Exploiting Weak Connectivity for Mobile File Access

Lily B. Mummert, Maria R. Ebling, M. Satyanarayanan

August 1995
CMU-CS-95-185

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in
Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain Resort, CO, December 1995

Abstract

Weak connectivity, in the form of intermittent, low-bandwidth, or expensive networks is a fact of life in mobile computing. In this paper, we describe how the Coda File System has evolved to exploit such networks. The underlying theme of this evolution has been the systematic introduction of *adaptivity* to eliminate hidden assumptions about strong connectivity. Many aspects of the system, including communication, cache validation, update propagation and cache miss handling have been modified. As a result, Coda is able to provide good performance even when network bandwidth varies over four orders of magnitude — from modem speeds to LAN speeds.

This research was supported by the Air Force Materiel Command (AFMC) and the Advanced Research Projects Agency (ARPA) under contract number F196828-93-C-0193. Additional support was provided by the IBM Corp., Digital Equipment Corp., Intel Corp., Xerox Corp., and AT&T Corp. The U.S. government is authorized to reproduce and distribute reprints for government purposes, notwithstanding any copyright notation thereon.

Keywords: distributed file systems, mobile computing, weak connectivity, Coda file system, performance evaluation, disconnected operation, trickle reintegration, user-assisted cache management, file reference traces, trace replay, user patience modelling

Exploiting Weak Connectivity for Mobile File Access

Lily B. Mummert, Maria R. Ebling, M. Satyanarayanan
School of Computer Science
Carnegie Mellon University

Abstract

Weak connectivity, in the form of intermittent, low-bandwidth, or expensive networks is a fact of life in mobile computing. In this paper, we describe how the Coda File System has evolved to exploit such networks. The underlying theme of this evolution has been the systematic introduction of *adaptivity* to eliminate hidden assumptions about strong connectivity. Many aspects of the system, including communication, cache validation, update propagation and cache miss handling have been modified. As a result, Coda is able to provide good performance even when network bandwidth varies over four orders of magnitude — from modem speeds to LAN speeds.

1. Introduction

For the forseeable future, mobile clients will encounter a wide range of network characteristics in the course of their journeys. Cheap, reliable, high-performance connectivity via wired or wireless media will be limited to a few oases in a vast desert of poor connectivity. Mobile clients must therefore be able to use networks with rather unpleasant characteristics: intermittence, low bandwidth, high latency, or high expense. We refer to connectivity with one or more of these properties as *weak connectivity.* In contrast, typical LAN environments have none of these shortcomings and thus offer *strong connectivity.*

In this paper, we report on our work toward exploiting weak connectivity in the Coda File System. Our mechanisms preserve usability even at network speeds as low as 1.2 Kb/s. At a typical modem speed of 9.6 Kb/s, performance on a family of benchmarks is only about 2% slower than at 10 Mb/s. When a client reconnects to a network, synchronization of state with a server typically takes only about 25% longer at 9.6 Kb/s than at 10 Mb/s. To make better use of a network, Coda may solicit advice from the user. But it preserves usability by limiting the frequency of such interactions.

Since *disconnected operation* [13] represents an initial step toward supporting mobility, we begin by reviewing its strengths and weaknesses. We then describe a set of *adaptive* mechanisms that overcome these weaknesses by exploiting weak connectivity. Next, we evaluate these mechanisms through controlled experiments and empirical observations. Finally, we discuss related work and close with a summary of the main ideas.

This research was supported by the Air Force Materiel Command (AFMC) and ARPA under contract number F196828-93-C-0193. Additional support was provided by the IBM Corp., Digital Equipment Corp., Intel Corp., Xerox Corp., and AT&T Corp. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AFMC, ARPA, IBM, DEC, Intel, Xerox, AT&T, CMU, or the U.S. Government.

2. Starting Point: Disconnected Operation

2.1. Benefits and Limitations

Disconnected operation is a mode of operation in which a client continues to use data in its cache during temporary network or server failures. It can be viewed as the extreme case of weakly-connected operation — the mobile client is effectively using a network of zero bandwidth and infinite latency.

The ability to operate disconnected can be useful even when connectivity is available. For example, disconnected operation can extend battery life by avoiding wireless transmission and reception. It can reduce network charges, an important feature when rates are high. It allows radio silence to be maintained, a vital capability in military applications. And, of course, it is a viable fallback position when network characteristics degrade beyond usability.

But disconnected operation is not a panacea. A disconnected client suffers from many limitations:

- *Updates are not visible* to other clients.
- *Cache misses* may impede progress.
- *Updates are at risk* due to theft, loss or damage.
- *Update conflicts* become more likely.
- *Exhaustion of cache space* is a concern.

Our goal is to alleviate these limitations by exploiting weak connectivity. How successful we are depends on the quality of the network. With a very weak connection, a user is little better off than when disconnected; as network quality improves, the limitations decrease in severity and eventually vanish.

To attain this goal, we have implemented a series of modifications to Coda. Since Coda has been extensively described in the literature [13, 25, 26], we provide only a brief review here.

2.2. Implementation in Coda

Coda preserves the model of security, scalability, and Unix compatibility of AFS [5], and achieves high availability through the use of two complementary mechanisms. One mechanism is disconnected operation. The other mechanism is *server replication,* which we do not discuss further in this paper because it is incidental to our focus on mobility.

A small collection of trusted Coda servers exports a location-transparent Unix file name space to a much larger collection of untrusted clients. These clients are assumed to be general-purpose computers rather than limited-function devices such as InfoPads [28] and ParcTabs [29]. Files are grouped into *volumes,* each forming a partial subtree of the name space and typically containing the files of one user or project. On each client, a user-level process, *Venus,* manages a file cache on the local disk. It is Venus that bears the brunt of disconnected operation.

As described by Kistler [13], Venus operates in one of three states: *hoarding, emulating,* and *reintegrating.* It is normally in the hoarding state, preserving cache coherence via *callbacks* [5]. Upon disconnection, Venus enters the emulating state and begins logging updates in a *client modify log (CML).* In this state, Venus performs *log optimizations* to improve performance and reduce resource usage. Upon reconnection, Venus enters the reintegrating state, synchronizes its cache with servers, propagates updates from the CML, and returns to the hoarding state. Since consistency is based on optimistic replica control, update conflicts may occur upon reintegration. The system ensures their detection and confinement, and provides mechanisms to help users recover from them [14].

In anticipation of disconnection, users may *hoard* data in the cache by providing a prioritized list of files in a per-client *hoard database (HDB).* Venus combines HDB information with LRU information to implement a cache management policy addressing both performance and availability concerns. Periodically, Venus *walks* the cache to ensure that the highest priority items are present, and consistent with the servers. A user may also explicitly request a hoard walk at any time.

3. Design Rationale and Overview

3.1. Strategy

We chose an incremental approach to extending Coda, relying on usage experience and measurements at each stage. The underlying theme of this evolution was the identification of hidden assumptions about strong connectivity, and their systematic elimination through the introduction of adaptivity.

Our design is based on four guiding 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 weakly-connected client must be contacted for token revocation or data propagation before other clients can proceed.

- *Don't make life worse than when disconnected.*
  While a minor performance penalty may be an acceptable price for the benefits of weakly-connected operation, a user is unlikely to tolerate substantial performance degradation.

- *Do it in the background if you can.*
  Network delays in the foreground affect a user more acutely than those in the background. As bandwidth decreases, network usage should be moved into the background whenever possible. The effect of this strategy is to replace intolerable performance delays by a degradation of availability or consistency — lesser evils in many situations.

- *When in doubt, seek user advice.*
  As connectivity weakens, the higher performance penalty for suboptimal decisions increases the value of user advice. Users also make mistakes, of course, but they tend to be more forgiving if they perceive themselves responsible. The system should perform better if the user gives good advice, but should be able to function unaided.

