the client is disconnected, the greater the potential for degradation. Whereas server replication preserves the quality of data in the face of failures, disconnected operation sacrifices quality for availability. Whether to use server replication is thus a trade-off between quality and cost. Since Coda supports nonreplicated volumes, an installation can choose to rely solely on disconnected operation for high availability.

4.3 Implementation

Although simple as an abstraction, disconnected operation turns out to be nontrivial to implement. To support disconnected operation, Venus operates in one of three states: *hoarding*, *emulating*, and *reintegrating*, as shown in Figure 3. Venus is normally in the hoarding state, relying on servers but always alert for possible disconnection. Upon disconnection, Venus enters the emulating state and remains there for the duration of disconnection. Upon reconnection, Venus enters the reintegrating state, resynchronizes its cache with servers, and then reverts to the hoarding state.

4.3.1 *Hoarding.* The key function of the hoarding state is to prepare for disconnection by ensuring that critical objects are cached. Venus combines implicit and explicit sources of information into a priority-based cache management algorithm. The implicit information consists of recent reference history, as in LRU caching algorithms. Explicit information takes the form of a per-client hoard database (HDB), whose entries are pathnames identifying objects

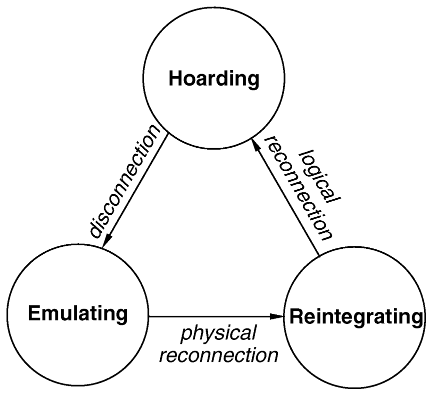

Fig. 3. State transitions for disconnected operation.

of interest to the user at that client. A simple frontend program 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 profiles may be constructed manually or using tools such as SEER [Kuenning and Popek 1997].

4.3.2 *Emulating.* 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 a persistent operation log called the *client modification log* (*CML*).

Venus implements a number of optimizations to reduce the size of the CML. Before a log record is appended to the CML, Venus checks if it cancels or overrides the effect of earlier records. For example, consider an editor that implements the `save` operation by creating a new empty file, copying data into it from memory, deleting the current version of the file, and renaming the new file. In terms of CML operations, this translates to `create` and `store`, followed by an `unlink` when the file is superseded. In this case, all three CML records and the data associated with the `store` can be eliminated. Both trace-driven simulations and measurements of Coda in actual use confirm the importance of log optimizations [Noble and Satyanarayanan 1994; Satyanarayanan et al. 1993a].

4.3.3 *Reintegrating.* Reintegration is performed independently for each volume. During reintegration, Venus propagates changes made in the emulation state and updates its cache to reflect current server state. Propagation is accomplished in two steps. In the first step, the CML of the volume is shipped to the AVSG, and examined there for correctness and consistency. If this check is successful, the servers perform the updates specified in the CML. For `store` operations, empty shadow files are created. Their contents are obtained by servers in the second step of reintegration, called *backfetching*. Further implementation details on disconnected operation can be found in earlier Coda papers [Kistler and Satyanarayanan 1992; Satyanarayanan et al. 1993a].

5. CONFLICT RESOLUTION: 1988–1995

A consequence of using optimistic replication in Coda is the possibility of conflicts arising from updates to partitioned replicas of the same file or directory by two different clients during periods of network failure. The replicas can be server replicas or, in the case of disconnected operation, a cache copy and a server replica. Early in the design of Coda, we established the principles for coping with this problem.

—*No updates should ever be lost without explicit user approval*. It is unacceptable to fail to detect a conflict or to silently ignore one.

—*The common case of no conflicts should be fast*. In other words, conflict detection and handling should have a minimal impact on mainline processing.

—*Conflicts are ultimately an application-specific concept*. For example, unsynchronized updates to a file representing a shared checkbook or an appointment calendar may, or may not, conflict depending on the specifics of the updates. It is therefore necessary to seek application assistance when resolving conflicts.

—*The buck stops with the user*. The goal of automating conflict resolution is only to reduce the frequency with which the user has to intervene. It is therefore acceptable to limit the complexity of automated resolution mechanisms by passing the problem up to the user.

5.1 Balancing Speed and Power

Our overall design based on these principles is a hybrid between a purely *syntactic* approach to conflict detection and handling, and a purely *semantic* one [Davidson et al. 1985]. Syntactic approaches tend to be simple and efficient because they use version information to verify one-copy serializability of partitioned updates. However, they are weak in their ability to resolve conflicts because they lack application-specific knowledge. Conversely, a purely semantic approach such as that of Bayou [Terry et al. 1995] can be powerful but tends to have greater overhead. Coda's approach is to use a syntactic mechanism to confirm the absence of conflicts. Only when this mechanism indicates a possible conflict is a semantic mechanism invoked. This hybrid strategy achieves good performance in the normal case, without sacrificing potency in the exception case.

The server replication protocols described in Section 3.2 maintain and validate version information across accessible server replicas. Similarly, the version checks during the hoarding and reintegration phases of disconnected operation maintain and validate version information between client and server replicas. Together these constitute the syntactic component of Coda's conflict handling approach. If a server detects violation of one-copy serializability on a client operation, it returns a distinct error code. It is the client's responsibility to then trigger an appropriate semantic conflict resolution mechanism. Different semantic mechanisms are used for directory and file resolution.

5.2 Log-Based Directory Resolution

Resolution of directory conflicts is handled entirely by Coda, with no application-specific assistance. There are two reasons for this design decision: first, a directory is a system-defined data type and it is therefore natural to delegate its resolution to the system; and second, a directory plays a structural role in a hierarchical file system, with resolution errors in a directory possibly leading to loss of entire subtrees rooted at that directory. By retaining complete control over directory resolution, Coda can better ensure that critical system invariants are preserved. The semantic component of directory resolution is encapsulated into a module called the *resolution subsystem* on servers. Venus plays no role in directory resolution other than triggering the resolution subsystem.

There are two distinct cases of directory resolution: one after disconnected operation, and the other during connected operation. The first case is simple: each server tries to apply the CML sent by Venus during reintegration to its directory replica, reconciling conflicts as it proceeds. If this attempt reveals an unresolvable conflict, the reintegration of the directory fails and the client's cached copy of the directory is marked in conflict. This conflict is not visible to any other client.

The second case, resolving directory conflicts across divergent server replicas, uses a log-based strategy. Every server replica of a volume is associated with a data structure in RVM known as its *resolution log*. Conceptually, a resolution log contains the entire list of directory mutating operations on a replica since its creation. In practice, of course, logs are of finite length and only the tail is preserved. The size of the log is specified when creating a volume, but can be later adjusted by a system administrator. By associating a resolution log with each volume, Coda ensures that logs are fully self-contained with respect to the information needed to resolve a set of mutually dependent directory conflicts. This is because operands of system calls in Coda may span directories, but not a volume boundary. We have found this small incompatibility with POSIX semantics an acceptable price for the simplification it enables in the implementation.

5.2.1 *Log Growth and Truncation.* When a server receives a directory update from a client, it commits a log entry for the update as part of its COP1 actions. If the later COP2 phase indicates that all VSG members participated successfully in this update, the log entry is deleted since it will never be required in any future resolution. Thus, a volume's resolution log is almost empty when there are no failures; it only contains entries for recent updates still awaiting a COP2. Only if the wait for a COP2 times out does the log start to grow. In that case, truncation of log entries occurs upon future completion of a successful directory resolution involving all VSG members.

In rare cases, a failure may last so long that a resolution log becomes full. A system administrator typically responds to this situation by extending the log. The alternative response is to allow the log to wrap. In that case, Coda loses the ability to transparently resolve those directories whose log entries are overwritten; users will have to manually repair them. Trace-driven simulations [Kumar and Satyanarayanan 1993a], as well as measurements from Coda servers in actual use [Noble and Satyanarayanan 1994] confirm that log growth is modest and easily accommodated by typical server configurations.

5.2.2 *Resolution Protocol.* The resolution protocol is coordinator-driven, with one server in the AVSG acting as coordinator and the others as subordinates. If this protocol detects unresolvable updates on different replicas, all server replicas of the directory are marked in conflict.

The protocol has several phases. The first phase locks volume replicas. The second phase consists of the coordinator collecting resolution logs from the subordinates and merging them with its own. The third phase consists of distributing the merged logs, inferring missed updates at each replica, and applying those updates. The final phase unlocks replicas and verifies success. The protocol is designed to be resilient to subordinate, coordinator, and network failures. It is also idempotent, thus ensuring that reattempting a failed resolution is safe. These issues are discussed further in an earlier paper [Kumar and Satyanarayanan 1993a].

5.2.3 *Merits of Log-Based Approach.* In hindsight, the use of an explicit resolution log at each server has proved to be an excellent design decision. In an early phase of Coda's design, we seriously examined an approach that avoided an explicit log by inferring missing updates from the final states of directory replicas. Unfortunately, the need to disambiguate between removals and creations complicates this strategy. If only one of two replicas contains an entry *A*, how does one decide if *A* was created in the first replica or deleted from the second? Since *A* can be the root of an entire subtree, this ambiguity can cascade to many levels. The only robust solution is to preserve a vestige of each deleted object in the form of a ghost entry. Systems such as Ficus [Guy 1987, 1990; Guy and Popek 1990] that use this approach have to perform distributed garbage collection in order to purge ghosts. The complexity of distributed garbage collection in the face of server and network failures deterred us from using this approach.

Maintaining an explicit log eliminates all doubt: we record history rather than trying to guess it. Reclaiming log space occurs automatically, as side effects of Coda's update and resolution protocols. The space overhead for disambiguation is concentrated in a single per-volume data structure rather than being dispersed. When free space becomes critically low on a server, allowing a resolution log to wrap offers a trivial way to allow continued use of the system. Overall, we have found the simplicity and flexibility of a log-based approach compelling.

