Disconnected Operation in the Coda File System

James J. Kistler and M. Satyanarayanan

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Disconnected operation is a mode of operation that enables a client to continue accessing critical data during temporary failures of a shared data repository. An important, though not exclusive, application of disconnected operation is in supporting portable computers. In this paper, we show that disconnected operation is feasible, efficient and usable by describing its design and implementation in the Coda File System. The central idea behind our work is that *caching of data*, now widely used for performance, can also be exploited to improve *availability*.

1. Introduction

Every serious user of a distributed system has faced situations where critical work has been impeded by a remote failure. His frustration is particularly acute when his workstation is powerful enough to be used standalone, but has been configured to be dependent on remote resources. An important instance of such dependence is the use of data from a distributed file system.

Placing data in a distributed file system simplifies collaboration between users, and allows them to delegate the administration of that data. The growing popularity of distributed file systems such as NFS [15] and AFS [18] attests to the compelling nature of these considerations. Unfortunately, the users of these systems have to accept the fact that a remote failure at a critical juncture may seriously inconvenience them.

This work was supported by the Defense Advanced Research Projects Agency (Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio, 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597), National Science Foundation (PYI Award and Grant No. ECD 8907068), IBM Corporation (Faculty Development Award, Graduate Fellowship, and Research Initiation Grant), Digital Equipment Corporation (External Research Project Grant), and Bellcore (Information Networking Research Grant).

How can we improve this state of affairs? Ideally, we would like to enjoy the benefits of a shared data repository, but be able to continue critical work when that repository is inaccessible. We call the latter mode of operation *disconnected operation*, because it represents a temporary deviation from normal operation as a client of a shared repository.

In this paper we show that disconnected operation in a file system is indeed feasible, efficient and usable. The central idea behind our work is that *caching of data*, now widely used to improve performance, can also be exploited to enhance *availability*. We have implemented disconnected operation in the Coda File System at Carnegie Mellon University.

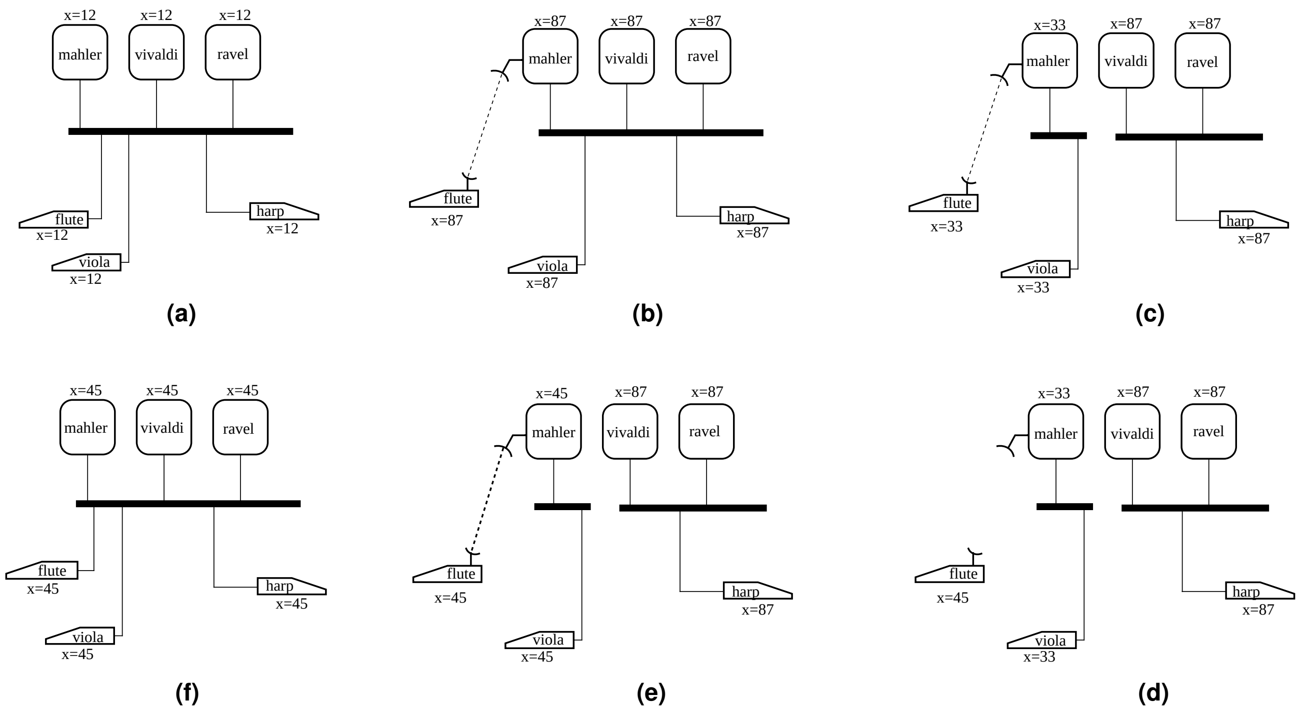
Our initial experience with Coda confirms the viability of disconnected operation. We have successfully operated disconnected for periods lasting four to five hours. For a disconnection of this duration, the process of reconnecting and propagating changes typically takes about a minute. A local disk of 100MB has been adequate for us during these periods of disconnection. Trace-driven simulations indicate that a disk of about half that size should be adequate for disconnections lasting a typical workday.

2. Design Overview

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

Each Coda client has a local disk and can communicate with the servers over a high bandwidth network. At certain times, a client may be temporarily unable to communicate with some or all of the servers. This may be due to a server or network failure, or due to the detachment of a *portable client* from the network.

[1]Unix is a trademark of AT&T.



Three servers (*mahler, vivaldi,* and *ravel*) have replicas of the volume containing file *x.* This file is potentially of interest to users at three clients (*flute*, *viola*, and *harp*). *Flute* is capable of wireless communication (indicated by a dotted line) as well as regular network communication. Proceeding clockwise, the steps above show the value of *x* seen by each node as the connectivity of the system changes. Note that in step (d), *flute* is operating disconnected.