More generally, we were strongly influenced by two classic principles of system design: favoring *simplicity* over unwarranted generality [15], and respecting the *end-to-end argument* when layering functionality [24].

3.2. Evolution

We began by modifying Coda's RPC and bulk transfer protocols to function over a serial-line IP (SLIP) connection [23]. These modifications were necessary because the protocols had been originally designed for good LAN performance. Once they functioned robustly down to 1.2 Kb/s, we had a reliable means of reintegrating and servicing critical cache misses from any location with a phone connection. Performance was atrocious because Venus used the SLIP connection like a LAN. But users were grateful for even this limited functionality, because the alternative would have been a significant commute to connect to a high-speed network.

Phone reintegration turned out to be much slower than even the most pessimistic of our estimates. The culprit was the validation of cache state on the first hoard walk after reconnection. Our solution raises the granularity at which cache coherence is maintained. In most cases, this renders the time for validation imperceptible even at modem speeds.

Next, we reduced update propagation delays by allowing a user to be logically disconnected while remaining physically connected. In this mode of use, Venus logged updates in the CML but continued to service cache misses. It was the user's responsibility to periodically initiate reintegration. Cache misses hurt performance, but there were few of them if the user had done a good job of hoarding.

Our next step was to eliminate manual triggering of reintegration when weakly connected. Since this removed user control over an important component of network usage, we had to be confident that a completely automated strategy could perform well even on very slow networks. This indeed proved possible, using a technique called *trickle reintegration.*

The last phase of our work was to substantially improve the handling of cache misses when weakly connected. Examination of misses showed that they varied widely in importance and cause. We did not see a way of automating the handling of all misses while preserving usability. So we decided to handle a subset of the misses transparently, and to provide users with a means of influencing the handling of the rest.

As a result of this evolution, Coda is now able to effectively exploit networks of low bandwidth and intermittent connectivity. Venus and the transport protocols transparently adapt to variations in network bandwidth spanning nearly four orders of magnitude — from a few Kb/s to 10 Mb/s. From a performance perspective, the user is well insulated from this variation. Network quality manifests itself mainly in the promptness with which updates are propagated, and in the degree of transparency with which cache misses are handled.

4. Detailed Design and Implementation

We provide more detail on four aspects of our system:

- Transport protocol refinements.
- Rapid cache validation.
- Trickle reintegration.
- User-assisted miss handling.

Although rapid cache validation has been described in detail an earlier paper [18], we provide a brief summary here for completeness. We also augment our earlier evaluation with measurements from the deployed system. The other three aspects of Coda are described here for the first time.

4.1. Transport Protocol Refinements

Coda uses the *RPC2* remote procedure call mechanism [27], which performs efficient transfer of file contents through a specialized streaming protocol called *SFTP*. Both RPC2 and SFTP are implemented on top of UDP. We made two major changes to them for slow networks.

One change addressed the isolation between RPC2 and SFTP. While this isolation made for clean code separation, it generated duplicate keepalive traffic. In addition, Venus generated its own higher-level keepalive traffic. Our fix was to share keepalive information between RPC2 and SFTP, and to export this information to Venus.

The other change was to modify RPC2 and SFTP to monitor network speed by estimating round trip times (*RTT*) using an adaptation of the timestamp echoing technique proposed by Jacobson [10]. The RTT estimates are used to dynamically adapt the retransmission parameters of RPC2 and SFTP. Our strategy is broadly consistent with Jacobson's recommendations for TCP [8].

With these changes, RPC2 and SFTP perform well over a wide range of network speeds. Figure 1 compares the performance of SFTP and TCP over three different networks: an Ethernet, a WaveLan wireless network, and a modem over a phone line. In almost all cases, SFTP's performance exceeds that of TCP.

Opportunities abound for further improvement to the transport protocols. For example, we could perform header compression as in TCP [9], and enhance the SLIP driver to prioritize traffic as described by Huston and Honeyman [7]. We could also enhance SFTP to ship file differences rather than full contents. But we have deliberately tried to minimize efforts at the transport level.

Protocol	Network	Nominal Speed	Receive (Kb/s)	Send (Kb/s)
TCP	Ethernet	10 Mb/s	1824 (64)	2400 (224)
	WaveLan	2 Mb/s	568 (136)	760 (80)
	Modem	9.6 Kb/s	6.8 (0.06)	6.4 (0.04)
SFTP	Ethernet	10 Mb/s	1952 (104)	2744 (96)
	WaveLan	2 Mb/s	1152 (64)	1168 (48)
	Modem	9.6 Kb/s	6.6 (0.02)	6.9 (0.02)

This table compares the observed throughputs of TCP and SFTP. The data was obtained by timing the disk-to-disk transfer of a 1MB file between a DECpc 425SL laptop client and a DEC 5000/200 server on an isolated network. Both client and server were running Mach 2.6. Each result is the mean of five trials. Numbers in parentheses are standard deviations.

Figure 1: Transport Protocol Performance

As the rest of this paper shows, mechanisms at higher levels of the system offer major benefits for weakly-connected operation. Additional transport level improvements may enhance those mechanisms, but cannot replace them.

4.2. Rapid Cache Validation

Coda's original technique for cache coherence while connected was based on *callbacks* [5, 25]. In this technique, a server remembers that a client has cached an object[1], and promises to notify it when the object is updated by another client. This promise is a *callback*, and the invalidation message is a *callback break*. When a callback break is received, the client discards the cached copy and refetches it on demand or at the next hoard walk.

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 the server.

4.2.1. Raising the Granularity of Cache Coherence

Our solution preserves the correctness of the original callback scheme, while dramatically reducing reconnection latency. It is based upon the observation that, in most cases, the vast majority of cached objects are still valid upon reconnection. The essence of our solution is for clients to track server state at multiple levels of *granularity*. Our current implementation uses only two levels: entire volume and individual object. We have not yet found the need to support additional levels.

A server now maintains version stamps for each of its volumes, in addition to stamps on individual objects. When an object is updated, the server increments 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 minimal cost.

When connectivity is restored, the client presents these volume stamps for validation. If a volume stamp is still valid, so is every object cached from that volume. In this case, validation of all

[1] For brevity, we use "object" to mean a file, directory, or symbolic link.

those objects has been achieved with a single RPC. We batch multiple volume validation requests in a single RPC for even faster validation. If a volume stamp is not valid, nothing can be assumed; each cached object from that volume must be validated individually. But even in this case, performance is no worse than in the original scheme.

4.2.2. Volume Callbacks

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, if present, or obtain them on demand.

Thus, volume callbacks improve speed of validation at the cost of precision of invalidation. This is an excellent performance tradeoff for typical Unix workloads [2, 19, 22]. Performance may be poorer with other workloads, but Coda's original cache coherence guarantees are still preserved.

4.3. Trickle Reintegration

Trickle reintegration is a mechanism that propagates updates to servers asynchronously, while minimally impacting foreground activity. Its purpose is to relieve users of the need to perform manual reintegration. The challenge is to meet this goal while remaining unobtrusive.

4.3.1. Relationship to Write-Back Caching

Trickle reintegration is conceptually similar to *write-back caching,* as used in systems such as Sprite [20] and Echo [16]. Both techniques strive to improve client performance by deferring the propagation of updates to servers. But they are sufficiently different in their details that it is appropriate to view them as distinct mechanisms.

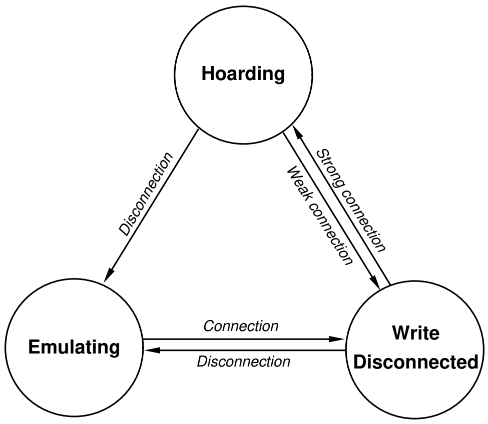
First, write-back caching preserves strict Unix write-sharing semantics, since it is typically intended for use in strongly-connected environments. In contrast, trickle reintegration has the opportunity to trade off consistency for performance because its users have already accepted the lower consistency offered by optimistic replication.

