Large Granularity Cache Coherence for Intermittent Connectivity

L. Mummert M. Satyanarayanan

April 1994

CMU-CS-94-100

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in the Proceedings of the 1994 Summer USENIX Conference, June 8-10 1994, Boston MA

Abstract

To function in mobile computing environments, distributed file systems must cope with networks that are slow, intermittent, or both. Intermittence vitiates the effectiveness of callback-based cache coherence schemes in reducing client-server communication, because clients must validate files when connections are reestablished. In this paper we show how maintaining *cache coherence at a large granularity* alleviates this problem. We report on the implementation and performance of large granularity cache coherence for the Coda File System. Our measurements confirm the value of this technique. At 9.6 Kbps, this technique takes only 4 – 20% of the time required by two other strategies to validate the cache for a sample of Coda users. Even at this speed, the network is effectively eliminated as the bottleneck for cache validation.

Research is sponsored in part by the Air Force Materiel Command (AFMC) and the Advanced Research Projects Agency (ARPA) under contract number F19628-93-C-0193. Support also came from IBM Corporation, Digital Equipment Corporation, Intel Corporation, and Bellcore.

Keywords: distributed file systems, weak connectivity, mobile computing, caching, callbacks, volume callbacks, Coda

1 Introduction

Callback-based cache coherence [4, 10] in distributed file systems has proven to be invaluable for preserving scalability while maintaining a high degree of consistency. This technique is based on the implicit assumption that the network is fast and reliable. Unfortunately, this assumption is often violated in mobile computing environments. Network communication in those environments is often slow and intermittent.

Instead of requiring a client to check the validity of a file on each access, a callback-based scheme places greater responsibility on the server. When a client caches a file from a server, the server promises to notify it if the file changes. This promise is called a *callback*. An invalidation message is called a *callback break*. If a client receives a callback break for a file, it discards the cached copy and re-fetches it when it is next referenced.

When a client with callbacks encounters a network failure, it must consider its cached files suspect because it can no longer depend on the server to notify it of updates. Upon repair of the failure, the client must validate cached files before use. Consequently, as failures become more frequent, the effectiveness of a callback-based scheme in reducing validation traffic decreases. In the worst case, client behavior may degenerate to contacting the server on every reference. This problem is exacerbated in systems that use anticipatory caching strategies such as *hoarding* to prepare for failures [1, 5]. In these systems, validation traffic is proportional not just to the file *working set,* but to the larger *resident set.* The more diligent the preparation for failures, the larger the resident set. The impact of this problem increases as network bandwidth becomes precious.

We can address this problem without weakening consistency by increasing the *granularity* at which cache coherence is maintained. This makes validation more efficient, allowing clients to recover from failures more quickly. Taken to an extreme, this idea would require maintaining a version stamp and callback on the entire file system. If the version stamp remains unchanged after a failure, the client can be confident that no files have been updated on the server. A callback on the entire file system is a very strong statement — it means every file cached at the client is valid. However a callback break on the file system conveys little information — anything in the file system could have changed, whether cached at the client or not. A practical implementation of this idea requires a choice of granularity that balances speed of validation with precision of invalidation.

In this paper we report on the implementation and performance evaluation of large granularity cache coherence in the Coda file system [10]. Our results show that large granularity cache coherence is well-suited for a significant fraction of what Coda users typically cache. At 9.6 Kbps, this technique takes only 4 – 20% of the time required by two other strategies to validate the cache. Effectively, this technique eliminates the network as the bottleneck for cache validation. At higher bandwidths, the value of this technique diminishes, but it is always at least as good as the other strategies.

The paper begins by introducing key aspects of Coda. Then we describe the operation of the system with multiple granularities. Details concerning the actual implementation are presented in Section 4. Finally we describe the current status of our system and evaluate its performance.

2 Coda File System

Coda is a descendant of AFS-2[1] [4] that has *high data availability* as its main goal. Like AFS, it provides a single, shared, location-transparent name space, and maintains cache coherence using callbacks. Files are stored in *volumes* [11], each forming a partial subtree of the name space. Volumes are administrative units, typically created for individual users or projects. A user-level process called *Venus* manages a file cache on the local disk of each client. Venus makes requests of servers through the *Vice interface* using remote procedure calls (RPC). Files are identified by *fids*, which are 96 bits long. The first 32 bits of the fid are the volume identifier.

Coda uses two strategies to achieve high data availability: *server replication* and *disconnected operation*. Server replication allows volumes to be stored at a group of servers called the *volume storage group* (VSG). At any time, the subset of those servers available is called the *accessible volume storage group* (AVSG). When making requests, clients contact all servers in the AVSG (though data is fetched from only one), and all servers maintain callbacks for objects cached from the VSG. If an AVSG grows, clients drop callbacks for objects stored at that VSG, because the newly available server may contain more recent data. Further details on server replication may be found in [10].