Figure 1: How Disconnected Operation Relates to Server Replication

Clients view Coda as a single, location-transparent shared Unix file system. The Coda namespace is mapped to individual file servers at the granularity of subtrees called *volumes.* At each client, a *cache manager (Venus)* dynamically obtains and caches volume mappings.

Coda uses two distinct, but complementary, mechanisms to achieve high availability. The first mechanism, *server replication,* allows volumes to have read-write replicas at more than one server. The set of replication sites for a volume is its *volume storage group (VSG).* The subset of a VSG that is currently accessible is a client's *accessible VSG (AVSG).* The performance cost of server replication is kept low by caching on disks at clients and through the use of parallel access protocols. Venus uses a cache coherence protocol based on *callbacks* [9] to guarantee that an open of a file yields its latest copy in the AVSG. This guarantee is provided by servers notifying clients when their cached copies are no longer valid, each notification being referred to as a "callback break." Modifications in Coda are propagated in parallel to all AVSG sites, and eventually to missing VSG sites.

Disconnected operation, the second high availability mechanism used by Coda, takes effect when the AVSG becomes empty. 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. When disconnection ends, Venus propagates modifications and reverts to server replication. Figure 1 depicts a typical scenario involving transitions between server replication and disconnected operation.

Earlier Coda papers [17, 18] have described server replication in depth. In contrast, this paper restricts its attention to disconnected operation. We discuss server replication only in those areas where its presence has significantly influenced our design for disconnected operation.

3. Design Rationale

At a high level, two factors influenced our strategy for high availability. First, we wanted to use conventional, off-the-shelf hardware throughout our system. Second, we wished to preserve *transparency* by seamlessly integrating the high availability mechanisms of Coda into a normal Unix environment.

At a more detailed level, other considerations influenced our design. These include the need to *scale* gracefully, the advent of *portable workstations,* the very different *resource, integrity,* and *security* assumptions made about clients and servers, and the need to strike a balance between *availability* and *consistency*. We examine each of these issues in the following sections.

3.1. Scalability

Successful distributed systems tend to grow in size. Our experience with Coda's ancestor, AFS, had impressed upon us the need to prepare for growth *a priori*, rather than treating it as an afterthought [16]. We brought this experience to bear upon Coda in two ways. First, we adopted certain mechanisms that enhance scalability. Second, we drew upon a set of general principles to guide our design choices.

An example of a mechanism we adopted for scalability is callback-based cache coherence. Another such mechanism, *whole-file caching*, offers the added advantage of a much simpler failure model: a cache miss can only occur on an `open`, never on a `read`, `write`, `seek`, or `close`. This, in turn, substantially simplifies the implementation of disconnected operation. A partial-file caching scheme such as that of AFS-4 [21], Echo [8] or MFS [1] would have complicated our implementation and made disconnected operation less transparent.

A scalability principle that has had considerable influence on our design is the *placing of functionality on clients* rather than servers. Only if integrity or security would have been compromised have we violated this principle. Another scalability principle we have adopted is the *avoidance of system-wide rapid change*. Consequently, we have rejected strategies that require election or agreement by large numbers of nodes. For example, we have avoided algorithms such as that used in Locus [22] that depend on nodes achieving consensus on the current partition state of the network.

3.2. Portable Workstations

Powerful, lightweight and compact laptop computers are commonplace today. It is instructive to observe how a person with data in a shared file system uses such a machine. Typically, he identifies files of interest and downloads them from the shared file system into the local name space for use while isolated. When he returns, he copies modified files back into the shared file system. Such a user is effectively performing manual caching, with write-back upon reconnection!

Early in the design of Coda we realized that 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. Thus portable machines are a champion application for disconnected operation.

The use of portable machines also gave us another insight. The fact that people are able to operate for extended periods in isolation indicates that they are quite good 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.

Functionally, *involuntary* disconnections caused by failures are no different from *voluntary* disconnections caused by unplugging portable computers. Hence Coda provides a single mechanism to cope with all disconnections. Of course, there may be qualitative differences: user expectations as well as the extent of user cooperation are likely to be different in the two cases.

3.3. First vs Second Class Replication

If disconnected operation is feasible, why is server replication needed at all? The answer to this question depends critically on the very different assumptions made about clients and servers in Coda.

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 like public utilities: they have much greater disk capacity, they are physically secure, and they are carefully 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. 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. 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 duration of disconnection, the greater the potential for degradation. Whereas server replication preserves the quality of data in the face of failures, disconnected operation forsakes quality for availability. Hence server replication is important because it reduces the frequency and duration of disconnected operation, which is properly viewed as a measure of last resort.

Server replication is expensive because it requires additional hardware. Disconnected operation, in contrast, costs little. Whether to use server replication or not is thus a tradeoff between quality and cost. Coda does permit a volume to have a sole server replica. Therefore, an installation can rely exclusively on disconnected operation if it so chooses.

3.4. Optimistic vs Pessimistic Replica Control

By definition, a network partition exists between a disconnected second class replica and all its first class associates. The choice between two families of replica control strategies, *pessimistic* and *optimistic* [5], is therefore central to the design of disconnected operation. A pessimistic strategy avoids conflicting operations by disallowing all partitioned writes or by restricting reads and writes to a single partition. An optimistic strategy provides much higher availability by permitting reads and writes everywhere, and deals with the attendant danger of conflicts by detecting and resolving them after their occurence.

A pessimistic approach towards disconnected operation would require a client to acquire shared or exclusive control of a cached object prior to disconnection, and to retain such control until reconnection. Possession of exclusive control by a disconnected client would preclude reading or writing at all other replicas. Possession of shared control would allow reading at other replicas, but writes would still be forbidden everywhere.

Acquiring control prior to voluntary disconnection is relatively simple. It is more difficult when disconnection is involuntary, because the system may have to arbitrate among multiple requestors. Unfortunately, the information needed to make a wise decision is not readily available. For example, the system cannot predict which requestors will actually use the object, when they will release control, or what the relative costs of denying them access would be.