Second, the focus of write-back caching is minimizing file system latency; reducing network traffic is only an incidental concern. In contrast, reducing traffic is a prime concern of trickle reintegration because network bandwidth is precious.

Third, write-back caching schemes maintain their caches in volatile memory. Their need to bound damage due to a software crash typically limits the maximum delay before update propagation to some tens of seconds or a few minutes. In contrast, local persistence of updates on a Coda client is assured by the CML. Trickle reintegration can therefore defer propagation for many minutes or hours, bounded only by concerns of theft, loss, or disk damage.

4.3.2. Structural Modifications

Supporting trickle reintegration required major modifications to the structure of Venus. Reintegration was originally a transient state through which Venus passed *en route* to the hoarding 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 2 shows the new states of Venus and the main transitions between them.



This figure shows the states of Venus, as modified to handle weak connectivity. The state labelled "Write Disconnected" replaces the reintegrating state in our original design. 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.

Figure 2: Venus States and Transitions

As in our original design, Venus is in the hoarding state when strongly connected, and in the emulating state when disconnected. When weakly connected, it is in the write disconnected state. In this state, Venus' behavior is a blend of its connected and disconnected mode behaviors. Updates are logged, as when disconnected; they are propagated to servers via trickle reintegration. Cache misses are serviced, as when connected; but some misses may require user intervention. Cache coherence is maintained as explained earlier in Section 4.2.2.

A user can force a full reintegration at any time that she is in the write disconnected state. This might be valuable, for example, if she wishes to terminate a long distance phone call or realizes that she is about to move out of range of wireless communication. It is also valuable if she wishes to ensure that recent updates have been propagated to a server before notifying a collaborator via telephone, e-mail, or other out-of-band mechanism.

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, we ignore the callback break and proceed as usual. When reintegration of the object is eventually attempted, it may be resolved successfully or may fail. In the latter case, the conflict becomes visible to the user just as if it had occured after a disconnected session. The existing Coda mechanisms for conflict resolution [14] are then applied.

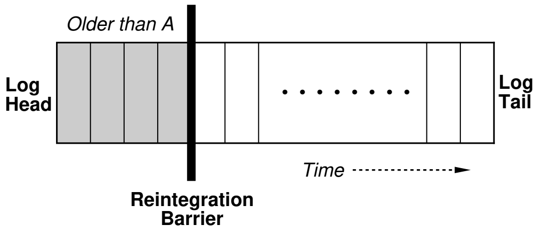
4.3.3. Preserving the Effectiveness of Log Optimizations

Early trace-driven simulations of Coda indicated that log optimizations were the key to reducing the volume of reintegration data [26]. Measurements of Coda in actual use confirm this prediction [21].

Applying log optimizations is conceptually simple. Except for `store` records, a CML record contains all the information needed to replay the corresponding update at the server. For a `store` record, the file data resides in the local file system. Before a log record is appended to the CML, Venus checks if it cancels or overrides the effect of earlier records. For example, consider the `create` of a file, followed by a `store`. If they are followed by an `unlink`, all three CML records and the data associated with the `store` can be eliminated.

Trickle reintegration reduces the effectiveness of log optimizations, because records are propagated to the server earlier than when disconnected. Thus they have less opportunity to be eliminated at the client. A good design must balance two factors. On the one hand, records should spend enough time in the CML for optimizations to be effective. On the other hand, updates should be propagated to servers with reasonable promptness. At very low bandwidths, the first concern is dominant since reduction of data volume is paramount. As bandwidth increases, the concerns become comparable in importance. When strongly connected, prompt propagation is the dominant concern.

Our solution, illustrated in Figure 3, uses a simple technique based on *aging.* A record is not eligible for reintegration until it has spent a minimal amount of time in the CML. This amount of time, called the *aging window*, (*A*), establishes a limit on the effectiveness of log optimizations.



This figure depicts a 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, the corresponding file data is locked; if contention occurs later, a shadow copy is created and the lock released.

Figure 3: CML During Trickle Reintegration

Since the CML is maintained in temporal order, the aging window partitions log records into two groups: those older than *A*, and those younger than *A*. Only the former group is eligible for reintegration. At the beginning of reintegration, a logical divider called the *reintegration barrier* is placed in the CML. During reintegration, which may take a while on a slow network, the portion of the CML to the left of the reintegration barrier is frozen. Only records to the right are examined for optimization.

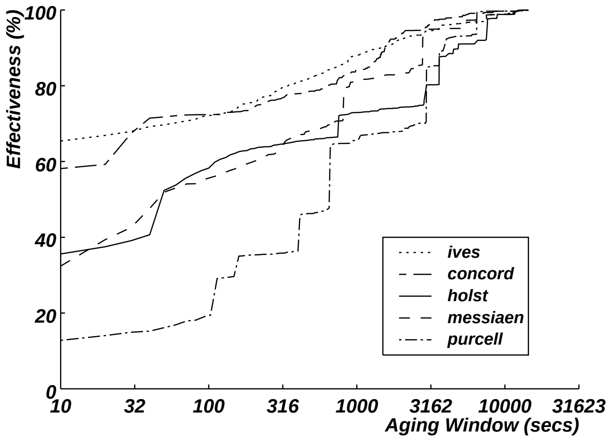
If reintegration is successful, the barrier and all records to its left are removed. If a network or server failure causes reintegration to be aborted, the barrier as well as any records rendered superfluous by new updates are removed. Our implementation of reintegration is atomic, ensuring that a failure leaves behind no server state that would hinder a future retry. Until the next reintegration attempt, all records in the CML are again eligible for optimization.

Discovery of records old enough for reintegration is done by a periodic daemon. Once the daemon finds CML records ripe for reintegration, it notifies a separate thread to do the actual work.

4.3.4. Selecting an Aging Window

What should the value of *A* be? To answer this question, we conducted a study using file reference traces gathered from workstations in our environment [19]. We chose five week-long traces (used in an earlier analysis [26]) in which there were extended periods of sustained high levels of user activity.

The traces were used as input to a Venus simulator. This simulator is the actual Venus code, modified to accept requests from a trace instead of the operating system. The output of the simulator includes the state of the CML at the end of the trace, and data on cancelled CML records. Our analysis includes all references from the traces, whether to the local file system, AFS, Coda, or NFS.



The X axis of this graph shows the aging window (*A*) on a logarithmic scale. Only CML records of age *A* or less are subject to optimization. Each curve corresponds to a different trace, and a point on a curve is the ratio of two quantities. The numerator is the amount of data saved by optimizations for the value of *A* at that point. The denominator is the savings when *A* is four hours (14,400 seconds). The value of the denominator is 84 MB for *ives*, 817 MB for *concord*, 40 MB for *holst*, 152 MB for *messiaen*, and 44 MB for *purcell*.

Figure 4: Effect of Aging on Optimizations

Figure 4 presents the results of our analysis. For each trace, this graph shows the impact of the aging window on the effectiveness of log optimizations. The results have been normalized with respect to a maximum aging window of four hours. We chose this period because it represents half a typical working day, and is a reasonable upper bound on the amount of work loss a user might be willing to tolerate.

The graph shows that there is considerable variation across traces. Values of *A* below 300 seconds barely yield an effectiveness of 30% on some traces, but they yield nearly 80% on others. For effectiveness above 80% on all traces, *A* must be nearly one hour. Since 600 seconds yields nearly 50% effectiveness on all traces, we have chosen it as the default value of *A*. This value can easily be changed by the user.