[1] AFS has evolved since the version from which Coda was derived, which was AFS-2. The currently deployed version is AFS-3. Unless qualified, the term AFS applies to both versions.

Disconnected operation arises when the AVSG becomes empty. To prepare for disconnection, users may *hoard* data in the cache by providing a prioritized list of files called the *hoard database*, or HDB. Venus combines hoard database entries with LRU information as in traditional caching schemes to implement a cache management policy that addresses both performance and availability concerns. Periodically, Venus *walks* the cache to ensure that the highest priority items in the HDB are cached and consistent with the AVSG. Hoard walks may also be requested explicitly by the user. If an object in the HDB is invalidated, it is re-fetched on the next reference or during the next hoard walk, whichever comes first. Hoard walks can create bursty network traffic. A hoard walk after an AVSG grows results in a validation request for every cached object from that VSG. More details on hoarding and disconnected operation may be found in [5].

3 Protocol Description

At how many granularities should cache coherence be maintained? In principle there can be many levels. An obvious mapping onto a Unix file name space would suggest a hierarchy of granularities. But the desire for a simple implementation led us to use just two: files[2] and volumes. Volumes are attractive as units of coherence because they tend to represent groups of files that are logically related and hence possess similar update characteristics.

When a client maintains coherence on files, it must validate them before use when the AVSG has grown. This approach is based on the assumption that the newly available server has rendered some file in the cache stale, necessitating a check of each one. As failures become more frequent, the price of suspicion increases. Increasing the granularity of coherence allows a client to summarize the contents of its cache for the purpose of validation. This approach is more optimistic, in that it assumes there are sets of cached files that have not changed during the failure.

To summarize cache state by volume, servers maintain version stamps for each volume they store. The version stamp for a volume is incremented whenever an object in the volume is updated. A client caches the version stamp, establishing a callback for the volume. When the AVSG grows, the client validates the files in a volume by sending its version stamp to the server. If the stamps match, all of the client's cached data from the volume is valid. The server grants a callback for the volume to allow the client to read the cached files without any additional communication. If the validation fails the client reverts to file callbacks.

We expect maintaining coherence on volumes to be beneficial for collections owned by the primary user of a client, and for collections that don't change frequently or change *en masse* (e.g., system binaries) [8]. In Section 5, we show that such collections represent a large fraction of the files that users cache. File callbacks are more appropriate for volumes that are shared or owned by users other than the primary user of a client.

Of course, the client must ensure that version stamps are consistent with the data they represent, and it must handle updates from other clients, which manifest themselves as callback breaks. We discuss these issues further in the remainder of this section.

3.1 Obtaining Callbacks

A client caches a volume version stamp to prepare for the next failure. If a client presents an up-to-date stamp after a failure, it is granted a callback on the volume. The volume callback is a substitute for file callbacks on all the files in that volume. The callback is actually on the version stamp. It means the client has files corresponding to the version of the volume designated by the value of the stamp.

Before obtaining a volume version stamp, we require all files in the cache from that volume to be valid and have callbacks. This ensures the files at the client correspond to the version stamp it receives. Since validating the files could be expensive, the client should employ a policy that balances this cost with the expected value of having a volume version stamp in case of a failure. We discuss policy further in Section 4.3. For volume callbacks to be effective, there should be more than one file cached from the volume.

If the client holds a volume callback and fetches a new file, the server establishes a file callback for the new file. This is not necessary for correctness, but it is useful for performance. Although one could imagine not establishing the file callback to conserve server memory, granting the file callback in this case requires no additional network traffic, and gives the client something to fall back on should the volume callback be broken.

[2]In this paper, we use the term *file* to refer to single objects in the file system, including directories and symbolic links.

3.2 Handling Callback Breaks

When a file is updated by a remote client, the server breaks callbacks to all other clients holding a callback for that file or its volume. If a client holds callbacks on both the file and the volume, the server breaks the callback on the file. The client interprets this as a callback break on the volume as well, and erases its version stamp. Note that if a client holds a volume callback, it will receive a callback break even if the updated file is not in its cache. This is *false sharing*, and if frequent, may indicate that the granularity of cache coherence is too large for that volume. The client should not blindly reestablish the callback when it is broken, because updates exhibit temporal locality [2, 9]. Not only would this be a waste of bandwidth, but it would also harm scalability. The client's policy should take this into account when determining whether the volume callback should be reestablished.