Retaining control until reconnection is acceptable in the case of brief disconnections. But it is unacceptable in the case of extended disconnections. A disconnected client with shared control of an object would force the rest of the system to defer all updates until it reconnected. With exclusive control, it would even prevent other users from making a copy of the object. Coercing the client to reconnect may not be feasible, since its whereabouts may not be known. Thus, an entire user community could be at the mercy of a single errant client for an unbounded amount of time.

Placing a time bound on exclusive or shared control, as done in the case of *leases* [7], avoids this problem but introduces others. Once a lease expires, a disconnected client loses the ability to access a cached object, even if no else in the system is interested in it. This, in turn, defeats the purpose of disconnected operation which is to provide high availability. Worse, updates already made while disconnected have to be discarded.

An optimistic approach has its own disadvantages. An update made at one disconnected client may conflict with an update at another disconnected or connected client. For optimistic replication to be viable, the system has to be more sophisticated. There needs to be machinery in the system for detecting conflicts, for automating resolution when possible, and for confining damage and preserving evidence for manual repair. Having to repair conflicts manually violates transparency, is an annoyance to users, and reduces the usability of the system.

We chose optimistic replication because we felt that its strengths and weaknesses better matched our design goals. The dominant influence on our choice was the low degree of write-sharing typical of Unix. This implied that an optimistic strategy was likely to lead to relatively few conflicts. An optimistic strategy was also consistent with our overall goal of providing the highest possible availability of data.

In principle, we could have chosen a pessimistic strategy for server replication even after choosing an optimistic strategy for disconnected operation. But that would have reduced transparency, because a user would have faced the anomaly of being able to update data when disconnected, but being unable to do so when connected to a subset of the servers. Further, many of the previous arguments in favor of an optimistic strategy also apply to server replication.

Using an optimistic strategy throughout presents a uniform model of the system from the user's perspective. At any time, he is able to read the latest data in his *accessible universe* and his updates are immediately visible to everyone else in that universe. His accessible universe is usually the entire set of servers and clients. When failures occur, his accessible universe shrinks to the set of servers he can contact, and the set of clients that they, in turn, can contact. In the limit, when he is operating disconnected, his accessible universe consists of just his machine. Upon reconnection, his updates become visible throughout his now-enlarged accessible universe.

4. Detailed Design and Implementation

In describing our implementation of disconnected operation, we focus on the client since this is where much of the complexity lies. Section 4.1 describes the physical structure of a client, Section 4.2 introduces the major states of Venus, and Sections 4.3 to 4.5 discuss these states in detail. A description of the server support needed for disconnected operation is contained in Section 4.5.

4.1. Client Structure

Because of the complexity of Venus, we made it a user level process rather than part of the kernel. The latter approach may have yielded better performance, but would have been less portable and considerably more difficult to debug. Figure 2 illustrates the high-level structure of a Coda client.

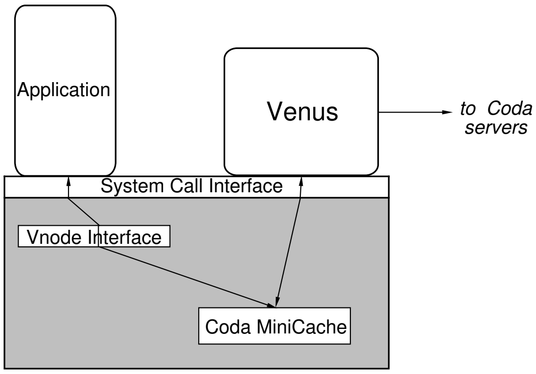


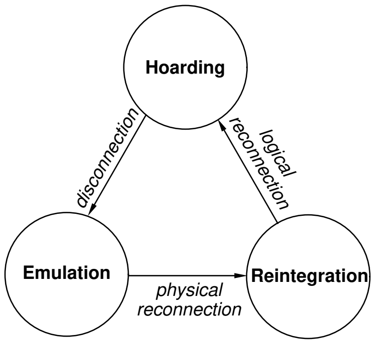
Figure 2: Structure of a Coda Client

Venus intercepts Unix file system calls via the widely-used Sun Vnode interface [10]. Since this interface imposes a heavy performance overhead on user-level cache managers, we use a tiny in-kernel *MiniCache* to filter out many kernel-Venus interactions. The MiniCache contains no support for remote access, disconnected operation or server replication; these functions are handled entirely by Venus.

A system call on a Coda object is forwarded by the Vnode interface to the MiniCache. If possible, the call is serviced by the MiniCache and control is returned to the application. Otherwise, the MiniCache contacts Venus to service the call. This, in turn, may involve contacting Coda servers. Control returns from Venus via the MiniCache to the application program, updating MiniCache state as a side effect. MiniCache state changes may also be initiated by Venus on events such as callback breaks from Coda servers. Measurements from our implementation confirm that the MiniCache is critical for good performance [20].

4.2. Venus States

Logically, Venus operates in one of three states: *hoarding*, *emulation*, and *reintegration*. Figure 3 depicts these states and the transitions between them. Venus is normally in the hoarding state, relying on server replication but always on the alert for possible disconnection. Upon disconnection, it enters the emulation state and remains there for the duration of disconnection. Upon reconnection, Venus enters the reintegration state, resynchronizes its cache with its AVSG, and then reverts to the hoarding state. Since all volumes may not be replicated across the same set of servers, Venus can be in different states with respect to different volumes, depending on failure conditions in the system.



When disconnected, Venus is in the emulation state. It transits to reintegration upon successful reconnection to an AVSG member, and thence to hoarding, where it resumes connected operation.

Figure 3: Venus States and Transitions

4.3. Hoarding

The hoarding state is so named because a key responsibility of Venus in this state is to hoard useful data in anticipation of disconnection. However, this is not its only responsibility. Rather, Venus must manage its cache in a manner that balances the needs of connected and disconnected operation. For instance, a user may have indicated that a certain set of files is critical but may currently be using other files. To provide good performance, Venus must cache the latter files. But to be prepared for disconnection, it must also cache the former set of files.