4.3.5. Reducing the Impact of Reintegration

Reintegrating all records older than *A* in one *chunk* could saturate a slow network for an extended period. The performance of a concurrent high priority network event, such as the servicing of a cache miss, could then be severely degraded. To avoid this problem, we have made reintegration chunk size adaptive.

The choice of a chunk size, (*C*), must strike a balance between two factors affecting performance. A large chunk size is more appropriate at high bandwidths because it amortizes the fixed costs of reintegration (such as transaction commitment at the server) over many log records. A small chunk size is better at low bandwidths because it reduces the maximum time of network contention. We have chosen a default value of 30 seconds for this time. 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.

Before initiating reintegration, we estimate *C* for the current bandwidth. We then select a maximal prefix of CML records whose age is greater than *A* and whose sizes sum to *C* or less. Most records are small, except for `store` records, whose sizes include that of the corresponding file data. In the limit, we select at least one record even if its size is larger than *C*. This prefix is the chunk for reintegration. The reintegration barrier is placed after it, and reintegration proceeds as described in Section 4.3.3. This procedure is repeated a chunk at a time, deferring between chunks to high priority network use, until all records older than *A* have been reintegrated.

With this procedure, the size of a chunk can be larger than *C* only when it consists of a single `store` record for a large file. In this case, we transfer the file as a series of *fragments* of size *C* or less. If a failure occurs, file transfer is resumed after the last successful fragment. Atomicity is preserved in spite of fragmentation because the server does not logically attempt reintegration until it has received the entire file. Note that this is the reverse of the procedure at strong connectivity, where the server verifies the logical soundness of updates before fetching file contents. The change in order reflects a change in the more likely cause of reintegration failure in the two scenarios.

We are considering a refinement that would allow a user to force immediate reintegration of updates to a specific directory or subtree, without waiting for propagation of other updates. Implementing this would require computing the precedence relationships between records, and ensuring that a record is not reintegrated before its antecedents. This computation is not necessary at present because the CML and every possible chunk are already in temporal order, which implies precedence order. We are awaiting usage experience to decide whether the benefits of this refinement merit its implementation cost.

4.4. Seeking User Advice

When weakly connected, the performance impact of cache misses is often too large to ignore. For example, a cache miss on a 1 MB file at 10 Mb/s can usually be serviced in a few seconds. At 9.6 Kb/s, the same miss causes a delay of nearly 20 minutes!

From a user's perspective, this lack of performance transparency can overshadow the functional transparency of caching. The problem is especially annoying because cache miss handling, unlike trickle reintegration, is a foreground activity. In most cases, a user would rather be told that a large file is missing than be forced to wait for it to be fetched over a weak connection.

But there are also situations where a file is so critical that a user is willing to suffer considerable delay. We refer to the maximum time that a user is willing to wait for a particular file as her *patience threshold* for that file. The need for user input arises because Venus has to find out how critical a missing object is.

Since the hoarding mechanism already provided a means of factoring user estimates of importance into cache management, it was the natural focal point of our efforts. Our extensions of this mechanism for weak connectivity are in two parts: an interactive facility to help augment the hoard database (HDB), and another to control the amount of data fetched during hoard walks. Together these changes have the effect of moving many cache miss delays into the background.

4.4.1. Handling Misses

When a miss occurs, Venus estimates its service time from the current network bandwidth and the object's size (as given by its status information). If the object's status information is not already cached, Venus obtains it from the server. The delay for this is acceptable even on slow networks because status information is only about 100 bytes long.

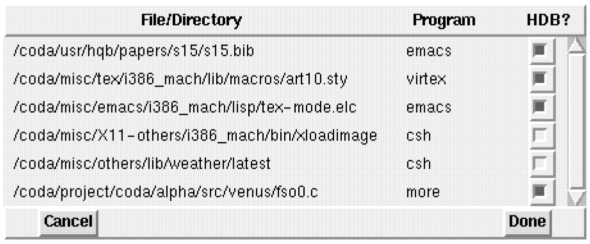
The estimated service time is then compared with the patience threshold. If the service time is below the threshold, Venus transparently services the miss. If the threshold is exceeded, Venus returns a cache miss error and records the miss.

4.4.2. Augmenting the Hoard Database

At any time, a user can ask Venus to show her all the misses that have occured since the previous such request. Venus displays each miss along with contextual information, as shown in Figure 5. The user can then select objects to be added to the HDB. This action does not immediately fetch the object; that is deferred until a future hoard walk. Hoard walks occur once every 10 minutes, or by explicit user request.

4.4.3. Controlling Hoard Walks

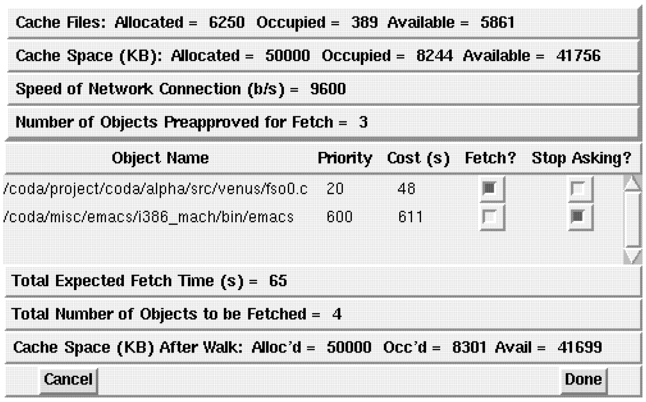
A hoard walk is executed in two phases. In the first phase, called the *status walk*, Venus obtains status information for missing objects and determines which objects, if any, should be fetched. Because of volume callbacks, the status walk usually involves little network traffic. During the second phase, called the *data walk*, Venus fetches the contents of objects selected by the status walk. Even if there are only a few large objects to be fetched, this phase can be a substantial source of network traffic.



This screen shows the name of each missing object and the program that referenced it. To add an object to the HDB, the user clicks the button to its right. A pop-up form (not shown here) allows the user to specify the hoard priority of the object and other related information.

Figure 5: Augmenting the Hoard Database

By introducing an interactive phase between the status and data walks, we allow users to limit the volume of data fetched in the data walk. Each object whose estimated service time is below the user's patience threshold is pre-approved for fetching. The fetching of other objects must be explicitly approved by the user.



This screen enables the user to suppress fetching of objects selectively during a hoard walk. The priority and estimated service time of each object are shown. The user approves the fetch of an object by clicking on its "Fetch" button. By clicking on its "Stop Asking" button, she can prevent the prompt and fetch for that object until strongly connected. The cache state that would result from the data walk is shown at the bottom of the screen. This information is updated as the user clicks on "Fetch" buttons.

Figure 6: Controlling the Data Walk

Figure 6 shows an example of the screen displayed by Venus between the status and data walks. If no input is provided by the user within a certain time, the screen disappears and all the listed objects are fetched. This handles the case where the client is running unattended.

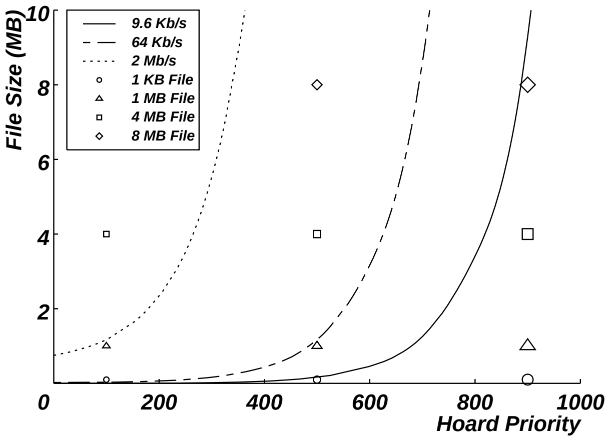
4.4.4. Modelling User Patience

Our goal in modelling user patience is to improve usability by reducing the frequency of user interaction. In those cases where we can predict a user's response with reasonable confidence, we can avoid the corresponding interactions. As mentioned earlier, a user's patience threshold, (τ), depends on how important she perceives an object to be: for a very important object, she is probably willing to wait many minutes.