5.3 Application-Specific File Resolution

To support semantic resolution of files, Coda provides a framework for installing and invoking customized pieces of code called *application-specific resolvers* (*ASRs*). Each ASR encapsulates knowledge that is specific to its application, including details of its file formats. In contrast to directory resolution, where the bulk of the machinery resides on servers, ASRs are executed entirely on clients.

The main reason for this is to preserve the security model of Coda: allowing arbitrary ASRs to execute on servers would have violated Coda's basic assumption that servers run only trusted software. Executing ASRs on clients has two other benefits: it enhances scalability because no resources are consumed on servers; and it avoids code duplication because much of the supporting machinery needed to execute an ASR is already present at clients but not at servers.

Logically, the Coda ASR mechanism can be viewed as comprising distinct parts: one part responsible for invoking an ASR when needed, a second part pertaining to selection of the correct ASR, and a third part responsible for overseeing the execution of an ASR. The mechanism works as follows. When Venus detects a syntactic conflict on a file, it searches for an ASR using rules specified by the user. If an ASR is found, it is executed on the client. The system call that triggered resolution is blocked while the ASR is executing. If the ASR is successful, the updated file is propagated to servers; the blocked system call then proceeds normally. In all other cases, the file is marked in conflict and a system call error is returned.

Many practical considerations complicate the implementation of this simple concept. Users need to be able to control which ASR is invoked for a specific application. Considerations of security imply the need to restrict the scope of damage caused by an errant ASR. Since failures due to intermittent connectivity are common in wireless networks, ASR execution has to be atomic for easy cleanup. The Coda ASR mechanism addresses these and other related concerns by combining a rule-based approach to ASR selection with transactional encapsulation of ASR execution. Further details can be found in earlier publications [Kumar 1994; Kumar and Satyanarayanan 1993b, 1995].

5.4 Conflict Representation

The process of conflict detection and resolution is transparent to users and applications unless the semantic attempt to fix a conflict fails. In that case, Coda prevents all further attempts to use or modify the object in conflict until a manual repair is performed by a user. This ensures damage containment and limits the incidental consequences of conflicting updates.

A vexing problem we faced early in the implementation of Coda was how to represent an object in conflict, and how to notify a user of the need to repair it. In a system primarily intended for interactive users, such as Windows or Macintosh, the solution would be obvious: just pop up a dialog box on each attempted access or on a timer. However, Coda was designed for an environment where unattended programs are common. Relying on the presence of a human user to receive alerts about conflicts is not acceptable. As an alternative, we considered generating a program exception when an object in conflict is accessed; an application's exception handler could then cope with the problem and eventually alert a user. Alas, the desire to support legacy applications eliminated this approach as well.

The solution we finally converged on was to represent an object in conflict as a dangling symbolic link, as shown in Figure 4(a). Even the most poorly written legacy applications typically cope with failed attempts to open files.

```
$ ls -l
total 364
-rw-r--r--   1 satya    4976 Jun 28 08:42 cevol99.aux
lr--r--r--   1 root       27 Jun 29 11:08 cevol99.bib -> @7f0004e6.0000a52c.0000b7dc
-rw-r--r--   1 satya     528 Jun 28 08:42 cevol99.err
-rw-r--r--   1 satya   87070 Jun 28 08:41 cevol99.mss
-rw-r--r--   1 satya    6937 Jun 28 08:42 cevol99.otl
-rw-r--r--   1 satya  267914 Jun 28 08:42 cevol99.ps
```

(a) Before repair

```
$ls -lR cevol99.bib
total 75
-rw-r--r--   1 satya   26290 Jun 29 11:04 marais.coda.cs.cmu.edu
-rw-r--r--   1 satya   20286 Jun 29 11:03 mozart.coda.cs.cmu.edu
-rw-r--r--   1 satya   26290 Jun 29 11:04 verdi.coda.cs.cmu.edu
```

(b) During repair

```
$ ls -l
total 390
-rw-r--r--   1 satya    4976 Jun 28 08:42 cevol99.aux
-rw-r--r--   1 ras     26290 Jun 29 11:09 cevol99.bib
-rw-r--r--   1 satya     528 Jun 28 08:42 cevol99.err
-rw-r--r--   1 satya   87070 Jun 28 08:41 cevol99.mss
-rw-r--r--   1 satya    6937 Jun 28 08:42 cevol99.otl
-rw-r--r--   1 satya  267914 Jun 28 08:42 cevol99.ps
```

(c) After repair

Fig. 4. Typical sequence of events in manual repair. (a) Directory listing in which the file `cevol99.bib` is in conflict. This name appears to be a dangling symbolic link, whose value is the file's low-level Coda identifier. (b) Shows what happens to `cevol99.bib` while a repair session is in progress. The dangling symbolic link now appears as a directory with one entry for each server replica. The name of a replica is the name of the corresponding server. Note that the replica on the server `mozart` has a different length and modification time from the other two replicas; this is the result of concurrent updates by different clients while `mozart` was partitioned from `verdi` and `marais`. (c) Final state of the directory after a successful repair by user `ras`, who had update privileges on the directory. The file `cevol99.bib` now appears as a normal file.

By forcing an object in conflict to appear as a dangling symbolic link, Venus prevents further accesses while preserving a visual reminder of the conflict in the file name space for user attention. This provides out-of-band information to the user without modifying the file system API. A purist may dislike this solution, but it has worked well in practice.

5.5 Manual Repair

Once an object is marked in conflict, it remains in that state until a user repairs it manually. The two different high availability mechanisms in Coda, server replication and disconnected operation, can lead to different types of conflicts. In the first case, a network partition can exist between two server replicas; clients connected to the two replicas may make updates to the same file or directory during the partition. This results in a *server–server conflict*. In the second case, a disconnnected client may update an object that is also updated on the servers by a connected client. This results in a *local–global conflict*.

In the early years of Coda, server–server and local–global conflicts were handled entirely differently. Based on usage experience, these mechanisms have converged over time. However, some differences still remain. Most important, server–server conflicts are visible to all clients, whereas a local–global conflict is visible only to one client. This reflects the deep distinction Coda makes between first- and second-class replicas.

To handle both types of conflicts, Coda today provides a repair tool that exposes individual replicas of the conflicting object and allows normal nonmutating file system operations to be performed on them. This allows use of standard tools such as `ls`, `diff`, and `emacs`. During a repair session, Venus creates the illusion of an in-place explosion of an object into a read-only directory with a subdirectory per replica. Once a repair is committed, Venus implodes the name space and makes replicas invisible once again. Servers are not involved in the repair process except at the end, when Venus performs a special server operation to commit the result. Figure 4 illustrates this sequence of events. For purposes of access control, servers treat repair operations as updates; hence, any user who can update objects in a directory can also repair those objects and the directory.

Special considerations apply to local–global conflicts. First, the repair tool provides the ability to examine and single-step a CML. This allows the user to replay or discard each operation. Second, Venus keeps a volume in disconnected state (even if physical connectivity has been restored) until all local–global conflicts in it are repaired. This preserves a stable repair context when other clients are using the volume.

5.6 Frequency of Conflicts

The use of optimistic replication in Coda was predicated on the assumption that sequential and concurrent write sharing would be rare in anticipated workloads. Validating this assumption posed a dilemma. Only empirical data from a large-scale deployment in extensive use could confirm that conflicts were rare in practice. Since no previous file system using optimistic replication had been deployed, we could only obtain such data by deploying Coda. On the other hand, the effort to build a deployable system would be wasted if our assumption proved wrong, and frequent conflicts rendered Coda unusable. We needed greater confidence in our assumption before investing the resources to make Coda deployable.

Our solution was based on the recognition that the intended workload for Coda was similar to that of AFS, for which a large-scale deployment already existed at Carnegie Mellon. We therefore instrumented AFS to record write-sharing statistics and performed a worst-case analysis of the resulting data. Based on nearly a year's worth of data, our analysis showed that the probability of two different users modifying the same object less than a day apart was at most 0.0075 [Kistler and Satyanarayanan 1992]. Our experience with the deployed Coda system has since confirmed that conflicts between users are indeed rare. A substantially different usage pattern from ours may, of course, result in more conflicts.

6. WEAKLY CONNECTED OPERATION: 1993–1996

As originally conceived, Coda assumed LAN connectivity. Experience with disconnected operation led us to realize that non-LAN connectivity might be useful. For example, a traveler with a Coda laptop often has modem access via the phone in his hotel room. In addition, non-LAN wireless technologies have been deployed in many cities. We have therefore extended Coda to exploit weak connectivity in the form of intermittent or low-bandwidth networks.

Our goal was to use weak connectivity to alleviate the limitations of disconnected operation. A disconnected client faces many constraints. Its updates are not visible to other clients and vice versa. Cache misses due to unanticipated file accesses may impede progress. Until successful reintegration with servers, updates are at risk from theft, loss, or damage of the client. Update conflicts become more likely as the duration of disconnected operation increases. Finally, cache and RVM space may be exhausted if disconnected operation is prolonged.

Our design is based on a few broad principles.

—*Don't punish strongly connected clients*. It is unacceptable to degrade the performance of strongly connected clients on account of weakly connected clients. This precludes use of a broad range of cache write-back schemes in which a server must synchronously contact a weakly connected client for token revocation or data propagation.

—*Don't make life worse than when disconnected*. Although a minor performance penalty may be acceptable, a user is unlikely to tolerate substantial performance degradation.

—*Do it in the background if you can*. As bandwidth decreases, network usage should be moved into the background whenever possible. This typically improves performance by degrading availability or consistency, lesser evils in many cases.