Many factors complicate the implementation of hoarding:

- File reference behavior, especially in the distant future, cannot be predicted with certainty.
- Disconnections and reconnections are often unpredictable.
- The true cost of a cache miss while disconnected is highly variable and hard to quantify.
- Activity at other clients must be accounted for, so that the latest version of an object is in the cache at disconnection.
- Since cache space is finite, the availability of less critical objects may have to be sacrificed in favor of more critical objects.

To address these concerns, we manage the cache using a *prioritized algorithm*, and periodically reevaluate which objects merit retention in the cache via a process known as *hoard walking*.

```
# Personal files
a /coda/usr/jjk d+
a /coda/usr/jjk/papers 100:d+
a /coda/usr/jjk/papers/sosp 1000:d+

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

(a)

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

(b)

```
# Venus source files
# (shared among Coda developers)
a /coda/project/coda/src/venus 100:c+
a /coda/project/coda/include 100:c+
a /coda/project/coda/lib c+
```

(c)

These are typical hoard profiles provided by a Coda user, an application maintainer, and a group of project developers. Each profile is interpreted separately by the HDB front-end program. The 'a' at the beginning of a line indicates an add-entry command. Other commands are delete an entry, clear all entries, and list entries. The modifiers following some pathnames specify non-default priorities (the default is 10) and/or meta-expansion for the entry. Note that the pathnames beginning with '/usr' are actually symbolic links into '/coda'.

Figure 4: Sample Hoard Profiles

4.3.1. Prioritized Cache Management

Venus combines implicit and explicit sources of information in its priority-based cache management algorithm. The implicit information consists of recent reference history, as in traditional caching algorithms. Explicit information takes the form of a per-workstation *hoard database (HDB),* whose entries are pathnames identifying objects of interest to the user at that workstation.

A simple front-end program allows a user to update the HDB using command scripts called *hoard profiles,* such as those shown in Figure 4. Since hoard profiles are just files, it is simple for an application maintainer to provide a common profile for his users, or for users collaborating on a project to maintain a common profile. A user can customize his HDB by specifying different combinations of profiles or by executing front-end commands interactively. To facilitate construction of hoard profiles, Venus can record all file references observed between a pair of start and stop events indicated by a user.

To reduce the verbosity of hoard profiles and the effort needed to maintain them, Venus supports *meta-expansion* of HDB entries. As shown in Figure 4, if the letter 'c' (or 'd') follows a pathname, the command also applies to immediate children (or all descendants). A '+' following the 'c' or 'd' indicates that the command applies to all future as well as present children or descendents. A hoard entry may optionally indicate a *hoard priority,* with higher priorities indicating more critical objects.

The current priority of a cached object is a function of its hoard priority as well as a metric representing recent usage. The latter is updated continuously in response to new references, and serves to age the priority of objects no longer in the working set. Objects of the lowest priority are chosen as victims when cache space has to be reclaimed.

To resolve the pathname of a cached object while disconnected, it is imperative that all the ancestors of the object also be cached. Venus must therefore ensure that a cached directory is not purged before any of its descendants. This *hierarchical cache management* is not needed in traditional file caching schemes because cache misses during name translation can be serviced, albeit at a performance cost. Venus performs hierarchical cache management by assigning infinite priority to directories with cached children. This automatically forces replacement to occur bottom-up.

4.3.2. Hoard Walking

We say that a cache is in *equilibrium,* signifying that it meets user expectations about availability, when no uncached object has a higher priority than a cached object. Equilibrium may be disturbed as a result of normal activity. For example, suppose an object, *A,* is brought into the cache on demand, replacing an object, *B.* Further suppose that *B* is mentioned in the HDB, but *A* is not. Some time after activity on *A* ceases, its priority will decay below the hoard priority of *B.* The cache is no longer in equilibrium, since the cached object *A* has lower priority than the uncached object *B.*

Venus periodically restores equilibrium by performing an operation known as a *hoard walk.* A hoard walk occurs every 10 minutes in our current implementation, but one may be explicitly requested by a user prior to voluntary disconnection. The walk occurs in two phases. First, the *name bindings* of HDB entries are reevaluated to reflect update activity by other Coda clients. For example, new children may have been created in a directory whose pathname is specified with the '+' option in the HDB. Second, the priorities of all entries in the cache and HDB are reevaluated, and objects fetched or evicted as needed to restore equilibrium.

Hoard walks also address a problem arising from callback breaks. In traditional callback-based caching, data is refetched only on demand after a callback break. But in Coda, such a strategy may result in a critical object being unavailable should a disconnection occur before the next reference to it. Refetching immediately upon callback break avoids this problem, but ignores a key characteristic of Unix environments: once an object is modified, it is likely to be modified many more times by the same user within a short interval [14, 6]. An immediate refetch policy would increase client-server traffic considerably, thereby reducing scalability.

Our strategy is a compromise that balances availability, consistency, and scalability. For files and symbolic links, Venus purges the object on callback break, and refetches it on demand or during the next hoard walk, whichever occurs earlier. If a disconnection were to occur before refetching, the object would be unavailable. For directories, Venus does not purge on callback break, but marks the cache entry suspicious. A stale cache entry is thus available should a disconnection occur before the next hoard walk or reference. The acceptability of stale directory data follows from its particular callback semantics. A callback break on a directory typically means that an entry has been added to or deleted from the directory. It is often the case that other directory entries and the objects they name are unchanged. Therefore, saving the stale copy and using it in the event of untimely disconnection causes consistency to suffer only a little, but increases availability considerably.

4.4. Emulation

In the emulation state, Venus performs many actions normally handled by servers. For example, Venus now assumes full responsibility for access and semantic checks. It is also responsible for generating temporary *file identifiers* (fids) for new objects, pending the assignment of permanent fids at reintegration. But although Venus is functioning as a *pseudo-server,* updates accepted by it have to be revalidated with respect to integrity and protection by real servers. This follows from the Coda policy of trusting only servers, not clients. To minimize unpleasant delayed surprises for a disconnected user, it behooves Venus to be as faithful as possible in its emulation.