Since user perception of importance is the notion captured by the hoard priority, (P), of an object, we posit that τ should be a function of P. At present, we are not aware of any data that could be the scientific basis for establishing the form of this relationship. Hence we use a function based solely on intuition, but have structured the implementation to make it easy to substitute a better alternative.

We conjecture that patience is similar to other human processes such as vision, whose sensitivity is logarithmic [3]. This suggests a relationship of the form $\tau = \alpha + \beta e^{\gamma P}$, where β and γ are scaling parameters and α represents a lower bound on patience. Even if an object is unimportant, the user prefers to tolerate a delay of α rather than dealing with a cache miss. We chose parameter settings based on their ability to yield plausible patience values for files commonly found in the hoard profiles of Coda users. The values we chose were α = 2 seconds, β = 1, γ = 0.01.

Figure 7 illustrates the resulting model of user patience. Rather than expressing τ in terms of seconds, we have converted it into the size of the largest file that can be fetched in that time at a given bandwidth. For example, 60 seconds at a bandwidth of 64 Kb/s yields a maximum file size of 480KB. Each curve in Figure 7 shows τ as a function of P for a given bandwidth. In the region below this curve, cache misses are transparently handled and pre-approval is granted during hoard walks.



Each curve in this graph expresses patience threshold, (τ), in terms of file size. Superimposed on these curves are points representing files of various sizes hoarded at priorities 100, 500, and 900. At 9.6 Kb/s, only the files at priority 900 and the 1KB file at priority 500 are below τ. At 64 Kb/s, the 1MB file at priority 500 is also below τ. At 2Mb/s, all files except the 4MB and 8MB files at priority 100 are below τ.

Figure 7: Patience Threshold versus Hoard Priority

The user patience model is the source of adaptivity in cache miss handling. It maintains usability at all bandwidths by balancing two factors that intrude upon transparency. At very low bandwidths, the delays in fetching large files annoy users more than the need for interaction. As bandwidth rises, delays shrink and interaction becomes more annoying. To preserve usability, we handle more cases transparently. In the limit, at strong connectivity, cache misses are fully transparent.

5. Deployment Status

The mechanisms described in this paper are being deployed to a user community of Coda developers and other computer science researchers. We have over 40 user accounts, of which about 25 are used regularly. Many users run Coda on both their desktop workstations and their laptops. We have a total of about 35 Coda clients, evenly divided between workstations and laptops. These clients access almost 4.0 GB of data stored on Coda servers.

The evolution described in Section 3.2 has spanned over two years. Early usage experience with each mechanism was invaluable in guiding further development. The transport protocol extensions were implemented in early 1993, and incorporated into the deployed system later that year. The rapid cache validation mechanism was implemented in late 1993, and has been deployed since early 1994. The trickle reintegration and user advice mechanisms were implemented between 1994 and early 1995, and have been released for general use.

6. Evaluation

6.1. Rapid Cache Validation

Two questions best characterize our evaluation of Coda's rapid cache validation mechanism:

- Under ideal conditions, how much do volume callbacks improve cache validation time?
- In practice, how close are conditions to ideal?

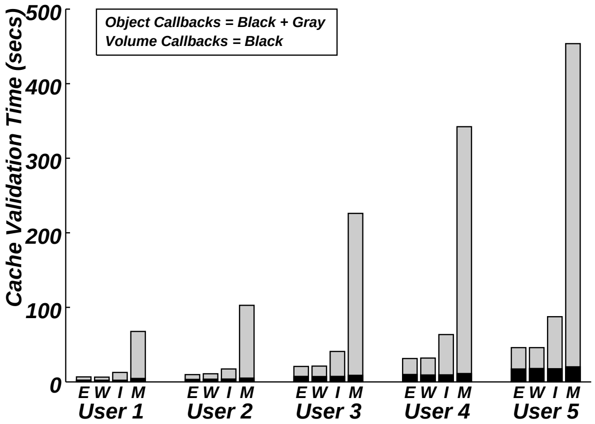
The first question was discussed in detail in an earlier paper [18]. Hence, we only present a brief summary of the key results. More recently, we have addressed the second question, and present the detailed results here.

6.1.1. Performance Under Ideal Conditions

For a given set of cached objects, the time for validation is minimal when two conditions hold. First, at disconnection, volume callbacks must exist for all cached objects. Second, while disconnected, the volumes containing these objects must not be updated at the server. Then, upon reconnection, communication is needed only to verify volume version stamps. Fresh volume callbacks are acquired as a side effect, at no additional cost.

Under these conditions, the primary determinants of performance are network bandwidth and the composition of cache contents. We conducted experiments to measure validation time as a function of these two variables. To study variation due to cache composition, we used the hoard profiles of five typical Coda users. To vary bandwidth, we used a network emulator.

Figure 8 shows that for all users, and at all bandwidths, volume callbacks reduce cache validation time. The reduction is modest at high bandwidths, but becomes substantial as bandwidth decreases. At 9.6 Kb/s, the improvement is dramatic, typically taking only about 25% longer than at 10 Mb/s.



This figure compares the time for validation using object and volume callbacks. Cache contents were determined by the hoard profiles of five Coda users. The network speeds correspond to the nominal speeds of Ethernet (**E**, 10 Mb/s), WaveLan (**W**, 2 Mb/s), ISDN (**I**, 64 Kb/s), and Modem (**M**, 9.6 Kb/s). The client and server were DECstation 5000/200s running Mach 2.6. Bandwidth was varied using an emulator on Ethernet.

Figure 8: Validation Time Under Ideal Conditions

6.1.2. Conditions Observed in Practice

There are two ways in which a Coda client in actual use may find conditions less than ideal. First, a client may not possess volume stamps for some objects at disconnection. If frequent, this event would indicate that our strategy of waiting for a hoard walk to acquire volume callbacks is not aggressive enough. Second, a volume stamp may prove to be stale when presented for validation. This would mean that the volume was updated on the server while the client was disconnected. If frequent, this event would indicate that acquiring volume stamps is futile, because it rarely speeds up validation. It could also be symptomatic of a volume being too large a granularity for cache coherence, for reasons analogous to false sharing in virtual memory systems with too large a page size.

To understand how serious these concerns are, we instrumented Coda clients to record cache validation statistics. Figure 9 presents data gathered from 26 clients. The data shows that our fears were baseless. On average, clients found themselves without a volume stamp only in 3% of the cases. The data on successful validations is even more reassuring. Most success rates were over 97%, and each successful validation saved roughly 53 individual validations.

6.2. Trickle Reintegration

How much is a typical user's update activity slowed when weakly connected? This is the question most germane to trickle reintegration, because the answer will reveal how effectively foreground activity is insulated from update propagation over slow networks.

The simplest way to answer this question would be to run a standard file system benchmark on a write-disconnected client over a wide range of network speeds. The obvious candidate is the *Andrew benchmark* [5] since it is compact, portable, and widely used. Unfortunately, this benchmark is of limited value in evaluating trickle reintegration.