Supporting weakly connected operation required two major changes to Coda: first, we modified the cache coherence protocol to allow rapid revalidation of a large cache after a period of disconnection; and second, we developed a mechanism called *trickle reintegration* that is frugal in its use of network bandwidth for update propagation.

Coda is now able to adapt transparently to variations in network bandwidth spanning nearly four orders of magnitude, from a few Kb/s to 100 Mb/s. A user is well insulated from this bandwidth variation and also from intermittent connectivity. If the transfer of a large file is interrupted, the client and server retain sufficient state to resume the transfer later.

6.1 Rapid Cache Validation

Cache coherence in Coda is based on callbacks. When a client is disconnected, it can no longer rely on callbacks. Upon reconnection, it must validate all cached objects before use to detect updates at servers. The more aggressively one hoards, the longer revalidation takes. This hurts in an intermittent networking environment because there is little time left for servicing cache misses or propagating updates before the next disconnection. Lazy revalidation avoids this problem, but fails to take advantage of the opportunity to improve consistency.

Our solution preserves correctness but greatly reduces reconnection latency. It is based upon the observation that most cached objects are valid upon reconnection. Clients track server state at two levels of granularity: on individual objects and on entire volumes. When an object is updated, the server increments both the version stamp of the object and that of its containing volume. A client caches volume version stamps at the end of a hoard walk. Since all cached objects are known to be valid at this point, mutual consistency of volume and object state is achieved at negligible cost.

When connectivity is restored, the client first validates its volume stamps. If a volume stamp is valid, so is every object cached from that volume. We batch multiple volume stamps in a single RPC for even faster validation. If a volume stamp is invalid, little can be inferred; objects from that volume must be checked individually. Even then, performance is no worse than in the original scheme.

When a client obtains (or validates) a volume version stamp, a server establishes a volume callback as a side effect. This is in addition to (or instead of) callbacks on individual objects. The server must break a client's volume callback when another client updates any object in that volume. Once broken, a volume callback is reacquired only on the next hoard walk. In the interim, the client must rely on object callbacks. Thus, volume callbacks improve speed of validation at the cost of precision of invalidation, a good tradeoff for typical UNIX workloads [Baker et al. 1991; Mummert and Satyanarayanan 1996; Ousterhout et al. 1985]. Our measurements show volume validation success rates over 97%, each saving over 50 object validations [Mummert et al. 1995].

6.2 Trickle Reintegration

Trickle reintegration propagates updates to servers asynchronously. Supporting trickle reintegration required major modifications to Venus. As shown in Figure 3, reintegration was a transient state. Since reintegration is now an ongoing background process, the transient state has been replaced by a stable one called the *write disconnected* state. Figure 5 shows the new states and transitions in Venus.

Our desire to avoid penalizing strongly connected clients implies that a weakly connected client cannot prevent them from updating an object awaiting reintegration. This situation results in a callback break for that object on the weakly connected client. Consistent with our optimistic philosophy, Venus ignores the callback break. When reintegration is attempted, a conflict may occur. This is handled just as if the client had been disconnected.

As discussed in Section 4.3.2, log optimizations are critical to keeping the CML compact. Trickle reintegration reduces their effectiveness because records may be propagated before a canceling or overriding operation arrives. Our solution, shown in Figure 6, is based on aging. A record is not eligible for reintegration until it has spent a minimal amount of time, called the *aging window* (A), in the CML. Based on trace-driven analysis, A has a default value of 600 seconds [Mummert et al. 1995]. It would be a simple extension to make A adaptive

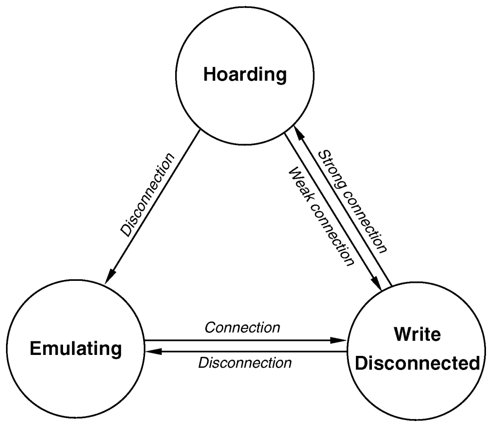

Fig. 5. Per-volume states of Venus, as modified to handle weak connectivity. The state labeled "Write Disconnected" replaces the "Reintegrating" state of Fig. 3. In this state, Venus relies on trickle reintegration to propagate changes to servers. The transition from the emulating to the write disconnected state occurs on any connection, regardless of strength. All outstanding updates are reintegrated before the transition to the hoarding state occurs. Venus is in the hoarding state when strongly connected, and in the emulating state when disconnected.

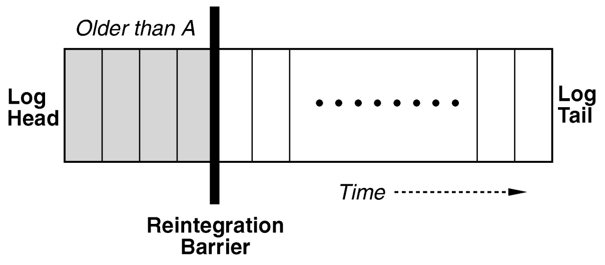

Fig. 6. Typical CML scenario while weakly connected. A is the aging window. The shaded records in this figure are being reintegrated. They are protected from concurrent activity at the client by the reintegration barrier. For `store` records, a shadow copy of the file is created.

based on bandwidth. Trickle reintegration is implemented as an atomic operation, ensuring that a failure leaves behind no state that would hinder a future retry.

To avoid saturating a slow network for an extended period, the reintegration chunk size (C) is chosen so that its expected transmission time is 30 seconds. This corresponds to C being 36 KB at 9.6 Kb/s, 240 KB at 64 Kb/s, and 7.7 MB at 2 Mb/s. Transfer of very large CMLs and files occurs as a series of fragments of size C or less. If a failure occurs, file transfer is resumed after the last successful fragment. The atomicity of a `store` record is preserved in spite of fragmentation because the server does not logically attempt its reintegration until it has received the entire file.

7. ISOLATION-ONLY TRANSACTIONS: 1993–1996

Our early fears about optimistic replication centered on *write/write conflicts*. The solutions described in Section 5 focused exclusively on such conflicts. We ignored *read/write* conflicts because they were a consistency hazard already tolerated by users of timesharing file systems. Unfortunately, disconnected operation widens the window of vulnerability to read/write conflicts and thus increases the likelihood of their occurence.

To address this problem, we extended Coda with a mechanism called *Isolation-Only Transaction* (*IOT*) [Lu and Satyanarayanan 1995; Lu 1996]. Unfortunately, the implementation proved much more complex than we had anticipated. Furthermore, the rest of Coda diverged considerably during the development of this mechanism. At this point, merging support for IOTs into the mainline code would amount to a full reimplementation. Since we do not have the resources for this, the IOT mechanism is only available today as a defunct experimental branch of Coda. Fortunately, our effort was not wasted from a research perspective. Measurements of the IOT implementation [Lu 1996; Lu and Satyanarayanan 1997] helped us confirm that demands on CPU, disk space, and RVM space were modest enough that even a severely resource-constrained Coda client could benefit from the improved consistency of IOTs.

7.1 Design and Implementation

The IOT mechanism focuses only on the *isolation* aspect of the classical *ACID* transactional properties (atomicity, consistency, isolation, and durability) [Eswaran et al. 1976; Gray and Reuter 1993]. IOTs can be viewed as a realization of Kung and Robinson's [1981] *optimistic concurrency control* (OCC) model that has been customized for a mobile computing environment and applied to a distributed file system rather than a distributed database. A disconnected Coda client's cache trivially satisfies the need for a private workspace for storing results of uncommitted transactions in the OCC model, a requirement that has historically limited the use of OCC.

IOTs are upward compatible with POSIX file system semantics. An IOT is a flat sequence of POSIX file system calls bracketed by `begin_iot` and `end_iot` system calls. The scope of an IOT includes all Coda objects accessed or modified by the process that created it and its descendants. A special shell allows legacy applications to be executed within IOTs.

The implementation of IOTs is almost entirely within Venus; little server support is required. The results of executing an IOT are not visible on servers until the IOT ends. If the client is disconnected, a completed IOT remains in pending state. Upon reconnection, Venus first checks whether the IOT's results are consistent with current server state. If consistent, the IOT is committed. Otherwise, it must be resolved using one of four options: automatic reexecution, invocation of an ASR, abort, or manual repair. During resolution, access is provided to both local (client) and global (server) states of objects. The local state of an object is the value last seen by the IOT being resolved.

8. TRANSLUCENT CACHING: 1995–1998

An unquestioned assumption in Coda from its earliest days was that caching be transparent to users. Indeed, transparency is one of the major attractions of caching as a mechanism: neither users nor applications are aware of it. However, usage experience with weakly connected operation convinced us that it was necessary to move away from this ideal. Unless users were alerted to the transition to weak connectivity, we found that they were unpleasantly surprised by long cache miss service times and by delayed propagation of updates.

Our solution was to make caching *translucent* [Ebling et al. 2002]. With translucency, a user has peripheral awareness of her cache state but is not overwhelmed with the details of cache management. By striking a good balance between total transparency and full manual control, translucency offers a better match between user expectations and system behavior. This balance can be varied dynamically based on network connectivity.

Support for translucency was developed in two parts: we modified the handling of cache misses to selectively involve the user; and we implemented a GUI for Venus to compactly convey cache management information to the user. These mechanisms are described in Sections 8.1 and 8.2. We also validated the GUI through a usability study [Ebling 1998].

As in the case of IOTs, the mechanisms for translucency were implemented in an experimental branch of Coda. Because the mainline code has evolved considerably since this branch was created, retrofitting the changes for translucency would amount to a full reimplementation. We have therefore deferred this effort.