Cache management during emulation is done with the same priority algorithm used during hoarding. Mutating operations directly update the cache entries of the objects involved. Cache entries of deleted objects are freed immediately, but those of other modified objects assume infinite priority so that they are not purged before reintegration. On a cache miss, the default behavior of Venus is to return an error code. A user may optionally request Venus to block his processes until cache misses can be serviced.

4.4.1. Logging

During emulation, Venus records sufficient information to replay update activity when it reintegrates. It maintains this information in a per-volume log of mutating operations called a *replay log.* Each log entry contains a copy of the corresponding system call arguments as well as the version state of all objects referenced by the call.

Venus uses a number of optimizations to reduce the length of the replay log, resulting in a log size that is typically a few percent of cache size. A small log conserves disk space, a critical resource during periods of disconnection. It also improves reintegration performance by reducing latency and server load.

One important optimization to reduce log length pertains to `write` operations on files. Since Coda uses whole-file caching, the `close` after an `open` of a file for modification installs a completely new copy of the file. Rather than logging the `open`, `close`, and intervening `write` operations individually, Venus logs a single `store` record during the handling of a `close`.

Another optimization consists of Venus discarding a previous `store` record for a file when a new one is appended to the log. This follows from the fact that a `store` renders all previous versions of a file superfluous. The `store` record does not contain a copy of the file's contents, but merely points to the copy in the cache.

We are currently implementing two further optimizations to reduce the length of the replay log. The first generalizes the optimization described in the previous paragraph such that any operation which *overwrites* the effect of earlier operations may cancel the corresponding log records. An example would be the cancelling of a `store` by a subsequent `unlink` or `truncate`. The second optimization exploits knowledge of *inverse* operations to cancel both the inverting and inverted log records. For example, a `rmdir` may cancel its own log record as well as that of the corresponding `mkdir`.

4.4.2. Persistence

A disconnected user must be able to restart his machine after a shutdown and continue where he left off. In case of a crash, the amount of data lost should be no greater than if the same failure occurred during connected operation. To provide these guarantees, Venus must keep its cache and related data structures in non-volatile storage.

Meta-data, consisting of cached directory and symbolic link contents, status blocks for cached objects of all types, replay logs, and the HDB, is mapped into Venus' address space as *recoverable virtual memory* (RVM). Transactional access to this memory is supported by the RVM library [12] linked into Venus. The actual contents of cached files are not in RVM, but are stored as local Unix files.

The use of transactions to manipulate meta-data simplifies Venus' job enormously. To maintain its invariants Venus need only ensure that each transaction takes meta-data from one consistent state to another. It need not be concerned with crash recovery, since RVM handles this transparently. If we had chosen the obvious alternative of placing meta-data in local Unix files, we would have had to follow a strict discipline of carefully timed synchronous writes and an ad-hoc recovery algorithm.

RVM supports local, non-nested transactions and allows independent control over the basic transactional properties of atomicity, permanence, and serializability. An application can reduce commit latency by labelling the commit as *no-flush,* thereby avoiding a synchronous write to disk. To ensure persistence of no-flush transactions, the application must explicitly flush RVM's write-ahead log from time to time. When used in this manner, RVM provides *bounded persistence*, where the bound is the period between log flushes.

Venus exploits the capabilities of RVM to provide good performance at a constant level of persistence. When hoarding, Venus initiates log flushes infrequently, since a copy of the data is available on servers. Since servers are not accessible when emulating, Venus is more conservative and flushes the log more frequently. This lowers performance, but keeps the amount of data lost by a client crash within acceptable limits.

4.4.3. Resource Exhaustion

It is possible for Venus to exhaust its non-volatile storage during emulation. The two significant instances of this are the file cache becoming filled with modified files, and the RVM space allocated to replay logs becoming full.

Our current implementation is not very graceful in handling these situations. When the file cache is full, space can be freed by truncating or deleting modified files. When log space is full, no further mutations are allowed until reintegration has been performed. Of course, non-mutating operations are always allowed.

We plan to explore at least three alternatives to free up disk space while emulating. One possibility is to compress file cache and RVM contents. Compression trades off computation time for space, and recent work [2] has shown it to be a promising tool for cache management. A second possibility is to allow users to selectively back out updates made while disconnected. A third approach is to allow portions of the file cache and RVM to be written out to removable media such as floppy disks.

4.5. Reintegration

Reintegration is a transitory state through which Venus passes in changing roles from pseudo-server to cache manager. In this state, Venus propagates changes made during emulation, and updates its cache to reflect current server state. Reintegration is performed a volume at a time, with all update activity in the volume suspended until completion.

4.5.1. Replay Algorithm

The propagation of changes from client to AVSG is accomplished in two steps. In the first step, Venus obtains permanent fids for new objects and uses them to replace temporary fids in the replay log. This step is avoided in many cases, since Venus obtains a small supply of permanent fids in advance of need, while in the hoarding state. In the second step, the replay log is shipped in parallel to the AVSG, and executed independently at each member. Each server performs the replay within a single transaction, which is aborted if any error is detected.

The replay algorithm consists of four phases. In phase one the log is parsed, a transaction is begun, and all objects referenced in the log are locked. In phase two, each operation in the log is validated and then executed. The validation consists of conflict detection as well as integrity, protection, and disk space checks. Except in the case of `store` operations, execution during replay is identical to execution in connected mode. For a `store`, an empty *shadow file* is created and meta-data is updated to reference it, but the data transfer is deferred. Phase three consists exclusively of performing these data transfers, a process known as *back-fetching*. The final phase commits the transaction and releases all locks.