Client	Missing Stamp	Validation Attempts	Fraction Successful	Objs per Success
bach	2%	970	99%	89
berlioz	8%	1178	97%	48
brahms	0%	542	99%	5
chopin	4%	1674	97%	102
copland	3%	1387	94%	171
dvorak	2%	5536	98%	75
gershwin	11%	467	95%	32
gs125	0%	897	99%	22
holst	0%	474	99%	29
ives	0%	1532	98%	56
mahler	1%	566	97%	6
messiaen	0%	827	98%	31
mozart	1%	1633	98%	126
varicose	0%	568	98%	32
verdi	6%	2370	98%	64
vivaldi	7%	344	89%	28
Mean	**3%**	**1310**	**97%**	**57**

(a) Desktops

Client	Missing Stamp	Validation Attempts	Fraction Successful	Objs per Success
caractacus	2%	650	97%	40
deidamia	2%	2257	98%	112
finlandia	13%	541	99%	32
gloriana	2%	1457	97%	29
guntram	0%	2977	99%	26
nabucco	1%	1301	96%	28
prometheus	6%	1617	97%	74
serse	8%	1790	98%	32
tosca	1%	652	99%	60
valkyrie	4%	759	96%	32
Mean	**4%**	**1400**	**98%**	**47**

(b) Laptops

These tables present data collected for approximately four weeks in July and August 1995 from 16 desktops and 10 laptops. The first column indicates how often validation could not be attempted because of a missing volume stamp. The last column gives a per-client average of object validations saved by a successful volume validation.

Figure 9: Observed Volume Validation Statistics

First, the running time of the benchmark on current hardware is very small, typically less than three minutes. This implies that no updates would be propagated to the server during an entire run of the benchmark for any reasonable aging window. Increasing the total time by using multiple iterations is not satisfactory because the benchmark is not idempotent. Second, although the benchmark captures many aspects of typical user activity, it does not exhibit overwrite cancellations. Hence, its file references are only marginally affected by log optimizations. Third, the benchmark involves no user think time, which we believe to be atypical of mobile computing applications.

For these reasons, our evaluation of trickle reintegration is based on *trace replay,* which is likely to be a much better indicator of performance in real use.

6.2.1. Trace Replay: Experiment Design

The ultimate in realism would be to measure trickle reintegration in actual use by mobile users. But this approach has serious shortcomings. First, a human subject cannot be made to repeat her behavior precisely enough for multiple runs of an experiment. Second, many confounding factors make timing results from actual use difficult to interpret. Third, such experiments cannot be replicated at other sites or in the future.

To overcome these limitations, we have developed an experimental methodology in which trace replay is used in lieu of human subjects. Realism is preserved since the trace was generated in actual use. Timing measurements are much less ambiguous, since experimental control and replicability are easier to achieve. The traces and the replay software can be exported.

Note that a trace replay experiment differs from a trace-driven simulation in that traces are replayed on a live system. Our replay software [19] generates Unix system calls that are serviced by Venus and the servers just as if they had been generated by a human user. The only difference is that a single process performs the replay, whereas the trace may have been generated by multiple processes. It would be fairly simple to extend our replay software to exactly emulate the original process structure.

How does one incorporate the effect of human think time in a trace replay experiment? Since a trace is often used many months or years after it was collected, the system on which it is replayed may be much faster than the original. But a faster system will not speed up those delays in the trace that were caused by human think time. Unfortunately, it is difficult to reliably distinguish think time delays from system-limited delays in a trace.

Our solution is to perform sensitivity analysis for think time, using a parameter called *think threshold*, (λ). This parameter defines the smallest delay in the input trace that will be preserved in the replay. When λ is 0, all delays are preserved; when it is infinity, the trace is replayed as fast as possible.

We rejected both extremities as parameter values for our experiments. At $\lambda = 0$, there is so much opportunity for overlapping data transmission with think time that experiments would be biased too much in favor of trickle reintegration. At λ = infinity, the absence of think time makes the experiment as unrealistic as the Andrew benchmark. In the light of these considerations, we chose values of λ equal to 1 second and 10 seconds for our experiments. These are plausible values for typical think times during periods of high activity, and they are not biased too far against or in favor of trickle reintegration.

Since log optimizations play such a critical role in trickle reintegration, we also conducted a sensitivity analysis for this factor. We divided the traces mentioned in Section 4.3.4 into 45-minute segments, selected segments with the highest activity levels, and analyzed their susceptibility to log optimizations. A segment longer than 45 minutes would have made the duration of each experiment excessive, allowing us to explore only a few parameter combinations.

We define the *compressibility* of a trace segment as the ratio of two quantities obtained when the segment is run through the Venus simulator. The numerator is the amount of data optimized out; the denominator is the length of the unoptimized CML. Figure 10 shows the observed distribution of compressibility in those trace segments with a final CML of 1MB or greater.

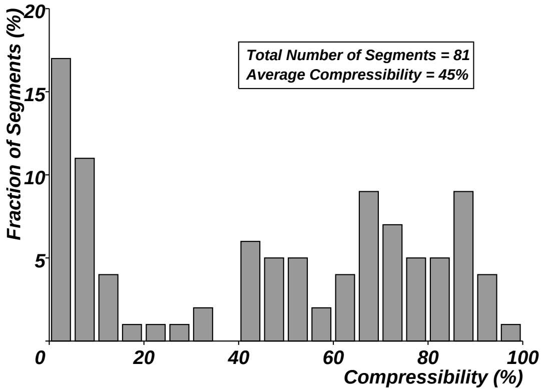


Figure 10: Compressibility of Trace Segments

The data shows that the compressibilities of roughly a third of the segments are below 20%, while those of the remaining two-thirds range from 40% to 100%. For our experiments, we chose one segment from each quartile of compressibility. The characteristics of these segments are shown in Figure 11.

Trace Segment	No. of References	No. of Updates	Unopt. CML (KB)	Opt. CML (KB)	Compress-ibility
Purcell	51681	519	2864	2625	8%
Holst	61019	596	3402	2302	32%
Messiaen	38342	188	6996	2184	69%
Concord	160397	1273	34704	2247	94%

Each of these segments is 45 minutes long. Since Coda uses the `open-close` session semantics of AFS, individual `read` and `write` operations are not included. Hence "Updates" in this table only refers to operations such as `close` after write, and `mkdir`. "References" includes, in addition, operations such as `close` after read, `stat`, and `lookup`.

Figure 11: Segments Used in Trace Replay Experiments

6.2.2. Trace Replay: Results

Figure 12 presents the results of our trace replay experiments. The same data is graphically illustrated in Figure 13. To ensure a fair comparison, we forced Venus to remain write disconnected at all bandwidths. We also deferred the beginning of measurements until 10 minutes into each run, thus warming the CML for trickle reintegration. The choice of 10 minutes corresponds to the largest value of *A* used in our experiments.

Figures 12 and 13 cover 64 combinations of experimental parameters: two aging windows (A = 300 and 600 seconds), two think thresholds (λ = 1 and 10 seconds), four trace compressibilities (8, 32, 69, and 94%), and four bandwidths (10 Mb/s, 2 Mb/s, 64 Kb/s, and 9.6 Kb/s).

These measurements confirm the effectiveness of trickle reintegration over the entire experimental range. Bandwidth varies over three orders of magnitude, yet elapsed time remains almost unchanged. On average, performance is only about 2% slower at 9.6 Kb/s than at 10 Mb/s. Even the worst case, corresponding to the Ethernet and ISDN numbers for Concord in Figure 12(d), is only 11% slower.

Trickle reintegration achieves insulation from network bandwidth by decoupling updates from their propagation to servers. Figure 14 illustrates this decoupling for one combination of λ and A. As bandwidth decreases, so does the amount of data shipped. For example, in Figure 14(b), the data shipped decreases from 2254 KB for Ethernet to 1536 KB for Modem. Since data spends more time in the CML, there is greater opportunity for optimization: 1067 KB versus 1081 KB. At the end of the experiment, more data remains in the CML at lower bandwidths: 70KB versus 2289 KB.

7. Related Work

Effective use of low bandwidth networks has been widely recognized as a vital capability for mobile computing [4, 11], but only a few systems currently provide this functionality. Of these, Little Work [6] is most closely related to our system.