The presence of both volume and file callbacks means clients must decide what kind of callback to obtain when one is broken. Suppose a client validates a version stamp for a volume, and it receives a volume callback. At this point it has no file callbacks. If the volume callback is broken, the client must validate its cached files from that volume before it can reestablish the volume callback. In terms of network usage, this is equivalent to recovery from a failure without volume callbacks. In effect, the client has delayed validation of individual files.

In the situation above one might imagine obtaining file callbacks in the background in case the volume callback is broken. This eager strategy assumes a remote update will occur before the next failure. However, this defeats the purpose of obtaining a volume callback. Instead, we employ a lazy strategy, obtaining file callbacks only if the volume callback is actually broken. If no remote updates occur between failures, we have saved the network bandwidth and server memory that would have been required to validate and obtain file callbacks.

4 Implementation

We layered volume callbacks on the existing callback mechanism as much as possible. Code changes were required in the Vice interface, the server, and Venus. We discuss these changes in the following subsections.

4.1 Vice Interface

We added two new calls to the Vice interface that manipulate version stamps, which were already being maintained by each server for replication. The first new call is `ViceGetVolVS`, which takes a volume identifier, and returns a version stamp and a flag indicating whether or not a callback has been established for the volume.

```
ViceGetVolVS(IN VolumeId Vid,
     OUT RPC2_Integer VS,
     OUT CallBackStatus CBStatus);
```

The second call, `ViceValidateVols`, takes a list of volume identifiers and version stamps and returns a code for each indicating if it is valid, and if so, whether a callback has been established for the volume. The structure `RPC2_CountedBS` consists of a length field and a sequence of bytes.

```
ViceValidateVols(
     IN ViceVolumeIdStruct Vids[],
     IN RPC2_CountedBS VS,
     OUT RPC2_CountedBS VFlagBS);
```

Besides the two new Vice calls, there are also new parameters to existing calls that perform updates (`mkdir`, `rename`, etc.).

4.2 Server side

Server code is required to support the new Vice RPCs, and volume callbacks themselves. We added about 400 lines of code to the server, which consists of approximately 14,500 lines of code excluding headers and libraries. Most of the changes involved supporting the new RPCs (200 lines) and debugging and printing statistics (150 lines). The remainder of the changes were for gathering statistics.

We minimized changes to data structures and code involving callbacks by designating an unused fid ($\langle$VolumeId$\rangle$.0.0) to represent an entire volume. We modified the callback break routine to break callbacks not only for a file, but also for the volume that contains it.

Updates change the volume version stamp, whether they are made remotely, or by a local client. When a client updates a file, it receives a status block containing the file's new version information and attributes. The status block is shown in Figure 1. Similarly, the client must be able to observe the effects of its updates on the volume version stamp, without receiving callback breaks or sending additional messages.

We considered two approaches for updating the client's version stamp when it performs an update – having the client compute the new stamp, or having the server compute and return it. The advantage of having the client compute the new stamp is no additional changes need to be made to the Vice interface. Unfortunately, since the server must maintain version stamps anyway, this approach duplicates a good deal of code, and is more difficult to test and maintain.

We chose to have the server compute and return the new version stamp. We have added three parameters to Vice calls that involve updates:

- the old version stamps
- the new version stamp
- the callback status

When a client performs an update, it sends its copy of the volume version stamp to the server along with the other parameters for the operation. If the client's stamp is current, the server returns the new stamp and a callback for the volume. If it is not, the server returns a zero stamp, and no callback. If the client does not have a stamp, or does not wish to obtain a volume callback, it simply sends a zero stamp. This is guaranteed never to match at the server.

This process is complicated by concurrency control. Files involved in an update are locked for the duration of the operation. For performance reasons, the server cannot lock a volume for the entire duration of an update. Therefore, it is possible for updates to different objects in a volume to be interleaved. To detect this, the server updates the client's version stamp along with its own, and checks for a match at the end of the call.

There are operations other than file updates that change volume version stamps. We made a few additional changes to two server libraries to ensure callbacks would be broken when these operations occurred. One of the libraries supports debugging; the other is part of the resolution subsystem [7].

Our implementation was complicated by a number of race conditions, pertaining to server replication, that manifested themselves during initial testing. These race conditions were present in the original AFS-2 code from which Coda is derived, but were triggered when clients eagerly acquired volume callbacks.

For example, the callback processing code is structured to prevent a server from adding a callback for a fid while breaking a callback for that fid. Callbacks are maintained at all servers in the AVSG. The race condition occurs when a client receives callback breaks from a subset of the AVSG and immediately tries to reestablish its volume callback. This request is sent to all the servers in the AVSG; this may include ones still breaking the callback. This used to cause the servers to crash. We fixed this by returning the callback status, and having the server not establish callbacks in this situation.