If reintegrations succeeds, Venus frees the replay log and resets the priority of cached objects referenced by the log. If reintegration fails, Venus writes out the replay log to a local *replay file* in a superset of the Unix `tar` format. The log and all corresponding cache entries are then purged, so that subsequent references will cause refetch of the current contents at the AVSG. A tool is provided which allows the user to inspect the contents of a replay file, compare it to the state at the AVSG, and replay it selectively or in its entirety.

Reintegration at finer granularity than a volume would reduce the latency perceived by clients, improve concurrency and load balancing at servers, and reduce user effort during manual replay. To this end, we are revising our implementation to reintegrate at the granularity of subsequences of dependent operations within a volume. Dependent subsequences can be identified using the *precedence graph* approach of Davidson [4]. In the revised implementation Venus will maintain precedence graphs during emulation, and pass them to servers along with the replay log.

4.5.2. Conflict Handling

Our use of optimistic replica control means that the disconnected operations of one client may conflict with activity at servers or other disconnected clients. The only class of conflicts we are concerned with are *write/write* conflicts. *Read/write* conflicts are not relevant to the Unix file system model, since it has no notion of atomicity beyond the boundary of a single system call.

The check for conflicts relies on the fact that each replica of an object is tagged with a *storeid* that uniquely identifies the last update to it. During phase two of replay, a server compares the storeid of every object mentioned in a log entry with the storeid of its own replica of the object. If the comparison indicates equality for all objects, the operation is performed and the mutated objects are tagged with a new storeid specified in the log entry.

If a storeid comparison fails, the action taken depends on the operation being validated. In the case of a `store` of a file, the entire reintegration is aborted. But for directories, a conflict is declared only if a newly created name collides with an existing name, if an object updated at the client or the server has been deleted by the other, or if directory attributes have been modified at the server and the client. This strategy of *resolving* partitioned directory updates is consistent with our strategy in server replication [11], and was originally suggested by Locus [22].

Our original design for disconnected operation called for preservation of replay files at servers rather than clients. This approach would also allow damage to be confined by marking conflicting replicas inconsistent and forcing manual repair, as is currently done in the case of server replication. We are awaiting more usage experience to determine whether this is indeed the correct approach for disconnected operation.

5. Status and Evaluation

Today, Coda runs on IBM RTs, Decstation 3100s and 5000s, and 386-based laptops such as the Toshiba 5200. A small user community has been using Coda on a daily basis as its primary data repository since April 1990. All development work on Coda is done in Coda itself. As of July 1991 there were nearly 350MB of triply-replicated data in Coda, with plans to expand to 2GB in the next few months.

A version of disconnected operation with minimal functionality was demonstrated in October 1990. A more complete version was functional in January 1991, and is now in regular use. We have successfully operated disconnected for periods lasting four to five hours. Our experience with the system has been quite positive, and we are confident that the refinements under development will result in an even more usable system.

In the following sections we provide qualitative and quantitative answers to three important questions pertaining to disconnected operation. These are:

1. How long does reintegration take?
2. How large a local disk does one need?
3. How likely are conflicts?

5.1. Duration of Reintegration

In our experience, typical disconnected sessions of editing and program development lasting a few hours require about a minute for reintegration. To characterize reintegration speed more precisely, we measured the reintegration times after disconnected execution of two well-defined tasks. The first task is the Andrew benchmark [9], now widely used as a basis for comparing file system performance. The second task is the compiling and linking of the current version of Venus. Table 1 presents the reintegration times for these tasks.

The time for reintegration consists of three components: the time to allocate permanent fids, the time for the replay at the servers, and the time for the second phase of the update protocol used for server replication. The first component will be zero for many disconnections, due to the preallocation of fids during hoarding. We expect the time for the second component to fall, considerably in many cases, as we incorporate the last of the replay log optimizations described in Section 4.4.1. The third component can be avoided only if server replication is not used.

One can make some interesting secondary observations from Table 1. First, the total time for reintegration is roughly the same for the two tasks even though the Andrew benchmark has a much smaller elapsed time. This is because the Andrew benchmark uses the file system more intensively. Second, reintegration for the Venus make takes longer, even though the number of entries in the replay log is smaller. This is because much more file data is back-fetched in the third phase of the replay. Finally, neither task involves any think time. As a result, their reintegration times are comparable to that after a much longer, but more typical, disconnected session in our environment.

5.2. Cache Size

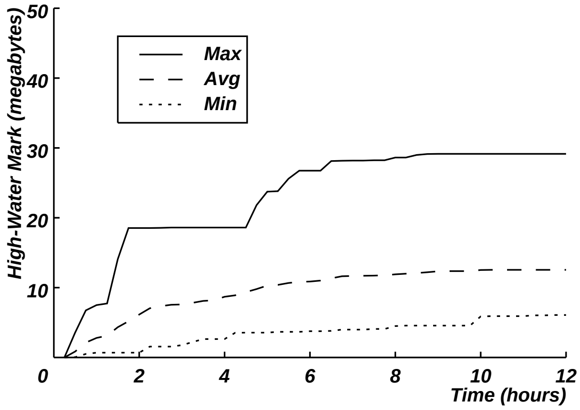
A local disk capacity of 100MB on our clients has proved adequate for our initial sessions of disconnected operation. To obtain a better understanding of the cache size requirements for disconnected operation, we analyzed file reference traces from our environment. The traces were obtained by instrumenting workstations to record information on every file system operation, regardless of whether the file was in Coda, AFS, or the local file system.

	Elapsed Time (seconds)	Reintegration Time (seconds)				Size of Replay Log		Data Back-Fetched (Bytes)
		Total	AllocFid	Replay	COP2	Records	Bytes	
Andrew Benchmark	288 (3)	43 (2)	4 (2)	29 (1)	10 (1)	223	65,010	1,141,315
Venus Make	3,271 (28)	52 (4)	1 (0)	40 (1)	10 (3)	193	65,919	2,990,120

This data was obtained with a Toshiba T5200/100 client (12MB memory, 100MB disk) reintegrating over an Ethernet with an IBM RT-APC server (12MB memory, 400MB disk). The values shown above are the means of three trials. Figures in parentheses are standard deviations.

Table 1: Time for Reintegration

Our analysis is based on simulations driven by these traces. Writing and validating a simulator that precisely models the complex caching behavior of Venus would be quite difficult. To avoid this difficulty, we have modified Venus to act as its own simulator. When running as a simulator, Venus is driven by traces rather than requests from the kernel. Code to communicate with the servers, as well code to perform physical I/O on the local file system are stubbed out during simulation.