Like Coda, Little Work provides transparent Unix file access to disconnected and weakly-connected clients, and makes use of log optimizations. But, for reasons of upward compatibility, it makes no changes to the AFS client-server interface. This constraint hurts its ability to cope with intermittent connectivity. First, it renders the use of large-granularity cache coherence infeasible. Second, it weakens fault tolerance because transactional support for reintegration cannot be added to the server.

Little Work supports *partially connected operation* [7], which is analogous to Coda's write disconnected state. But there are important differences. First, users cannot influence the servicing of cache misses in Little Work. Second, update propagation is less adaptive than trickle reintegration in Coda. Third, much of Little Work's efforts to reduce and prioritize network traffic occur in the SLIP driver. This is in contrast to Coda's emphasis on the higher levels of the system.

AirAccess 2.0 is a recent product that provides access to Novell and other DOS file servers over low-bandwidth networks [1]. Its implementation focuses on the lower levels, using techniques such as data compression and differential file transfer. Like Little Work, it preserves upward compatibility with existing servers and therefore suffers from the same limitations. AirAccess has no analog of trickle reintegration, nor does it allow users to influence the handling of cache misses.

From a broader perspective, application packages such as Lotus Notes [12] and cc:Mail [17] allow use of low-bandwidth networks. These systems differ from Coda in that support for mobility is entirely the responsibilty of the application. By providing this support at the file system level, Coda obviates the need to modify individual applications. Further, by mediating the resource demands of concurrent applications, Coda can better manage resources such as network bandwidth and cache space.

Trace Segment	Ethernet 10 Mb/s	WaveLan 2 Mb/s	ISDN 64 Kb/s	Modem 9.6 Kb/s
Purcell	2025 (16)	1999 (15)	2002 (20)	2096 (32)
Holst	1960 (3)	1961 (5)	1964 (5)	1983 (5)
Messiaen	1950 (2)	1970 (9)	1959 (3)	1995 (6)
Concord	1897 (9)	1952 (20)	1954 (43)	2002 (13)

(a) λ = 1 second, A = 300 seconds

Trace Segment	Ethernet 10 Mb/s	WaveLan 2 Mb/s	ISDN 64 Kb/s	Modem 9.6 Kb/s
Purcell	2086 (28)	2064 (6)	2026 (20)	2031 (4)
Holst	2004 (13)	1984 (11)	1970 (11)	2009 (17)
Messiaen	1949 (2)	1974 (8)	1969 (16)	1986 (3)
Concord	2078 (49)	2051 (38)	2017 (39)	2079 (10)

(b) λ = 1 second, A = 600 seconds

Trace Segment	Ethernet 10 Mb/s	WaveLan 2 Mb/s	ISDN 64 Kb/s	Modem 9.6 Kb/s
Purcell	1747 (20)	1622 (12)	1624 (5)	1744 (8)
Holst	1026 (6)	1000 (3)	1005 (10)	1047 (2)
Messiaen	1234 (2)	1241 (2)	1238 (5)	1278 (9)
Concord	1254 (7)	1323 (16)	1312 (17)	1362 (18)

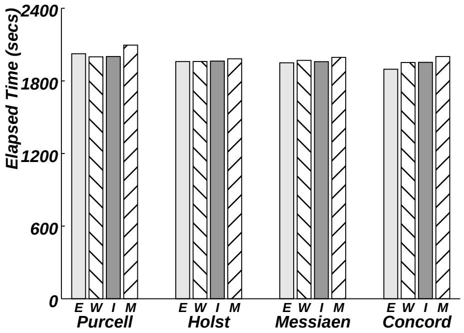
(c) λ = 10 seconds, A = 300 seconds

Trace Segment	Ethernet 10 Mb/s	WaveLan 2 Mb/s	ISDN 64 Kb/s	Modem 9.6 Kb/s
Purcell	1704 (9)	1658 (14)	1664 (23)	1683 (16)
Holst	1060 (10)	1027 (8)	1021 (8)	998 (3)
Messiaen	1234 (3)	1265 (13)	1263 (11)	1279 (7)
Concord	1258 (7)	1383 (27)	1402 (30)	1340 (16)

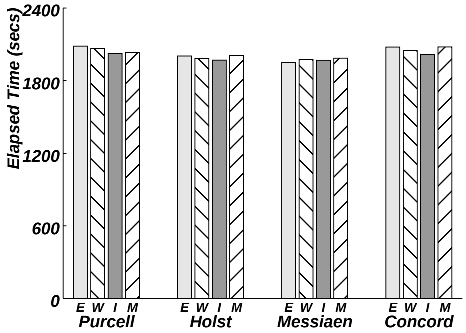
(d) λ = 10 seconds, A = 600 seconds

This table presents the elapsed time, in seconds, of the trace replay experiments described in Section 6.2.1. The think threshold is λ, and the aging window is *A*. Each data point is the mean of five trials; figures in parentheses are standard deviations. The experiments were conducted using a DEC pc425SL laptop client and a DECstation 5000/200 server, both with 32 MB of memory, and running Mach 2.6. The client and server were isolated on a separate network. The Ethernet, WaveLan and Modem experiments used actual networks of the corresponding type. The ISDN experiments were conducted on an Ethernet using a network emulator. Measurements began after a 10 minute warming period.

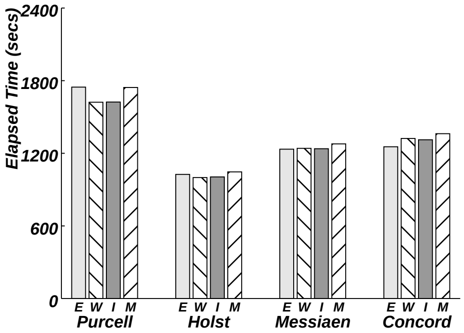
Figure 12: Performance of Trickle Reintegration on Trace Replay



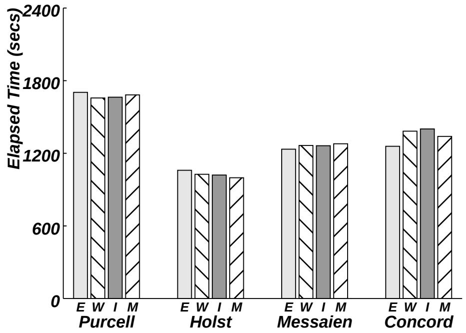
(a) λ = 1 second, A = 300 seconds



(b) λ = 1 second, A = 600 seconds



(c) λ = 10 seconds, A = 300 seconds



(d) λ = 10 seconds, A = 600 seconds

These graphs illustrate the data in Figure 12. Network speed is indicated by **E** (Ethernet), **W** (WaveLan), **I** (ISDN), or **M** (Modem).

Figure 13: Comparison of Trace Replay Times

Network Type	Begin CML (KB)	End CML (KB)	Shipped (KB)	Optimized (KB)
Ethernet	0 (0)	59 (0)	2618 (0)	238 (0)
WaveLan	0 (0)	59 (0)	2618 (0)	238 (0)
ISDN	0 (0)	2128 (0)	800 (105)	238 (0)
Modem	0 (0)	2538 (0)	114 (19)	238 (0)

(a) Trace Segment = Purcell

Network Type	Begin CML (KB)	End CML (KB)	Shipped (KB)	Optimized (KB)
Ethernet	2133 (0)	70 (0)	2254 (0)	1067 (0)
WaveLan	2133 (0)	70 (0)	2254 (0)	1067 (0)
ISDN	2133 (0)	70 (0)	2252 (0)	1069 (0)
Modem	2133 (0)	2289 (0)	1536 (68)	1081 (0)

(b) Trace Segment = Holst

Network Type	Begin CML (KB)	End CML (KB)	Shipped (KB)	Optimized (KB)
Ethernet	896 (0)	0 (0)	2270 (0)	3022 (0)
WaveLan	896 (0)	0 (0)	2270 (0)	3022 (0)
ISDN	896 (0)	0 (0)	2270 (0)	3022 (0)
Modem	896 (0)	1060 (0)	1309 (16)	3103 (0)

(c) Trace Segment = Messiaen

Network Type	Begin CML (KB)	End CML (KB)	Shipped (KB)	Optimized (KB)
Ethernet	63 (0)	2103 (0)	2496 (0)	30209 (0)
WaveLan	63 (0)	2103 (0)	2496 (0)	30209 (0)
ISDN	63 (0)	2103 (0)	2407 (0)	30291 (0)
Modem	63 (0)	2180 (0)	1142 (46)	32322 (0)

(d) Trace Segment = Concord

This table shows components of the data generated in the experiments of Figure 12(b). Results for other combinations of λ and A are comparable. The columns labelled "Begin CML" and "End CML" give the amount of data in the CML at the beginning and end of the measurement period. This corresponds to the amount of data waiting to be propagated to the servers at those times. The column labelled "Shipped" gives the amount of data actually transferred over the network; "Optimized" gives the amount of data saved by optimizations.