4.3 Client side

Most of the logic for supporting volume callbacks is in Venus. In addition to using the new RPCs, Venus must cope with replication, and decide when using volume callbacks is appropriate. The changes represented an addition of 700 lines to about 36,000 lines of code excluding headers and libraries.

The implementation of Venus is layered with respect to files and volumes. The changes for volume callbacks are concentrated in the volume layer, leaving the heart of Venus unchanged. We augmented the volume data structure to store a volume version stamp, the status of a volume callback, and summary statistics such as the number of callbacks established, broken, and cleared.

There are a number of background processes within Venus that run periodically. The hoard daemon, for example, runs a hoard walk every ten minutes. The volume daemon checks each volume to effect state changes every

```
typedef RPC2_Struct
{
RPC2_Unsigned        InterfaceVersion;
ViceDataType         VnodeType;
RPC2_Integer         LinkCount;
RPC2_Unsigned        Length;
FileVersion          DataVersion;
ViceVersionVector    VV;
Date                 Date;
UserId               Author;
UserId               Owner;
CallBackStatus       CallBack;
Rights               MyAccess;
Rights               AnyAccess;
RPC2_Unsigned        Mode;
VnodeId              vparent;
Unique               uparent;
}                    ViceStatus;
```

Figure 1: Vice Status Block

This figure shows the Vice status block, which is returned for the objects of most Vice calls. It includes version information for the object, whether or not the server has extended a callback promise for it, and the access rights of the requesting user and the anonymous user System:Anyuser.

---

five seconds. Our code is structured such that volume version stamps are likely to be obtained or validated in the background by one of these daemons. This greatly reduces the chance that the cost of these tasks is incurred on demand during a user request.

4.3.1 Policy

As mentioned in Section 3, Venus should have some policy to determine when to obtain a volume callback. The optimal policy would obtain a volume callback only if a failure was going to occur and be repaired before the next remote update. Otherwise, either the volume callback would be broken, or the next validation would fail.

One could invent a variety of policies to approximate the optimal one. We decided to use a simple policy, in which Venus obtains volume callbacks only during hoard walks. We chose this policy for several reasons.

1. Volume version stamps are intended to be useful in preparing for failures. This is synonymous with the purpose of hoarding.
2. During a hoard walk, cached files are validated anyway. The additional overhead of obtaining a version stamp for each volume is low.
3. This strategy satisfies our scalability concerns. If a volume callback is broken, the client will not request another one until the next hoard walk.
4. Since hoard walks are periodic, the window of vulnerability to failures is bounded. For a client to lose the opportunity to validate files by volume, a remote update would have to be followed by a failure within one hoard walk interval (typically ten minutes). In this case, the client is no worse off than it was before the use of volume callbacks.

This policy also copes nicely with *voluntary* disconnections, when a user deliberately removes a laptop computer from the network. In our environment, many users have both desktop and laptop computers. While at work, they work from the desktop computers, leaving their laptops connected nearby. Some users modify files hoarded on their laptops from their desktop. Before disconnecting, they run a hoard walk on the laptop to fetch the files they just changed from the desktop. While connected, the laptop observes the remote updates to volumes that are referenced in its hoard database. These volumes are prime candidates for volume callbacks. A policy that becomes more conservative about obtaining volume callbacks when remote updates occur would be unlikely to obtain them in this case. In contrast, our policy takes advantage of explicit hoard walks as hints of imminent disconnection.

4.3.2 Access Rights

Directories in Coda have access lists associated with them that specify the operations that a user or group of users may perform on them. Venus caches access information to perform access checking locally. It obtains the information from the Vice status block, which is a result of most Vice calls. The access cache for a directory consists of a fixed number of entries containing a user identifier and that user's rights on the directory. Entries are considered valid when they are installed from the Vice status block. They are considered invalid (or suspect) if the object is invalidated, the user's authentication tokens expire, or if the AVSG grows.

When files are validated in groups, such as by volume, access information is not returned for the individual files. To avoid sending messages to the server to check access information, Venus must use the access cache more aggressively than it did in the past. If an object is deemed valid, clearly its access rights have not changed. Venus now considers entries in the rights cache for a file valid if the file is valid, and the entry corresponds to a user who is authenticated.

4.3.3 Effects of Replication

Coda's support of replicated volumes affects the client's handling of volume version state in two ways. First, Venus communicates with the AVSG as a group, sending the same copy of each request to each member of the group. This is performed by the underlying RPC protocol, which was designed to support remote procedure call to a set of machines in parallel. Because of this, a validation request must contain the stamps for all the servers in the VSG. Each server simply checks the one corresponding to it.

Second, Venus must collate multiple responses to its requests. When requesting version stamps, it must store the stamp for each server that responds. When validating version stamps, all servers must agree that the stamps are valid before Venus can declare them valid. Similarly, all servers must agree that a callback has been established before Venus can assume it has a callback on the volume.