8.1 User Patience Model

In most cases, a user would rather be asked whether a file is important before he is made to suffer a long fetch delay. However, just popping up a dialog box on each cache miss is unsatisfactory. First, user reaction time can sometimes exceed fetch delay. Second, cascaded misses can occur as embedded file references in a newly fetched file are encountered. This leads to a flurry of user interactions, which can be more annoying than modest fetch delays. Third, the client may be unattended (for example, when a user starts a compilation and leaves for lunch).

Coda's solution is to suppress user interaction whenever it is reasonably confident of the user's response. The decision-making code is encapsulated in a Venus procedure called the *user patience model* [Mummert et al. 1995] for ease of experimentation and modification. This model balances two factors that intrude upon transparency. At low bandwidths, fetch delays annoy users more than interaction; hence, users are given more control. As bandwidth rises, delays shrink and the annoyance from interaction dominates; hence, more cases are handled transparently. In the limit, at strong connectivity, cache misses are fully transparent.

On a cache miss, Venus first invokes the model with the file name, current bandwidth, and other system information as parameters. If the model suggests that no user interaction is needed, the miss is serviced transparently. Otherwise, Venus attempts to put up a dialog box and to block awaiting a response. Venus will proceed with servicing the miss unless the user suppresses it by responding positively within a timeout period.

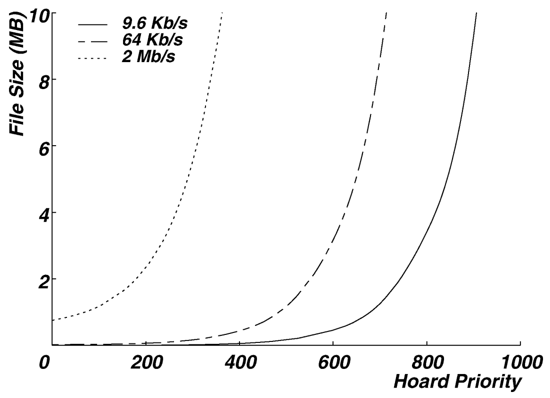

Fig. 7. User patience model. Each curve in this graph expresses patience threshold (τ) as a function of user-specified hoard priority P. Note that τ is given in terms of the largest file that can be fetched at a given bandwidth. User interaction is suppressed and cache misses transparently handled in the region below a curve. These curves correspond to $\tau = \alpha + \beta e^{\gamma P}$, with $\alpha = 6$ seconds, $\beta = 1$, $\gamma = 0.01$.

Designing and validating a user patience model is a challenging and open-ended problem. We have implemented the simple model shown in Figure 7 to demonstrate our approach. This model is based on the conjecture that patience is similar to other human processes such as vision and hearing, whose sensitivity is logarithmic [Cornsweet 1971]. Validating this model remains future work.

8.2 Exposing Cache Management

We faced two challenges in exposing cache management: it is important to provide adequate detail without overwhelming the user, and the mechanism should normally be unobtrusive, yet alert the user promptly when necessary. Our solution is based on the design of dashboard information in automobiles. The demands on the two interfaces are similar: a driver needs to be alerted promptly to potential problems such as an overheated engine or a failed brakelight; but the dashboard should not distract the driver from her focus on the road.

Figure 8 shows the GUI. The factors affecting cache management are grouped into categories and the labels of these catergories are displayed in green, yellow, or red corresponding to normal, warning, and alert states. This interface is physically compact and occupies less than two square inches of screen area, just a few percent of a typical laptop's screen size. A user can obtain more detail by clicking on a label.

9. PORT TO MICROSOFT WINDOWS: 1997–PRESENT

Although most of Coda's evolution has taken place in UNIX, we have recently succeeded in porting the Coda client code to Windows 95/98. This is noteworthy for two reasons. Our implementation preserves a single code base across UNIX and Windows. This simplifies maintenance, minimizes divergence of

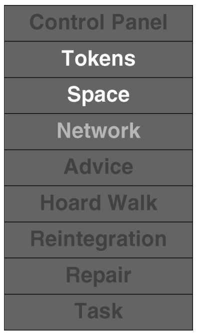

Fig. 8. Indicator lights interface: the red light for “Tokens” indicates that authentication tokens have expired, while the red light for “Space” indicates that some critical resource (such as RVM) is full; the yellow light for “Network” indicates that connectivity is weak.

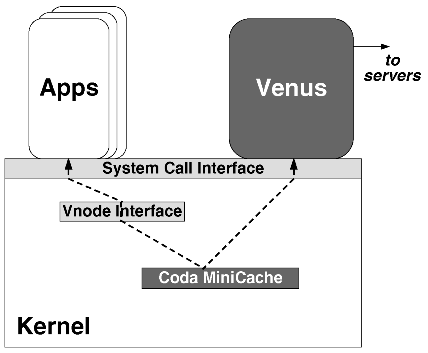

Fig. 9. Structural relationship between applications, the operating system, and Venus. File system calls are routed to the MiniCache via a demultiplexing switch that is conceptually derived from Sun Microsystem’s Vnode interface [Kleiman 1986]. The MiniCache directs calls such as `open` and `close` to Venus, causing the application to be blocked until Venus responds. It routes I/O calls such as `read` and `write` directly to the cache copy of a Coda file, avoiding Venus altogether. The results of calls such as `stat` and `readdir` are cached by the MiniCache.

versions, and ensures that new Coda releases are available promptly in both worlds. In addition, our implementation does not require access to Windows source code.

The key to our success is the Coda client structure shown in Figure 9. This structure was originally developed in 1990 for Mach [Steere et al. 1990]. It has been refined over time as we have gained experience with porting Coda to many different flavors of UNIX. The OS-dependent aspects of Coda fit into a small module called the *MiniCache*, representing just a few percent of total code size. The MiniCache intercepts file references from applications, translates them to a standard representation, and redirects them to Venus. Venus encapsulates the bulk of client complexity, and is written to a standard POSIX interface. Since the implementation of the Win32 API on Windows 95/98 is nonreentrant [Braam et al. 1999a; Pietrek 1995; Schulman 1995], Venus runs as a

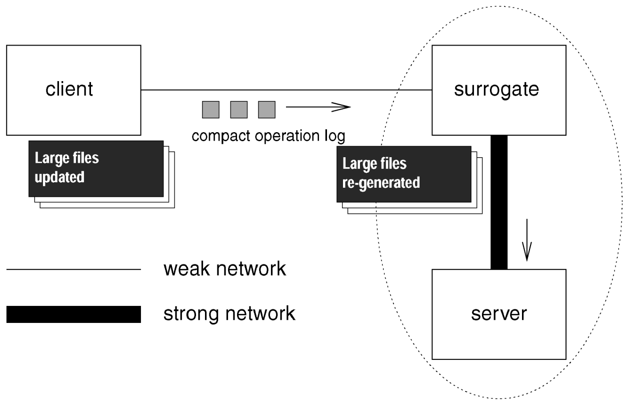

Fig. 10. Update propagation using a surrogate.

DOS Virtual Machine Manager to avoid deadlock. Further details on the Windows 95/98 port can be found in an earlier paper [Braam et al. 1999]. Recently, we have also succeeded in porting Coda to Windows 2000.

10. EXPLOITING SURROGATES: 1997–PRESENT

A typical Coda user has a LAN-connected desktop that sits idle while the user is traveling with a laptop. We have demonstrated that the desktop can be used as a surrogate to speed reintegration from the laptop over a slow network. Our approach, called *operation shipping*, is motivated by certain observations. First, large files are often created or modified by user operations that can be easily intercepted and compactly represented. Second, shipping and reexecuting those operations is often much cheaper than sending large files over a weak network. Our measurements of operation shipping in an experimental branch of Coda show greatly reduced bandwidth usage: from a factor of three to nearly three orders of magnitude [Lee et al. 1999; Lee 2000].

Figure 10 shows how operation shipping is implemented. The surrogate must be of an appropriate machine type, provide an adequate level of security, and offer an execution environment identical to that of the client. To propagate a file, the weakly connected client ships the operation that generated the file to the surrogate. The surrogate reexecutes the operation, validates the regenerated file to ensure that it is identical to the original, and then propagates it to the server. If the regenerated file does not match the original, the system falls back to shipping the original from client to server over the slow connection. Validation with fallback is essential for correctness. Our prototype uses 128-bit MD5 fingerprints [Rivest 1992; Schneier 1996] for validation. Most of our experience has been with application-transparent logging, using a modified version of the `bash` shell. To ensure fault-tolerance, the surrogate performs reexecution as an atomic transaction by operating in the write-disconnected state.

Early experiments revealed that a surprising number of applications produce slightly different output files upon reexecution. The most common cause is embedding timestamps or authoring information in output files. To cope with this problem, we compute and ship a forward error correction (FEC) code with the operation log [Houghton 1997]. If applying the FEC succeeds in eliminating a reexecution discrepancy, the surrogate uses the corrected results. Only if the FEC fails to correct the discrepancy is fallback needed.

We are now investigating whether a surrogate can help improve the servicing of cache misses on a client. This could prove useful when the network topology of Figure 10 is reversed: a client has a low-latency and high-bandwidth connection to a surrogate that has a high-latency connection to a server. We are especially interested in situations where the mobile client is so resource-starved that it is unable to hoard all the data it might access.

Our approach is to prefetch a read-only snapshot of a Coda volume on a surrogate in anticipation of a user's arrival in its neighborhood. Cache misses on that volume from the user's handheld or wearable computer are serviced by the surrogate. There is some loss of consistency, since the read-only snapshot may be stale; but this may be acceptable for slowly changing data where mutual consistency within a volume is more critical than currency. To allow use of an untrusted machine as a surrogate, files in the read-only snapshot can be encrypted and fingerprinted during the cloning process. The encryption keys and fingerprints can be cached even when the corresponding data are much too large. When servicing a cache miss, the client validates the data it receives by decrypting them and verifying their fingerprints. We refer to this approach as *caching trust* [Satyanarayanan 2000].