It may appear at first glance that the sum of the "End CML", "Shipped", and "Optimized" columns should equal the "Unopt. CML" column of Figure 11. But this need not be true for the following reasons. First, optimizations that occur prior to the measurement period are not included in "Optimized". Second, if an experiment ends while a large file is being transferred as a series of fragments, the fragments already transferred are counted both in the "End CML" and "Shipped" columns. Third, log records are larger when shipped than in the CML.

Figure 14: Data Generated During Trace Replay (λ = 1 second, A = 600 seconds)

8. Conclusion

Adaptation is the key to mobility. Coda's approach is best characterized as *application-transparent adaptation* — Venus bears full responsibility for coping with the demands of mobility. Applications remain unchanged, preserving upward compatibility.

The quest for adaptivity has resulted in major changes to many aspects of Coda, including communication, cache validation, update propagation, and cache miss handling. In making these changes, our preference has been to place functionality at higher levels of Venus, with only the bare minimum at the lowest levels. Consistent with the end-to-end argument, we believe that this is the best approach to achieving good performance and usability in mobile computing.

In its present form, Coda can use a wide range of communication media relevant to mobile computing. Examples include regular phone lines, cellular modems, wireless LANs, ISDN lines, and cellular digital packet data (CDPD) links. But Coda may require further modifications to use satellite networks, which have enormous delay-bandwidth products, and cable TV networks, whose bandwidth is asymmetric.

Our work so far has assumed that performance is the only metric of cost. In practice, many networks used in mobile computing cost real money. We therefore plan to explore techniques by which Venus can electronically inquire about network cost, and base its adaptation on both cost and quality. Of course, full-scale deployment of this capability will require the cooperation of network providers and regulatory agencies.

Weak connectivity is a fact of life in mobile computing. In this paper, we have shown how such connectivity can be exploited to benefit mobile users of a distributed file system. Our mechanisms allow users to focus on their work, largely ignoring the vagaries of network performance and reliability. While many further improvements will undoubtedly be made, Coda in its present form is already a potent and usable tool for exploiting weak connectivity in mobile computing.

Acknowledgements

Brian Noble, Puneet Kumar, David Eckhardt, Wayne Sawdon, and Randy Dean provided insightful comments that substantially strengthened this paper. Our SOSP shepherd, Mary Baker, was helpful in improving the presentation. Brent Welch helped with the TCL mud-wrestling involved in implementing user-assisted cache management.

This work builds upon the contributions of many past and present Coda project members. Perhaps the most important contribution of all has been made by the Coda user community, through its bold willingness to use and help improve an experimental system.

References

[1] *AirSoft AirAccess 2.0: Mobile Networking Software* AirSoft, Inc., Cupertino, CA, 1994.

[2] Baker, M.G., Hartmann, J.H., Kupfer, M.D., Shirriff, K.W., Ousterhout, J.K. Measurements of a Distributed File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. Pacific Grove, CA, October, 1991.

[3] Cornsweet, T.N. *Visual Perception.* Academic Press, 1971.

[4] Forman, G.H., Zahorjan, J. The Challenges of Mobile Computing. *IEEE Computer* 27(4), April, 1994.

[5] 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.

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

[7] Huston, L., Honeyman, P. Partially Connected Operation. In *Proceedings of the Second USENIX Symposium on Mobile and Location-Independent Computing*. Ann Arbor, MI, April, 1995.

[8] Jacobson, V. Congestion Avoidance and Control. In *Proceedings of SIGCOMM88*. Stanford, CA, August, 1988.

[9] Jacobson, V. RFC 1144: Compressing TCP/IP Headers for Low-Speed Serial Links. February, 1990.

[10] Jacobson, V., Braden, R., Borman, D. RFC 1323: TCP Extensions for High Performance. May, 1992.

[11] Katz, R.H. Adaptation and Mobility in Wireless Information Systems. *IEEE Personal Communications* 1(1), 1994.

[12] Kawell Jr., L., Beckhardt, S., Halvorsen, T., Ozzie, R., Greif, I. Replicated Document Management in a Group Communication System. In Marca, D., Bock, G. (editors), *Groupware: Software for Computer-Supported Cooperative Work*, pages 226-235. IEEE Computer Society Press, Los Alamitos, CA, 1992.

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

[14] Kumar, P. *Mitigating the Effects of Optimistic Replication in a Distributed File System*. PhD thesis, School of Computer Science, Carnegie Mellon University, December, 1994.

[15] Lampson, B. W. Hints for Computer System Design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*. Bretton Woods, NH, October, 1983.

[16] Mann, T., Birrell, A., Hisgen, A., Jerian, C., Swart, G. A Coherent Distributed File Cache with Directory Write-Behind. *ACM Transactions on Computer Systems* 12(2), May, 1994.

[17] Moeller, M. Lotus Opens cc:Mail to Pagers. *PC Week* 11(35):39, September, 1994.

[18] Mummert, L.B., Satyanarayanan, M. Large Granularity Cache Coherence for Intermittent Connectivity. In *Proceedings of the 1994 Summer USENIX Conference*. Boston, MA, June, 1994.

[19] Mummert, L.B., Satyanarayanan, M. *Long-Term Distributed File Reference Tracing: Implementation and Experience*. Technical Report CMU-CS-94-213, School of Computer Science, Carnegie Mellon University, November, 1994.

[20] Nelson, M.N., Welch, B.B., Ousterhout, J.K. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems* 6(1), February, 1988.

[21] Noble, B., Satyanarayanan, M. An Empirical Study of a Highly-Available File System. In *Proceedings of the 1994 ACM Sigmetrics Conference*. Nashville, TN, May, 1994.

[22] 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 Tenth ACM Symposium on Operating System Principles*. Orcas Island, WA, December, 1985.

[23] Romkey, J. RFC 1055: A Nonstandard for Transmission of IP Datagrams Over Serial Lines: SLIP. June, 1988.

[24] Saltzer, J.H., Reed, D.P., Clark, D.D. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems* 2(4), November, 1984.

[25] 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.

[26] Satyanarayanan, M., Kistler, J.J., Mummert, L.B., Ebling, M.R., Kumar, P., Lu, Q. Experience with Disconnected Operation in a Mobile Computing Environment. In *Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing*. Cambridge, MA, August, 1993.

[27] Satyanarayanan, M. (Editor). *RPC2 User Guide and Reference Manual* Department of Computer Science, Carnegie Mellon University, 1995 (Last revised).

[28] Sheng, S., Chandrakasan, A., Brodersen, R.W. A Portable Multimedia Terminal. *IEEE Communications Magazine* 30(12), December, 1992.

[29] Weiser, M. The Computer for the Twenty-First Century. *Scientific American* 265(3), September, 1991.