5 Status and Evaluation

Servers supporting volume callbacks have been in use for several months. The corresponding Venus is currently in alpha test, and we expect to release it for production use shortly.

The primary reason for using large granularity cache coherence is to validate a client's cache quickly after a failure is repaired. In this section, we present measurements of cache validation times for five typical Coda users under a variety of conditions.

5.1 Experiment Design

The time required to validate a client's cache after a failure is the figure of merit for our experiments. We call this the *recovery time* of the cache. Obviously, recovery time depends on the contents of the cache. For the experiments, we gathered the *hoard profiles* of five Coda users, summarized in Table 1. A hoard profile is the input to a program that updates the HDB. These profiles are used primarily for laptops. To broaden our study, we deliberately chose users whose profiles were dissimilar.

We performed the experiments with a single client and server, both DECstation 5000/200s with 32 MB of memory, running Mach 2.6. The client used a 50 MB Coda file cache. The machines were connected via Ethernet. To emulate slower networks and inject failures, we used a *failure library* linked into Venus and the server. The library allows packets to be delayed or suppressed according to a *filter*, which specifies under what conditions the mischief is to occur. For example, one might request packets to a certain host be dropped with some probability, or delayed as if the network were a lower speed. Requests to insert and remove filters are issued to the failure package via RPC.

We began each experiment by initializing the hoard database with the profiles for a single user. Then we ran a hoard walk, and partitioned the client from the server. Once the client detected the failure, we healed the partition,

Volume Type	Number of Files Cached User 1	User 2	User 3	User 4	User 5
X11	38	127	133	125	142
$\TeX$			560	158	
System	9	6	190	342	689
Cboard					361
Other tools		42	13	13	42
Coda binaries			2	6	4
Coda sources			4	549	6
Kernel sources					24
User 1 personal	114				
User 2 personal		234			
User 3 personal			190		
User 4 personal				220	6
User 5 personal					537
Other personal	107	4	5	10	10
Total files	268	413	1097	1423	1821
Total volumes	7	6	9	11	12
Cache size (MB)	2.4	16.5	9.2	37.3	22.3

Table 1: Contents of Hoard Profiles for Five Coda Users, by Volume

This table characterizes the contents of the hoard profiles for the five Coda users studied in the experiments described in Section 5.1. Entries represent the number of files hoarded from each volume by each user.

The system volume contains system binaries, utilities, and include files. Cboard is a project volume for a calendar program; its maintainer is user 5. "Other tools" refers to five volumes containing utilities such as GNU-Emacs and `less`. The "Coda binaries" volume contains Coda-related programs that many users hoard. The "Coda sources" category is of interest primarily to Coda developers. It consists of two volumes containing scaffolding for the project tree, libraries, include files, and sources. User 4's personal files are split into a home volume and a volume solely for object files. "Other personal" is a set of five volumes belonging to users other than the ones we studied. Two of those volumes contain versions of `kermit` that most users hoard, and one contains a popular window manager.

caused the client to notice the server was up, and immediately ran a hoard walk. We measured the time it took for Venus to validate its cache entries, from when it noticed the server was up to the end of the hoard walk. We assume no updates on cached volumes were made to the server by any other client during the failure. Although this is the best case, we believe it is an important common case in intermittent environments.

5.2 Parameters Explored

We studied recovery times over four network speeds and three validation strategies for each user. The network speeds were 10 Mb/sec, representing Ethernet; 2 Mb/sec, representing packet radio (such as NCR WaveLan$^{\text{TM}}$); 64 Kb/sec, representing ISDN, and 9.6 Kb/sec, representing a typical dialup connection. The validation strategies were "NoOpt", "Batched", and "VCB". The NoOpt strategy validates an object by fetching its status block from the server and comparing it to the cached copy. This corresponds to the Vnode operation `GetAttr` [6]. The Batched strategy allows a group of files to be validated in one RPC. More specifically, in Coda up to 50 fids may be piggybacked with version information on a GetAttr request. The VCB strategy validates objects by volume using previously cached version stamps. These are also batched; for these experiments only 1 RPC is needed to validate the volumes.