11. CONTINUING EVOLUTION

In spite of its maturity, Coda remains a vibrant source of new research ideas and development activity. The previous sections have already described the work in progress towards a Windows port and the use of surrogates. At Carnegie Mellon, Coda serves as the mobile file access component of a new research project in pervasive computing called *Aura* [Satyanarayanan 2001; Garlan et al. 2002].

Reengineering the code base is a major ongoing effort. This work spans many activities: restructuring and rewriting for ease of maintenance; effecting robustness and usability improvements; developing debugging tools, regression tests, and performance instrumentation; and improving the documentation of external and internal details of the system. A task of particular importance is improving the security engineering of Coda. Although its architecture has been designed with security in mind, the implementation in this area is incomplete. For example, support for strong encryption needs to be added to clients and servers. Performance engineering is another important task that lies ahead. Compared to a local file system, read performance is acceptable but write performance is slow [Braam and Nelson 1999].

There has been growing interest in Coda in the Open Source community. The number of source code downloads of successive Coda releases, although an imperfect metric, confirms this. There were 22 downloads of the sources of Coda release 5.3.1, published in mid-1999. This increased to 463 downloads for release 5.3.8, published in June 2000; 985 downloads for release 5.3.12, published in January 2001; and 1585 downloads for release 5.3.15, published in June 2001. The sites downloading Coda span a wide range of Internet domains: US and international, commercial and noncommercial. Popular talks and articles on Coda by authors not associated with the development team are beginning to appear [LeBlanc 2000; Lymn 20001a,b]. In 1999, Coda received the *Linuxworld* Editor's Award in the category "File Management."

Although the future is impossible to predict with certainty, all the signs point to a long and healthy life for Coda. It appears to hold continued promise as a tool for serious day-to-day use, as well as a convenient vehicle for experimental research in mobile computing and the emerging field of pervasive computing.

12. LESSONS LEARNED

This section describes the major lessons that I have learned from the evolution of Coda. Some are technical, and others blend technical and nontechnical considerations. Generalizing from a single experience is subject to many obvious limitations. My hope is that the designers of future systems will intelligently combine these lessons with their own experience and understanding of specific system context to arrive at good decisions.

In interpreting this material, one should keep in mind what kind of system Coda represents. It is a system with ambitious and often-expanding functionality that was built in a university. Its primary goal was to validate a high-level research hypothesis, but the validation required the system to be sufficiently robust and complete to support a community of real users. This is in contrast to a system developed as a commercial product, or primarily as a contribution to the Open Source community.

12.1 Fear Not Optimistic Replication

Perhaps the most powerful lesson offered by the Coda experience is that optimistic replication can be usable and effective in a distributed file system based on the client–server model. When the Coda project began, it was very much an open question whether a system permitting updates to any accessible replica (including a cache copy) could be a usable storage repository. Skepticism often greeted the early Coda talks: audiences were worried that frequent update conflicts would render the system unusable. LOCUS, the only previous file system to have advocated optimistic replication, was often used to exemplify the fatal shortcomings of the technique. The decision to avoid optimistic replication in a commercial version of LOCUS [Popek and Walker 1985] was cited as damning evidence that even proponents of the technique lacked faith in it.

Although supporting optimistic replication in Coda proved more difficult than we had anticipated, the difficulties have been overcome. Coda now represents one viable approach to optimistic replication. Alternative approaches have been demonstrated by systems such as Bayou [Demers et al. 1994; Terry et al. 1995], Ficus [Guy 1987, 1990], and Lotus Notes [Collin 1997]. Commercial data reconciliation products such as IntelliSync for Windows 95/98 laptops and HotSync for the PalmOS [Pogue 1998] are now widely used. Today, the use of optimistic replica control in mobile computing environments is so common that it is hard to remember that it was once controversial!

Of course, optimistic replication is not always applicable. Pessimistic replica control techniques continue to be important for applications in which consistency is paramount. The message to take away from our experience is that one should not dismiss optimistic replication simply out of fear that it will be too complex to be efficient or that it will hurt usability unacceptably. Only a careful examination of the application context, balancing the negative impact of lower consistency against the positive impact of higher availability, can determine whether one should use pessimistic or optimistic replication. In many real-world contexts, the enduring benefits of high availability easily outweigh the occasional pain of having to cope with a conflict.

12.2 Real Systems Need Real Users

The impact of real use of Coda cannot be overstated. Earlier sections of this paper have identified many instances where usage experience was instrumental in motivating a new capability or an important design change. For example, Section 3.2.4 mentioned how first-hand experience with Camelot in Coda led to the design and implementation of RVM. Section 5 described how the handling of conflicts was influenced by usage experience. The use of dangling symbolic links to represent conflicts (Section 5.4), and the unification of techniques for handling server–server and local–global conflicts (Section 5.5) both arose from dissatisfaction with earlier approaches. It was usage experience that enabled us to identify the mechanisms most valuable for weakly connected operation; without it, we would not have recognized the need for rapid cache validation (Section 6.1). Recognizing the need for translucent caching (Section 8) was also a direct result of experience with weakly connected operation.

A small group of faculty and students began implementing Coda in early 1991. Since that time, a user community has been in continuous existence at Carnegie Mellon. Its size has fluctuated over the years, ranging from about 40 users at its maximum to a handful at its minimum. A detailed empirical study of this community's use of Coda was reported in an earlier paper [Noble and Satyanarayanan 1994]. The degree of dependence on Coda has at all times been significant enough that we have had to pay careful attention to robustness and correct operation. Over the last few years, Coda has also been deployed outside Carnegie Mellon. The resulting usage feedback has further helped in improving the system.

Gaining usage experience is not easy. It requires deployment of an experimental system and the recruitment of a user community willing to brave the pains of an immature system. This is an especially difficult challenge when persistence is involved. A bug in a file system or database can result in lost or corrupted data, possibly destroying many hours or days of work by a user. Rebooting or power cycling does not fix the problem, as is often possible with other experimental systems. Even supportive users will not entrust valuable data to an experimental file system unless they are confident that the development team will do everything possible to promptly recover from crashes. Indeed, there were many occasions early in the life of Coda when heroic rescue efforts by the development team were necessary.

Sustaining the commitment and effort needed for deployment is particularly difficult in an academic environment. The work is hard and the tangible rewards, such as publications, are few. It is not surprising that very few academic research projects in the past decade have followed through to deployment. Yet, as our experience has shown, the long-term benefits of deployment and use are substantial. Coda would not be the system it is today without the hard-won insights derived from real use.

12.3 Timing is Everything

Support for disconnected operation has proved to be the most influential aspect of Coda. In hindsight, it is easy to see that this is largely due to lucky timing; we happened to be at the right place and time with the right ideas. When first described in October 1991, the concepts and terminology introduced by our work addressed a problem whose importance was already beginning to be recognized by the research community. Fortunately, there was no established or competing viewpoint or system design that our ideas had to displace. Replication, the classic approach to high availability, was not rejected by Coda; rather, it was embraced in the form of server replication. Disconnected operation was offered as an enhancement, not a substitute.

Timing also played a major role in associating Coda with mobile computing in the public mind. The appearance of our 1991 paper coincided with the earliest availability of portable machines that had sufficient resources to be credible Coda clients yet weighed no more than six to eight pounds, the heaviest that a user is typically willing to carry. Mobile computing along the lines advocated by Coda thus became an immediate reality, not just a distant possibility. Today, Coda and disconnected operation are most commonly associated with mobile computing rather than the broader issues of high availability that motivated them. Mobile computing has thus proved to be a "champion application" for Coda: that is, an application domain where the merits of a new technology have such overwhelming benefits that they overshadow the cost and inconvenience of adopting untried technology and suffering its shortcomings until they are overcome.

12.4 Beware the Long Software Tail

From a software engineering viewpoint, the evolution of Coda does not neatly fit either of the best-known development models: the waterfall model or the rapid prototyping model [Somerville 1989]. In hindsight, the best characterization is *pipelined development of capabilities*. A "capability" represents a major functional enhancement such as server replication, disconnected operation, or log-based directory resolution. The pipeline metaphor is appropriate because a snapshot of the system at any time would have revealed different capabilities in different stages of progress.

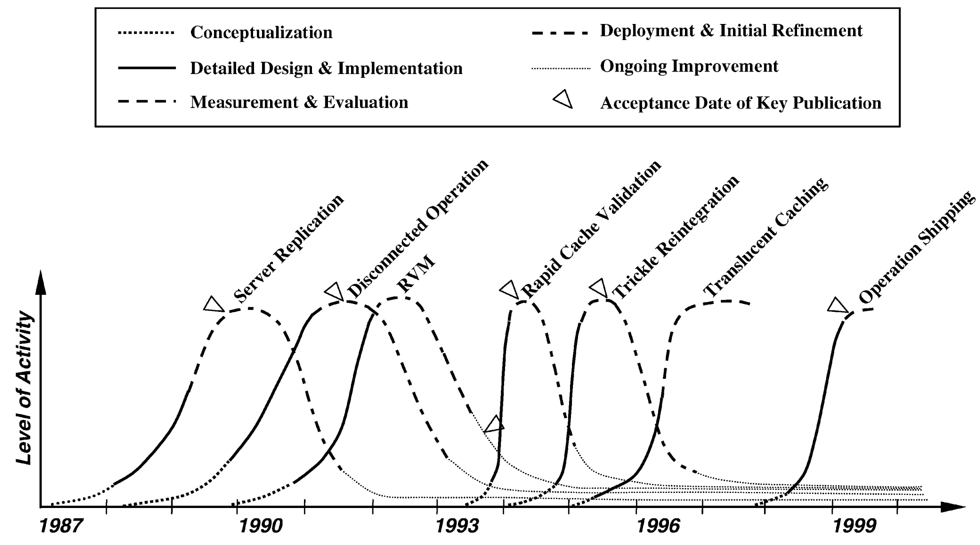

Fig. 11. Capabilities in Coda developed over time. Note that translucent caching and operation shipping were never deployed. To avoid making the figure too busy, efforts such as directory resolution, ASR support, IOT support, porting, and reengineering have been omitted. The triangle on each capability indicates the date of final acceptance of its key research publication. We show acceptance date rather than publication date because it is a more accurate indicator of when reporting was completed.