This graph is based on a total of 10 traces from 5 active Coda workstations. The curve labelled "Avg" corresponds to the values obtained by averaging the high-water marks of all workstations. The curves labelled "Max" and "Min" plot the highest and lowest values of the high-water marks across all workstations. Note that the high-water mark does not include space needed for paging, the HDB or replay logs.

Figure 5: High-Water Mark of Cache Usage

Figure 5 shows the *high-water mark* of cache usage as a function of time. The actual disk size needed for disconnected operation has to be larger, since both the explicit and implicit sources of hoarding information are imperfect. From our data it appears that a disk of 50-60MB should be adequate for operating disconnected for a typical workday. Of course, user activity that is drastically different from what was recorded in our traces could produce significantly different results.

We plan to extend our work on trace-driven simulations in three ways. First, we will investigate cache size requirements for much longer periods of disconnection. Second, we will be sampling a broader range of user activity by obtaining traces from many more machines in our environment. Third, we will evaluate the effect of hoarding by simulating traces together with hoard profiles that have been specified *ex ante* by users.

5.3. Likelihood of Conflicts

In our use of optimistic server replication in Coda for nearly a year, we have seen virtually no conflicts due to multiple users updating an object in different network partitions. While gratifying to us, this observation is subject to at least three criticisms. First, it is possible that our users are being cautious, knowing that they are dealing with an experimental system. Second, perhaps conflicts will become a problem only as the Coda user community grows larger. Third, perhaps extended voluntary disconnections will lead to many more conflicts.

To obtain data on the likelihood of conflicts at larger scale, we instrumented the AFS servers in our environment. These servers are used by over 400 computer science faculty, staff and graduate students for research, program development, and education. Their usage profile includes a significant amount of collaborative activity. Since Coda is descended from AFS and makes the same kind of usage assumptions, we can use this data to estimate how frequent conflicts would be if Coda were to replace AFS in our environment.

Every time a user modifies an AFS file or directory, we compare his identity with that of the user who made the previous mutation. We also note the time interval between mutations. For a file, only the `close` after an `open` for update is counted as a mutation; individual `write` operations are not counted. For directories, all operations that modify a directory are counted as mutations.

Type of Volume	Number of Volumes	Type of Object	Total Mutations	Same User	Different User					
					Total	< 1min	< 10 min	< 1hr	< 1 day	< 1 wk
User	529	Files	3,287,135	99.87 %	0.13 %	0.04 %	0.05 %	0.06 %	0.09 %	0.09 %
		Directories	4,132,066	99.80 %	0.20 %	0.04 %	0.07 %	0.10 %	0.15 %	0.16 %
Project	108	Files	4,437,311	99.66 %	0.34 %	0.17 %	0.25 %	0.26 %	0.28 %	0.30 %
		Directories	5,391,224	99.63 %	0.37 %	0.00 %	0.01 %	0.03 %	0.09 %	0.15 %
System	398	Files	5,526,700	99.17 %	0.83 %	0.06 %	0.18 %	0.42 %	0.72 %	0.78 %
		Directories	4,338,507	99.54 %	0.46 %	0.02 %	0.05 %	0.08 %	0.27 %	0.34 %

This data was obtained between June 1990 and May 1991 from the AFS servers in the `cs.cmu.edu` cell. The servers stored a total of about 12GB of data. The column entitled "Same User" gives the percentage of mutations in which the user performing the mutation was the same as the one performing the immediately preceding mutation on the same file or directory. The remaining mutations contribute to the column entitled "Different User".

Table 2: Sequential Write-Sharing in AFS

Table 2 presents our observations over a period of twelve months. The data is classified by volume type: *user* volumes containing private user data, *project* volumes used for collaborative work, and *system* volumes containing program binaries, libraries, header files and other similar data. On average, a project volume has about 2600 files and 280 directories, and a system volume has about 1600 files and 130 directories. User volumes tend to be smaller, averaging about 200 files and 18 directories, because users often place much of their data in their project volumes.

Table 2 shows that over 99% of all modifications were by the previous writer, and that the chances of two different users modifying the same object less than a day apart is at most 0.75%. We had expected to see the highest degree of write-sharing on project files or directories, and were surprised to see that it actually occurs on system files. We conjecture that a significant fraction of this sharing arises from modifications to system files by operators, who change shift periodically. If system files are excluded, the absence of write-sharing is even more striking: more than 99.5% of all mutations are by the previous writer, and the chances of two different users modifying the same object within a week are less than 0.4%! This data is highly encouraging from the point of view of optimistic replication. It suggests that conflicts would not be a serious problem if AFS were replaced by Coda in our environment.

6. Related Work

Coda is unique in that it exploits caching for both performance and high availability while preserving a high degree of transparency. We are aware of no other system, published or unpublished, that duplicates this key aspect of Coda.

By providing tools to link local and remote name spaces, the Cedar file system [19] provided rudimentary support for disconnected operation. But since this was not its primary goal, Cedar did not provide support for hoarding, transparent reintegration or conflict detection. Files were versioned and immutable, and a Cedar cache manager could substitute a cached version of a file on reference to an unqualified remote file whose server was inaccessible. However, the implementors of Cedar observe that this capability was not often exploited since remote files were normally referenced by specific version number.

Birrell and Schroeder pointed out the possibility of "stashing" data for availability in an early discussion of the Echo file system [13]. However, a more recent description of Echo [8] indicates that it uses stashing only for the highest levels of the naming hierarchy.

The FACE file system [3] uses stashing but does not integrate it with caching. The lack of integration has at least three negative consequences. First, it reduces transparency because users and applications deal with two different name spaces, with different consistency properties. Second, utilization of local disk space is likely to be much worse. Third, recent usage information from cache management is not available to manage the stash. The available literature on FACE does not report on how much the lack of integration detracted from the usability of the system.

An application-specific form of disconnected operation was implemented in the PCMAIL system at MIT [Lambert88]. PCMAIL allowed clients to disconnect, manipulate existing mail messages and generate new ones, and re-synchronize with a central repository at reconnection. Besides relying heavily on the semantics of mail, PCMAIL was less transparent than Coda since it required manual re-synchronization as well as pre-registration of clients with servers.