Although the current production version of Coda uses the Batched strategy, we measured the NoOpt strategy

Network Speed	Validation Strategy	Recovery Time in Seconds User 1	User 2	User 3	User 4	User 5	Relative Times
10Mb/s	NoOpt	6.8 (.5)	9.9 (.8)	20.9 (.6)	31.5 (.5)	46.0 (1.1)	100.0%
	Batched	2.5 (.5)	3.6 (.5)	8.2 (.5)	11.0 (.0)	19.0 (.8)	38.5%
	VCB	2.5 (.5)	3.5 (.5)	7.4 (.5)	10.0 (1.3)	17.5 (.8)	35.5%
2Mb/s	NoOpt	6.5 (.5)	11.0 (2.6)	21.3 (1.2)	32.0 (.5)	46.0 (.9)	100.0%
	Batched	3.0 (.0)	4.1 (.4)	9.4 (.5)	12.6 (.5)	21.0 (.8)	42.9%
	VCB	2.3 (.5)	3.7 (.5)	7.3 (.5)	9.4 (.5)	18.1 (.8)	34.9%
64Kb/s	NoOpt	12.8 (1.4)	17.5 (.5)	40.9 (1.4)	63.6 (1.6)	87.5 (2.2)	100.0%
	Batched	5.4 (.5)	7.2 (.5)	16.9 (.4)	24.3 (.5)	36.5 (.9)	40.6%
	VCB	2.3 (.5)	4.0 (.5)	7.4 (.5)	9.6 (.5)	17.8 (.9)	18.5%
9.6Kb/s	NoOpt	67.8 (1.4)	102.8 (.9)	226.1 (2.2)	342.4 (4.0)	453.8 (9.7)	100.0%
	Batched	23.8 (2.8)	31.4 (2.5)	80.9 (15.8)	103.1 (9.7)	136.3 (8.7)	31.5%
	VCB	4.8 (.5)	5.3 (.5)	8.9 (.6)	11.3 (.5)	20.3 (.9)	4.2%

Table 2: Cache Recovery Time (Seconds)

This table presents the time in seconds needed by a client to validate cached files when it discovers a server is up. The cache contents are determined by the hoard profiles for each of the five users. The rightmost column is the average reduction in validation time compared to NoOpt for each of the other two strategies. The reduction is given as a percentage, and is calculated as $(100 \times t_{\text{Other}})/t_{\text{NoOpt}}$. These results are conservative in a number of respects, as explained in Section 5.5.

The experiments were conducted with DECstation 5000/200s as the client and server, and volumes stored at one server. Measurements were taken over an Ethernet; for the three slower speeds, an emulator was used to delay packets. Each entry is the mean and standard deviation (in parentheses) of the most consistent eight trials from a set of ten.

---

for two reasons. First, it allows our results to be compared to file systems that do not batch validations, such as AFS. Second, even though batching takes less time and bandwidth at any speed than NoOpt, it has some disadvantages at low bandwidth. Batching can result in large request packets – nearly 3KB in Coda. These requests stress the underlying RPC protocol, because retransmissions at low bandwidth can starve other requests, and cause Venus to declare servers down. Indeed, we experienced such failures while conducting our experiments! It may be more appropriate to use a smaller batching factor for low bandwidth networks. Latency is also significantly affected by request size when bandwidth is low. Currently a demand (user) request for one file will cause a batch validation of up to 50 files, which incurs additional latency that could be deferred to background processes.

Batching of volume validations does not have as great an impact on the system as batching of file validations because clients have information on many fewer volumes than files, and volume identifiers and version stamps are much smaller than their counterparts for files.

5.3 Results

Our results confirm that VCB compensates successfully for the reduction in bandwidth. Table 2 presents our observations. For all users and networks, recovery times are smallest using VCB, followed by the Batched and NoOpt strategies. There is variation across users proportional to the number of files cached. The improvement increases as bandwidth decreases. At 9.6 Kb/sec, where VCB is likely to be most important, recovery time takes only 4–7% of the time required by NoOpt, and 11–20% of the time required by batching. At higher bandwidths, the value of VCB diminishes, but it is always at least as good as the other two strategies. A glance at Table 2 reveals that the results for VCB at 9.6 Kb/sec and 10Mb/sec are not significantly different.