The existence of a user community dependent on Coda discouraged disruptive or incompatible changes. Each capability typically went through the stages of

—*conceptualization*, identifying the importance of the new capability and how it fits into existing Coda functionality;

—*detailed design and implementation* in an experimental branch of the code;

—*measurement and evaluation* of the implementation through controlled experiments;

—*research publication* in a conference or journal, reporting on the design, implementation, and evaluation;

—*deployment and initial refinement* based on usage experience. This step includes integration of the capability into the production version of the source code; and

—*ongoing improvement* to enhance robustness, performance, and usability. This stage is driven by discovery of bugs and unexpected feature interactions as new users and new capabilities are added to the system. It continues the process of refinement begun in the previous stage.

Figure 11 illustrates this development process. The *Y*-axis gives an approximate measure of the time-relative level of effort applied to a particular capability by the Coda development team; it is a meaningful measure within a capability, but not across capabilities. The *X*-axis depicts the years since the beginning of the project; time comparisons are meaningful both within and across capabilities. The type of a line segment indicates the stage of development of a capability. Except for RVM, all the capabilities followed the sequence of stages listed earlier. In the case of RVM, deployment preceded evaluation.

A major surprise of Figure 11 is the astonishing length of the last stage. Debugging continues to this day in parts of Coda long considered "done." The resilient nature of the system makes the discovery of subtle load-induced bugs especially difficult. The importance of this long "software tail" has been the subject of recent interest in the software engineering community [Rajlich and Bennett 2000]. The quality of talent needed to make effective changes in the software tail is no less than in the earlier, higher-visibility stages. In fact, more talent is needed because later changes have to preserve the architectural coherence of the system. As code stabilizes, it is often possible to shrink its size through careful recoding. At this point in Coda's life, we typically measure progress by the number of lines of code deleted rather than added.

12.5 Hail Moore's Law, the Savior of Portability

It is unusual for a complex piece of system software, such as a file system, to remain portable across many diverse operating systems and many generations of hardware. Coda was originally implemented in the Mach 2.5, and the first eight years of its evolution occurred in Mach. Although nontrivial, the move away from Mach was much less painful than we had feared. The client structure shown in Figure 9 was an important contributor to this smooth transition. That structure has a small and simple in-kernel MiniCache to provide OS-specific support; more complex code is encapsulated in a platform-independent, user-level Venus.

In the early years of Coda, the performance impact of this client structure was high. Even a simple system call, such as open on a cached file, requires a redirection: from application to kernel, and thence to Venus; the reply also requires redirection. We were tempted on many occasions to move Venus into the kernel to reduce this overhead. Indeed, Coda's ancestor AFS chose exactly this path in moving from AFS-2 to AFS-3 [Satyanarayanan 1990]. The fact that Coda was a university research project helped us resist this temptation. Ease of experimentation, ease of debugging, and ease of educating new team members in the internals of Coda were greater concerns in our mind than performance.

Over time, the overhead due to redirection has become less of a concern. The 14 years from 1987 to 2001 represent about eight generations of hardware by Moore's law. Processors are so much faster that the cost of redirection is much less noticeable today. Moving Venus into the kernel is no longer attractive.

The evolution of Coda has overlapped the research community's interest in microkernels and other modular operating system structures. Many projects such as Mach 3.0 [Rashid et al. 1989], Exokernel [Engler et al. 1995], SPIN [Bershad et al. 1995], and Vino [Seltzer et al. 1996] engaged the attention of leading OS researchers from the late 1980s to mid-1990s. Those projects took a principled, top-down view of OS decomposition and developed general mechanisms for modularizing OS functionality. One can view Figure 9 as Coda's approach to modularizing one important part of OS functionality. Although less general than a microkernel or exokernel structure, it has the merit of being shaped by genuine need rather than ideology.

Throughout the evolution of Coda we have also tried to avoid dependence on functionality that is not widely supported on many OS platforms. Today, POSIX defines this interface. We had both philosophical and pragmatic reasons for this self-imposed constraint. Since our goal was to show that a highly available distributed file system could be build out of commodity hardware and widely accepted software standards, exploiting specialized OS support would have weakened our validation. In addition, such functionality is often not robust: it is not stressed enough in real use to solidify the implementation. The replacement of Camelot by RVM, which depends only on POSIX [Satyanarayanan et al. 1993b], is perhaps the best example of Coda's commitment to portability.

Another example is our use of a portable RPC package based on the `socket` interface. RPC performance was a major topic of interest in the research community in the late 1980s and early 1990s. Using custom-designed kernel support, intermachine round-trip RPC times in the neighborhood of a few milliseconds were demonstrated in systems such as Firefly [Schroeder and Burrows 1990], Amoeba [van Renesse et al. 1988], V [Cheriton 1988], and Sprite [Ousterhout et al. 1988]. In contrast, Coda's round-trip RPC time was nearly an order of magnitude worse.

Rather than sacrificing portability, our response was to reduce the frequency of RPCs: a strategy captured by the slogan "The fastest RPC is the one you don't make." Techniques such as aggressive client caching, callback-based consistency, trickle reintegration, and volume callbacks lower the frequency of client–server communication and thus reduce the impact of poor RPC performance. Over time, the improvement in processor performance through Moore's law has contributed to modest improvements in RPC performance; typical round-trip time is a few milliseconds today. Since disk and network latency have not improved as much over the years, the impact of using a portable RPC package is more acceptable today.

12.6 Code Reuse is Bittersweet

One of the first decisions facing any systems project is whether to implement the new system from scratch or to extend an existing system. Each approach has well-known merits and shortcomings. In the case of Coda, reusing existing code appeared to be the obvious choice. We had an excellent starting point in AFS-2, since it was already in serious use and addressed the same kinds of workload and application domain as Coda.

In hindsight, our decision has proved to be a wise one. The starting code base represented nearly 15 high-quality man-years of development effort on AFS-1 and AFS-2. It allowed us to focus on high availability immediately, and to quickly come to grips with implementation issues in server replication. Using AFS-2 was thus critical to the success of Coda. However, it had two negative consequences that we did not foresee in 1987. These relate to legal encumbrance and code longevity.

By mid-1991 there was considerable interest in Coda from other researchers and from companies. Although we were eager to freely distribute Coda, we discovered that it was legally viewed as “derived code” and could not be distributed without the permission of IBM, the owner of AFS-2. The long and frustrating process of obtaining this permission was made more difficult by the fact that we could offer no tangible benefit in return. To IBM’s credit, our request was eventually honored. Between 1992 and 1995, the code was distributable for research and educational purposes but an explicit license had to be signed by each recipient. After a further round of negotiation in late 1995, IBM finally released control of Coda. Only then were we able to freely distribute Coda through the Internet.

The long delay in making Coda freely available outside Carnegie Mellon had a negative effect on its external impact. Unlike systems such Kerberos, X Windows, and Linux whose external use took place relatively early in their evolutions, Coda could not profit from the energy and talent of external adopters and contributors until late in its life. Although Coda is now gaining visibility in the Open Source community, earlier external distribution would have enabled it to play a more prominent and influential role in shaping the growth of that community.

The second negative consequence of using AFS-2 relates to obsolete design and coding assumptions. Although there is virtually no unmodified AFS-2 code left in Coda, the original code channeled our implementation along paths that seem anachronistic today. For example, in typical UNIX implementations of the mid-1980s, a process could use relatively few file descriptors (20 was a common limit). Using a file descriptor per client was clearly not a viable strategy for an AFS-2 file server process. As another example, typical UNIX kernels of the mid-1980s could support relatively few TCP connections. Allocating one or more open TCP connection per client was therefore not feasible for a server hoping to support a hundred or more clients. Typical disk sizes of that era were 100 to 200 Mb, whereas they are 10 Gb or more today, far beyond the limit of 32-bit addressing. It is only as we reengineer Coda that we are unearthing and fixing these long-forgotten assumptions. Since they often have a long reach, the fixes are a lot of work.

On balance, the use of AFS-2 has been beneficial but expensive for Coda. The lesson to extract from our experience is to be aware that code reuse has many subtle consequences that are not immediately apparent. In particular, one should be more sophisticated than we were about licensing issues and should ensure the ability to freely distribute derivatives before investing the energy to extend existing code. Once energy has been invested, it is too late to be in a favorable negotiating position.

12.7 System Administration Gets No Respect

The most compelling reason for using a distributed file system is the separation of concerns that it provides. Users can focus exclusively on access and use of files, confident that mundane but necessary tasks such as backup, server load balancing, capacity growth, and preventive maintenance will be competently handled by well-trained operational staff. A good design allows a small administrative staff to meet the needs of a large user community. Such a design can reduce the total cost of ownership, a metric of growing importance as hardware costs decline. Ease of system administration has been an explicit goal of many distributed file systems, including AFS [Satyanarayanan 1990] and Frangipani [Thekkath et al. 1997]. This separation of concerns suggests that user- and administration-centric mechanisms should receive equal attention in distributed file system design; they are both equally important to the overall success of a deployed system.

Unfortunately, our experience has been that it is far easier to motivate effort and to demonstrate innovation in the user-centric aspects of Coda such as disconnected operation and bandwidth adaptation. It is much harder to motivate effort in administration-centric development, even though lip service is paid to its importance. The absence of good metrics to quantify progress may be an important factor, as suggested by Hennessy [1999]. Today, it is Coda's administration-centric mechanisms that are most in need of improvement. A production-quality system will require development of a full range of robust tools for system administration, work for which there are few tangible rewards in an academic research environment.