The use of optimistic replication in distributed file systems was pioneered by Locus [22]. Since Locus used a peer-to-peer model rather than a client-server model, availability was achieved solely through server replication. There was no notion of caching, and hence of disconnected operation.

Coda has benefited in a general sense from the large body of work on transparency and performance in distributed file systems. In particular, Coda owes much to AFS [18], from which it inherits its model of trust and integrity, as well as its mechanisms and design philosophy for scalability.

7. Future Work

Disconnected operation in Coda is a facility under active development. In earlier sections of this paper we described work in progress in the areas of log optimization, granularity of reintegration, and evaluation of hoarding. Much additional work is also being done at lower levels of the system. In this section we consider two ways in which the scope of our work may be broadened.

An excellent opportunity exists in Coda for adding *transactional support to Unix.* Explicit transactions become more desirable as systems scale to hundreds or thousands of nodes, and the informal concurrency control of Unix becomes less effective. Many of the mechanisms supporting disconnected operation, such as operation logging, precedence graph maintenance, and conflict checking would transfer directly to a transactional system using optimistic concurrency control. Although transactional file systems are not a new idea, no such system with the scalability, availability, and performance properties of Coda has been proposed or built.

A different opportunity exists in extending Coda to support *weakly-connected operation,* in environments where connectivity is intermittent or of low bandwidth. Such conditions are found in networks that rely on voice-grade lines, or that use wireless technologies such as packet radio. The ability to mask failures, as provided by disconnected operation, is of value even with weak connectivity. But techniques which exploit and adapt to the communication opportunities at hand are also needed. Such techniques may include more aggressive write-back policies, compressed network transmission, partial file transfer, and caching at intermediate levels.

8. Conclusion

Disconnected operation is a tantalizingly simple idea. All one has to do is to pre-load one's cache with critical data, continue normal operation until disconnection, log all changes made while disconnected, and replay them upon reconnection.

Implementing disconnected operation is not so simple. It involves major modifications and careful attention to detail in many aspects of cache management. While hoarding, a surprisingly large volume and variety of interrelated state has to be maintained. When emulating, the persistence and integrity of client data structures become critical. During reintegration, there are dynamic choices to be made about the granularity of reintegration.

Only in hindsight do we realize the extent to which implementations of traditional caching schemes have been simplified by the guaranteed presence of a lifeline to a first-class replica. Purging and refetching on demand, a strategy often used to handle pathological situations in those implementations, is not viable when supporting disconnected operation. However, the obstacles to realizing disconnected operation are not insurmountable. Rather, the central message of this paper is that disconnected operation is indeed feasible, efficient and usable.

One way to view our work is to regard it as an extension of the idea of *write-back caching.* Whereas write-back caching has hitherto been used for performance, we have shown that it can be extended to mask temporary failures too. A broader view is that disconnected operation allows graceful transitions between states of *autonomy* and *interdependence* in a distributed system. Under favorable conditions, our approach provides all the benefits of remote data access; under unfavorable conditions, it provides continued access to critical data. We are certain that disconnected operation will become increasingly important as distributed systems grow in scale, diversity and vulnerability.

Acknowledgments

We wish to thank Lily Mummert for her invaluable assistance in collecting and postprocessing the file reference traces used in Section 5.2, and Dimitris Varotsis, who helped instrument the AFS servers which yielded the measurements of Section 5.3. We also wish to express our appreciation to past and present contributors to the Coda project, especially Puneet Kumar, Hank Mashburn, Maria Okasaki, and David Steere.

References

[1] Burrows, M.
Efficient Data Sharing.
PhD thesis, University of Cambridge, Computer Laboratory, December, 1988.

[2] Cate, V., Gross, T.
Combining the Concepts of Compression and Caching for a Two-Level File System.
In *Proceedings of the 4th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*. April, 1991.

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

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

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

[6] Floyd, R.A.
Transparency in Distributed File Systems.
Technical Report TR 272, Department of Computer Science, University of Rochester, 1989.

[7] Gray, C.G., Cheriton, D.R.
Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency.
In *Proceedings of the 12th ACM Symposium on Operating System Principles*. December, 1989.

[8] Hisgen, A., Birrell, A., Mann, T., Schroeder, M., Swart, G.
Availability and Consistency Tradeoffs in the Echo Distributed File System.
In *Proceedings of the Second Workshop on Workstation Operating Systems*. September, 1989.

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

[10] Kleiman, S.R.
Vnodes: An Architecture for Multiple File System Types in Sun UNIX.
In *Summer Usenix Conference Proceedings*. 1986.

[11] Kumar, P., Satyanarayanan, M.
Log-Based Directory Resolution in the Coda File System.
Technical Report CMU-CS-91-164, School of Computer Science, Carnegie Mellon University, 1991.

[12] Mashburn, H., Satyanarayanan, M.
RVM: Recoverable Virtual Memory User Manual
School of Computer Science, Carnegie Mellon University, 1991.

[13] Needham, R.M., Herbert, A.J.
Report on the Third European SIGOPS Workshop: "Autonomy or Interdependence in Distributed Systems".
SIGOPS Review 23(2), April, 1989.

[14] Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M., Thompson, J.
A Trace-Driven Analysis of the 4.2BSD File System.
In *Proceedings of the 10th ACM Symposium on Operating System Principles*. December, 1985.

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

[16] Satyanarayanan, M.
On the Influence of Scale in a Distributed System.
In *Proceedings of the 10th International Conference on Software Engineering*. April, 1988.

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

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

[19] Schroeder, M.D., Gifford, D.K., Needham, R.M.
A Caching File System for a Programmer's Workstation.
In *Proceedings of the 10th ACM Symposium on Operating System Principles*. December, 1985.

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

[21] *Decorum File System*
Transarc Corporation, 1990.

[22] Walker, B., Popek, G., English, R., Kline, C., Thiel, G.
The LOCUS Distributed Operating System.
In *Proceedings of the 9th ACM Symposium on Operating System Principles*. October, 1983.