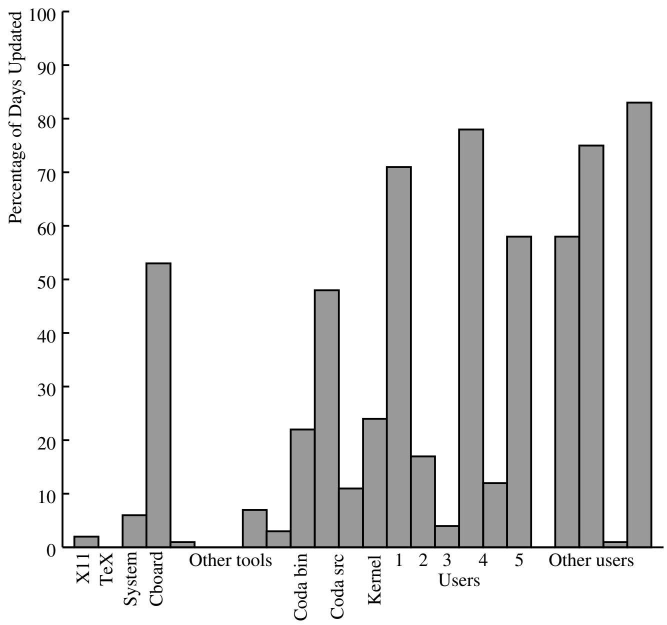

Figure 2: Daily Update Frequency of Volumes

This figure shows how often volumes used in our experiments are updated on a daily basis. The data was gathered from daily backup logs from January through March 1994. Each bar indicates the percentage of the days in the period during which at least one object in the volume was updated. We show volumes in the “Other users”, “Other tools”, and “Coda sources” categories separately, as well as both of User 4’s personal volumes.

An unexpected result was that the recovery time using VCB on a slow network was not constant over all users. We expected the bottleneck in this case would be the network. Since only one RPC was required to validate the volumes, we thought the recovery times would be similar. We observed recovery times proportional to the number of files cached, indicating the bottleneck is Venus. Most of its time is spent on two tasks: marking cached objects suspect when the server appears up, and performing the hoard walk, which involves iterating through all of the objects in the cache to ensure they are valid.

The number of callbacks at the server can be derived from Table 1, from the number of objects and volumes each user hoards. In these experiments, clients using the Batched or NoOpt strategies obtain callbacks for each file validated. Clients using VCB obtain callbacks only for the volumes they validated. The number of callbacks obtained by clients using VCB is less than 3% of the number obtained by the other two strategies.

The results presented are for the case in which all validations succeed. Over longer periods, or for more active volumes, some validations will fail because of updates at the server. As long as some validations succeed, VCB will still perform better than the other strategies. The only case for which VCB is worse is if every volume validation fails, and then it is worse by 1 RPC. Considering what users hoard, this case is unlikely.

The volumes most likely to change are the personal or project volumes of other users, as shown in Figure 2. All of the users we studied hoard files from other user volumes; however, in all but one case they represent less than 1% of the total files. Therefore validating these files individually if necessary does not have a large impact on recovery time. Further, some user volumes were inactive during the period shown in Figure 2.

The next most frequently changed set of volumes are the Coda and kernel source volumes, which are shared by up to six project members. These change relatively slowly; Figure 2 indicates that the most active of these volumes, the Coda source area, was completely unchanged for half of the days in the period we studied. Since update traffic is bursty, the results from Figure 2 are conservative, especially for intermittent environments. Thus we are confident that the benefits listed in Table 2 are realistic.

Packet Size	Time (seconds)			
	Emulated		Real	
60	.11	(.01)	.33	(.01)
260	.43	(.03)	.76	(.04)
560	.96	(.01)	1.4	(.0)
1060	1.8	(.0)	3.3	(2.6)
2060	3.5	(.0)	4.6	(.28)
3060	5.2	(.0)	6.6	(.0)
4060	7.9	(1.9)	8.7	(.0)

Table 3: Emulated vs. Real RPC at 9.6 Kbps

This table compares the round trip time for an RPC request and response of the same size, using the network emulator set to 9.6 Kbps over an Ethernet, and using a dialup SLIP link nominally rated at 9.6 Kbps. The experiments were conducted using an i386-based laptop as the client and a DECstation 5000/200 as the server. RPC packet headers are 60 bytes long; the first line gives the times for a null RPC. We show the mean and standard deviation for the most consistent eight trials from a set of ten. The large standard deviations for 4060 bytes (emulated) and 1060 bytes (real) were due to retransmissions during one or more runs.

5.4 Overhead

Of course, fast validation isn't free. There are several sources of network overhead caused by volume callbacks – obtaining callbacks, breaking callbacks, and validating volumes. Obtaining a callback on a volume requires validation of every cached file in the volume. Since this is already done by hoard walks, and the number of volumes is small compared to the number of files cached by clients, the additional overhead to obtain the volume callback is low.

In the worst case, all the volumes from which a client has cached files are being updated actively. The client then loses every volume callback it obtains, and its volume validations fail. If the sharing is false, the effort expended to get volume callbacks is wasted. Fortunately, callback requests and breaks are small messages, well under 100 bytes. Since these occur only once in every hoard walk period, the network overhead is still low. The failed volume validation costs one extra RPC. For volumes from which many files are cached, the cost of validating the files renders that RPC insignificant. If the sharing is real, the overhead due to volume callbacks is likely to be insignificant compared to the cost of re-fetching the shared data. Overall, the benefits of volume callbacks far outweigh the costs.