12.8 Short Projects Never Die

In the era of "Internet time," a computer systems research project that lasts well over a decade is an oxymoron. Most research projects in universities are of less than half that duration. As mentioned in Section 2, we ourselves began Coda in the belief that it was just a footnote to AFS and only expected it to last two or three years. How has Coda managed to generate sustained interest, activity, and results for so long?

The primary reason for Coda's longevity has been its ability to generate exciting yet attainable research goals in a timely manner. In most cases, these goals were not visible earlier, but came into view as the project moved forward. In the beginning, when we worked on server replication, we had not even identified the concept of disconnected operation. Only when we were in the midst of implementing server replication did the idea of disconnected operation strike us. Once we were on the verge of success in implementing disconnected operation, we asked ourselves, "Now that we can handle the case of zero bandwidth, how can we make life better with a little bandwidth?" This led to the work on bandwidth-adaptive, weakly connected operation. That, in turn, opened the doors to new research opportunities, and so on. On more than one occasion, Coda has managed to stay alive by revealing new research possibilities just in the nick of time.

Long project life has some drawbacks. First, funding sources are often reluctant to invest in something that they perceive as "old." It takes considerable effort to make the case that Coda is being used only as a vehicle, and that the research itself is new. Second, it is hard to change early stereotypes. More than one person has remarked, “I have read *the* Coda paper,” meaning the key Coda paper on disconnected operation. Their concept of Coda is frozen at that stage, and it is often an uphill task to explain the significance of the evolution that has happened since.

A more savvy strategy for marketing and publicity would have been to choose distinct project names for the different phases of Coda’s evolution. This would have preserved the continuity of work, while giving the illusion of a sequence of relatively short, well-defined projects. However, such a strategy would have been intellectually dishonest because it would have hidden one of the major strengths of Coda: its graceful and seamless integration of the many capabilities described in this article into a single system.

A secondary reason for Coda’s longevity has been the near-absence of commercial systems with similar capabilities. IBM Transarc, the custodian of AFS, would have been the most likely candidate to build a commercially supported distributed file system along the lines explored by Coda. But DFS [Kazar et al. 1990], Transarc’s successor to AFS, ignored the problems of high availability and mobility. Sun Microsystem’s NFS [Sandberg et al. 1985], another possible candidate, has remained relatively stagnant over its entire life. The IntelliMirror component of Microsoft’s Windows 2000 file system is the only commercial product to have absorbed ideas from Coda. On Open Source platforms, Coda continues to be the most advanced distributed file system available today. On a lighter note, perhaps the lesson to take away from Coda’s longevity is to avoid project names that imply a short duration!

13. CONCLUSION

Providing scalable, secure, and highly available access to shared data remains an important requirement for enterprise-scale computing. Meeting this requirement is hard enough with stationary users and a wired network; it becomes a difficult challenge with mobile users and wireless networks.

Coda can help meet this challenge in several distinct ways. First, it can serve users directly. This was Coda’s original design goal, and is clearly attainable even though the system is not yet of production quality. Second, Coda can serve as the data access layer of mobile devices customized to specific applications. There are many hooks in Coda for customization: ASRs, hoard profiles, policies for translucent caching, mapping of application data to file system structure, and so on. By using Coda as a substrate, the development effort for such customized solutions can be reduced. Third, the ideas explored and validated by Coda can be used in new contexts and implementations. As mentioned earlier, Microsoft has used this approach in IntelliMirror. Other examples include the Caubweb disconnectable browser [LoVerso and Mazer 1997] and the Inter-Mezzo file system [Braam et al. 1999b].

The evolution of Coda bears witness to the power of the deceptively simple question posed at the beginning of this paper: *Can the virtues of AFS be preserved, while alleviating its vulnerability to failures?* When we asked this question in 1987, we had no idea that answering it positively would involve so much work, require so much research, or take so long. But the effort has not been in vain. From a scientific as well as a practical viewpoint, Coda has been a valuable investment that will pay rich dividends well into the future.

ACKNOWLEDGMENTS

Although this article has a single author, it describes the collective effort of many talented people who have worked very hard over many years. Without their creative energy and commitment to Coda there would have been no paper to write. I would like to express my deepest appreciation and gratitude to the following people for their technical contributions to Coda: Bob Baron, Peter Braam, Michael Callahan, Maria Ebling, Jason Flinn, Jan Harkes, Hiroshi Inamura, James Kistler, Anders Klemets, Masashi Kudo, Puneet Kumar, Yui-wah Lee, Qi Lu, Hank Mashburn, Lily Mummert, Philip Nelson, Brian Noble, Adrian Pavlykevych, Henry Pierce, Josh Raiff, Marc Schnieder, Ellen Siegel, Shafeeq Sinnamohideen, David Steere, and Remington Svarcas. In addition, I would also like to thank a number of Coda users whose constructive feedback has been particularly helpful: David Andersen, David Eckhardt, Tetsuro Muranaga, Owen LeBlanc, Brett Lymn, Dushyanth Narayanan, Stefen Neumann, Ivan Popov, Henry Rowley, and Greg Troxel. Many of the individuals above read early drafts of this article and helped to improve it. Bill Tetzlaff, Dejan Milojicic, and the anonymous journal reviewers also offered valuable suggestions to improve the presentation.

Major support was also provided by IBM, Digital Equipment, and Intel. Additional support was provided by AT&T, Bellcore, General Electric, Hughes Research, Novell, Sun Microsystems, and Xerox. The views and conclusions in this paper are those of the author, and do not represent the opinions, express or implied, of the U.S. Government, NSF, DARPA, USAF, USN, Carnegie Mellon, or any of the corporate sponsors listed above.

REFERENCES

Babaoglu, O. and Marzullo, K. 1993. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, ed., *Distributed Systems*, Addison Wesley, Reading, Mass., Chapter 4.

Baker, M. G., Hartmann, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K. 1991. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, Calif., Oct.).

Bartlett, J., Garcia, D., Gray, J., Horst, B., Lenoski, D., McGuire, D., and Worsencroft, K. 1988. *Fault Tolerance in Tandem Computer Systems*. Tandem Computer Corp.

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Chambers, C., and Eggers, S. 1995. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (Copper Mountain, Co., Dec.).

Borg, A., Blau, W., and Graetsch, W. 1989. Fault tolerance under Unix. *ACM Trans. Comput. Syst. 7*, 1 (Feb.).

Braam, P. J. and Nelson, P. A. 1999. Removing bottlenecks in distributed filesystems: Coda & InterMezzo as examples. In *Proceedings of Linux Expo 1999* (Raleigh, N.C., May).

Braam, P. J., Callahan, M. J., Satyanarayanan, M., and Schnieder, M. 1999. Porting the Coda file system to Windows. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference* (Monterey, Calif., June).

Braam, P. J., Callahan, M. J., and Schwan, P. 1999. The InterMezzo filesystem. In *The Perl Conference 3, O'Reilly Open Source Convention* (Monterey, Calif., August).

Cheriton, D. R. 1988. The V distributed system. *Commun. ACM 31*, 3 (March).

Collin, S. 1997. *A Complete Guide to Lotus Notes 4.5*. Digital Press, Maynard, Mass.

Cornsweet, T. N. 1971. *Visual Perception*. Academic Press.

Davidson, S. B., Garcia-Molina, H., and Skeen, D. 1985. Consistency in partitioned networks. *ACM Comput. Surv. 17*, 3 (Sept.).

Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M., and Welch, B. 1994. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the Workshop on Mobile Computing Systems and Applications* (Santa Cruz, Calif., Dec.).

Ebling, M. R. 1998. Translucent cache management for mobile computing. Phd Thesis, School of Computer Science, Carnegie Mellon University, March.

Ebling, M. R., John, B. E., Satyanarayanan, M. 2002. The importance of translucence in mobile computing systems. *ACM Trans. on Comput.-Human Interaction. 9*, 1 (March), pp. 42–67.

El Abbadi, A. and Toueg, S. 1989. Maintaining availability in partitioned replicated databases. *ACM Trans. Database Syst. 14*, 2 (June).

Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (Copper Mountain, Co., Dec.).

Eppinger, J. L. 1989. Virtual memory management for transaction processing systems. PhD Thesis, School of Computer Science, Carnegie Mellon University, February.

Eppinger, J. L., Mummert, L. B., and Spector, A. Z. 1991. *Camelot and Avalon*. Morgan Kaufmann, San Mateo, Calif.

Eswaran, K., Gray, J., Lorie, R., and Traiger, I. 1976. The notion of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov.), 624–633.

Feeley, M. J., Chase, J. S., Narasayya, V. R., and Levy, H. M. 1994. Integrating coherency and recovery in distributed systems. In *Proceedings of the First Usenix Symposium on Operating System Design and Implementation* (Monterey, Calif., Nov.).

Fidge, C. J. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Eleventh Australian Computer Science Conference* (University of Queensland, Australia, Feb.).

Garlan, D., Siewiorek, D., Smailagic, A., Steenkiste, P. 2002. Project Aura: towards distraction-free pervasive computing. *IEEE Pervasive Computing 1*, 2 (April–June).

Gifford, D. K. 1979. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec.).

Gray, J. 1986. Why do computers stop and what can be done about it? In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems*.

Gray, J. 1997. Windows NT to the max—Just how far can it scale up? In *The USENIX Windows NT Workshop Proceedings* (Seattle, Wash., August). Invited Talk Slides available at //www.usenix.org/publications/library/proceedings/usenix-nt97/presentations/index.html.

Gray, J. and Reuter, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, San Mateo, Calif.