5.5 Accuracy of Results

The results in Table 2 understate the benefits of VCB in a number of respects. First, our failure library underestimates the delay for a given network speed. Emulation is performed by a package which intercepts outgoing packets and delays them based on the size of the packet, the network speed requested in the filter, and the delays for any packets queued ahead of the one to be sent. The delay is a simpleminded calculation, and does not take into account overheads such as UDP and IP header sizes, or IP fragmentation. A comparison of emulated and real times at 9.6Kbps is shown in Table 3.

Second, we used volumes with only one replica, when most volumes in Coda are triply replicated. Since many networks do not support multicast, an RPC request to an AVSG with more than one member is currently sent as separate messages to each member. If the network is the bottleneck, the time required to validate cached files for each of the strategies in Table 2 will be proportionately larger.

Last, caches typically contain more than what is hoarded. This occurs for several reasons – name space exploration, objects left over from other tasks, and execution of a task to find files not included by hoard profiles.

Each of these effects underestimates the savings due to VCB, especially over low bandwidth networks.

6 Conclusion

This work was motivated by the demands of mobile computing. Large granularity cache coherence is valuable in that context because it allows a high level of consistency to be preserved even when communication is intermittent or expensive. But we anticipate that this mechanism will have broader applicability. For example, we expect it to be valuable in systems such as AFS, where recent measurements indicate over 50% of requests to servers are for fetching status [12]. We conjecture that a significant fraction of these are validation requests for files that once had callbacks. These callbacks may have been lost due to failures or expiry, since AFS-3 callbacks are effectively *leases* [3].

Another argument for maintaining cache coherence at a large granularity has been put forth independently by Wang and Anderson [13]. They proposed maintaining cache coherence on clusters of files, such as subtrees. Their primary motivation is to reduce server state rather than communication.

Regardless of specific motivation, we are convinced that large granularity cache coherence is a practical and important technique for distributed computing. Our experience and measurements confirm that it is valuable in preserving the quality of file access in intermittent networking environments. Large granularity cache coherence costs little, and offers the potential for big savings.

Acknowledgements

We wish to thank members of the Coda group, in particular Maria Ebling, Puneet Kumar, Brian Noble, and David Steere, and the members of our user community, especially David Eckhardt. We also wish to thank our referees, our shepherd Mike Jones, Randy Dean, Hugo Patterson, and Mirjana Spasojevic for their helpful comments.

References

[1] R. Alonso, D. Barbara, and L. Cova. Using Stashing to Increase Node Autonomy in Distributed File Systems. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, October 1990.

[2] Rick Floyd. Short-Term File Reference Patterns in a UNIX Environment. Technical Report TR 177, Department of Computer Science, University of Rochester, March 1986.

[3] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *The Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210. ACM, December 1989.

[4] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

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

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

[7] Puneet Kumar and M. Satyanarayanan. Log-Based Directory Resolution in the Coda File System. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 202 – 213, January 1993. Also available as technical report CMU-CS-91-164, School of Computer Science, Carnegie Mellon University.

[8] L. Mummert and M. Satyanarayanan. Variable Granularity Cache Coherence. *Operating Systems Review*, 28(1), January 1994.

[9] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Knupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, December 1985.

[10] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4), April 1990.

[11] R.N. Sidebotham. Volumes: The Andrew File System Data Structuring Primitive. In *European Unix User Group Conference Proceedings*, August 1986. Also available as Tech. Rep. CMU-ITC-053, Carnegie Mellon University, Information Technology Center.

[12] Mirjana Spasojevic and M. Satyanarayanan. A Usage Profile and Evaluation of a Wide-Area Distributed File System. In *USENIX Winter Conference Proceedings*. USENIX Association, January 1994.

[13] Randolph Y. Wang and Thomas E. Anderson. xFS: A Wide Area Mass Storage File System. In *Proceedings of the Fourth Workshop on Workstation Operation Systems*, pages 71 – 78, October 1993.

Author Information

Lily Mummert received the B.S. and M.S. degrees in computer science from Texas A& M University in 1985 and 1986, respectively. She is currently a Ph.D. student at Carnegie Mellon University, working on using weak connectivity in distributed file systems. Prior to joining the Coda project, she worked on the Camelot distributed transaction facility, and on file reference tracing.

Mahadev Satyanarayanan is an Associate Professor of Computer Science at Carnegie Mellon University. He is currently investigating the connectivity and resource constraints of mobile computing in the context of the Coda File System. Prior to his work on Coda, he was a principal architect and implementor of the Andrew File System. Satyanarayanan received the PhD in Computer Science from Carnegie Mellon University in 1983, after a Bachelor's degree in Electrical Engineering and a Master's degree in Computer Science from the Indian Institute of Technology, Madras. He is a member of the ACM, IEEE, Sigma Xi, and Usenix, and has been a consultant to industry and government.