Guy, R. G. 1987. A replicated filesystem design for a distributed Unix system. Master's Thesis, Department of Computer Science, University of California, Los Angeles.

Guy, R. G. 1990. Ficus: A very large scale reliable distributed file system. PhD Thesis, Department of Computer Science, University of California, Los Angeles.

Guy, R. G. and Popek, G. J. 1990. Reconciling partially replicated name spaces. Tech. Rep. CSD-900010, University of California, Los Angeles, April.

Harrison, E. S. and Schmitt, E. J. 1987. The structure of System/88, a fault-tolerant computer. *IBM Syst. J. 26*, 3.

Hennessy, J. 1999. The future of systems research. *IEEE Computer 32*, 8 (August).

Herlihy, M. P. 1986. A quorum-consensus replication method for abstract data types. *ACM Trans. Comput. Syst. 4*, 1 (Feb.).

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

Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M.J. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1 (Feb.).

Jordan, M. 1996. Early experiences with persistent Java. In M. Jordan and M. Atkinson, eds., *Proceedings of the First International Workshop on Persistence and Java* (Drymen, Scotland, Sept.).

Kazar, M., Leverett, B., Anderson, O., Apostolides, V., Bottos, B., Chutani, S., Everhart, C., Mason, W., Tu, S., and Zayas, E. 1990. Decorum file system architectural overview. In *Proceedings of the Summer 1990 Usenix Conference* (Anaheim, Calif., June).

Kistler, J. J. and Satyanarayanan, M. 1992. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb.).

Kleiman, S. R. 1986. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Summer Usenix Conference Proceedings*.

Kuenning, G. H. and Popek, G. J. 1997. Automated hoarding for mobile computers. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint-Malo, France, Oct.).

Kumar, P. 1994. Mitigating the effects of optimistic replication in a distributed file system. PhD Thesis, School of Computer Science, Carnegie Mellon University, December.

Kumar, P. and Satyanarayanan, M. 1993a. Log-based directory resolution in the Coda file system. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems* (San Diego, Calif., Jan.).

Kumar, P. and Satyanarayanan, M. 1993b. Supporting application-specific resolution in an optimistically replicated file system. In *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems* (Napa, Calif., Oct.).

Kumar, P. and Satyanarayanan, M. 1995. Flexible and safe resolution of file conflicts. In *Proceedings of the USENIX Winter 1995 Technical Conference* (New Orleans, La., Jan.).

Kung, H. T. and Robinson, J. 1981. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June), 213–226.

LeBlanc, O. 2000. Coda. In *Linux 2000 UK Linux Developers Conference* (Hammersmith, UK, July). Available at http://www.ukuug.org/events/linux2000/speaker-ol.shtml.

Lee, Y. 2000. Operation-based update propagation in a mobile file system. PhD Thesis, Department of Computer Science and Engineering, The Chinese University of Hong Kong, January.

Lee, Y., Leung, K. S., and Satyanarayanan, M. 1999. Operation-based update propagation in a mobile file system. In *Proceedings of the USENIX Annual Technical Conference.* (Monterey, Calif., June).

Long, D. E. 1990. Analysis of replication control protocols. In *Proceedings of the IEEE Workshop on Management of Replicated Data* (Houston, Tex, Nov.).

LoVerso, J. R. and Mazer, M. S. 1997. Caubweb: Detaching the web with Tcl. In *Proceedings of the Fifth Annual Tcl/Tk Workshop*. Boston, Mass. July).

Lowell, D. E. and Chen, P. M. 1997. Free transactions with Rio Vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint-Malo, France, Oct.).

Lu, Q. 1996. Improving data consistency in mobile file access using isolation-only transactions. PhD Thesis, School of Computer Science, Carnegie Mellon University, May.

Lu, Q. and Satyanarayanan, M. 1995. Improving data consistency in mobile computing using isolation-only transactions. In *Proceedings of the Fifth IEEE Workshop on Hot Topics in Operating Systems* (Orcas Island, Wash., May).

Lu, Q. and Satyanarayanan, M. 1997. Resource conservation in a mobile transaction system. *IEEE Trans. Comput. 46*, 3 (March).

Lymn, B. 2001a. What to do when the server doesn't serve—Using Coda. *SysAdmin 10*, 4 (April).

Lymn, B. 2001b. Coda—The disconnectable file system. *SysAdmin 10*, 7 (July).

Mummert, L. B. and Satyanarayanan, M. 1996. Long-term distributed file reference tracing: Implementation and experience. *Softw. Pract. Exper. 6*, 6 (June).

Mummert, L. B., Ebling, M. R., and Satyanarayanan, M. 1995. Exploiting weak connectivity for mobile file access. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (Copper Mountain, Co., Dec.).

Noble, B. and Satyanarayanan, M. 1994. An empirical study of a highly-available file system. In *Proceedings of the 1994 ACM Sigmetrics Conference* (Nashville, May).

O'Toole, J., Nettles, S., and Gifford, D. 1993. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, N.C., Dec.).

Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M., and Welch, B. 1988. The Sprite network operating system. *Computer 21*, 2 (Feb.).

Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M., and Thompson, J. 1985. A trace-driven analysis of the 4.2BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating System Principles* (Orcas Island, Wash., Dec.).

Paris, J.-F. 1986. Voting with witnesses: A consistency scheme for replicated files. In *Proceedings of the Sixth IEEE International Conference on Distributed Computing Systems* (Cambridge).

Parker, D. S. Jr., Popek, G. J., Rudisin, G., Stoughton, A., Walker, B. J., Walton, E., Chow, J. M., Edwards, D., Kiser, S., and Kline, C. 1983. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng. SE-9*, 3 (May).

Pietrek, M. 1995. *Windows 95 System Programming Secrets*. IDG Books.

Pogue, D. 1998. *Palm Pilot: The Ultimate Guide*. O'Reilly & Associates, Sebastopol, Calif.

Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G. 1981. LOCUS: A network transparent, high reliability distributed system. In *Proceedings of the Eighth ACM Symposium on Operating System Principles* (Pacific Grove, Calif., Dec.).

Popek, G. J. and Walker, B. J. 1985. *The LOCUS Distributed System Architecture*. MIT Press, Cambridge, Mass.

Rajlich, V. T. and Bennett, K. H. 2000. A staged model for the software life cycle. *IEEE Computer 33*, 7 (July).

Rashid, R., Baron, R., Forin, A., Golub, D., Jones, M., Julin, D., Orr, D., and Sanzi, R. 1989. Mach: A foundation for open systems. In *Proceedings of the Second Workshop on Workstation Operating Systems* (Pacific Grove, Calif., Sept.).

Rivest, R. 1992. The MD5 message-digest algorithm, Internet RFC 1321. Available at http://theory.lcs.mit.edu/~rivest/publications.html.

Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. 1985. Design and implementation of the Sun Network Filesystem. In *Summer Usenix Conference Proceedings* (Portland, Ore.).

Satyanarayanan, M. 1990. Scalable, secure, and highly available distributed file access. *IEEE Computer 23*, 5 (May).

Satyanarayanan, M. 2000. Caching trust rather than content. *Oper. Syst. Rev. 34*, 4 (Oct.).

Satyanarayanan, M. 2001. Pervasive computing: Vision and challenges. *IEEE Pers. Commun. 8*, 4 (August).

Satyanarayanan, M. and Siegel, E. H. 1990. Parallel communication in a large distributed environment. *IEEE Trans. Comput. 39*, 3 (March).

Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C. 1990. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput. 39*, 4 (April).

Satyanarayanan, M., Kistler, J. J., Mummert, L. B., Ebling, M. R., Kumar, P., and Lu, Q. 1993a. Experience with disconnected operation in a mobile computing environment. In *Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing* (Cambridge, Mass., August).

Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., and Kistler, J. J. 1993b. Lightweight recoverable virtual memory. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, N.C., Dec.).

Satyanarayanan, M., Kistler, J. J., and Siegel, E. H. 1987. Coda: A resilient distributed file system. In *Proceedings of the IEEE Workshop on Workstation Operating Systems* (Cambridge, Mass. Nov.).

Schneier, B. 1996. *Applied Cryptography*. Wiley, New York.

Schroeder, M. D. and Burrows, M. 1990. Performance of Firefly RPC. *ACM Trans. Comput. Syst. 8*, 1 (Feb.).

Schulman, A. 1995. *Unauthorized Windows 95*. IDG Books.

SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. 1996. Dealing with disaster: surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation* (Seattle, Wash., Oct.).

SHORT, R. 1997. What a tangled mess! Untangling user-visible complexity in windows systems. In *The USENIX Windows NT Workshop Proceedings* (Seattle, Wash., August). Invited Talk Slides available at //www.usenix.org/publications/library/proceedings/usenix-nt97/presentations/index.html.

SOMERVILLE, I. 1989. *Software Engineering, Third ed.* Addison Wesley, Reading, Mass.

STEERE, D. C., KISTLER, J. J., AND SATYANARAYANAN, M. 1990. Efficient user-level cache file management on the Sun Vnode interface. In *Summer Usenix Conference Proceedings* (Anaheim, Calif., June).

TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (Copper Mountain, Co., Dec.).

THEKKATH, C. A., MANN, T., AND LEE, E. K. 1997. Frangipani: A scalable distributed file system. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles* (Saint-Malo, France, Oct.).

VAN RENESSE, R., VAN STAVEREN, H., AND TANENBAUM, A. S. 1988. Performance of the world's fastest distributed operating system. *Oper. Syst. Rev. 22*, 4 (Oct.).

WALKER, B., POPEK, G., ENGLISH, R., KLINE, C., AND THIEL, G. 1983. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating System Principles* (Bretton Woods, N.H., Oct.).

WEBSTER'S NINTH NEW COLLEGIATE DICTIONARY. 1988. Merriam-Webster, Springfield, Mass.

Received October 2000; revised October 2001; accepted December 2001