Variable Granularity Cache Coherence

L. Mummert and M. Satyanarayanan
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3891
{lily,satya}@cs.cmu.edu

Abstract

Weak connectivity is characterized by slow or intermittent networks. Distributed file systems using weak connections must function in spite of limited bandwidth and frequent connectivity changes. Callback-based cache coherence schemes were designed to minimize client-server communication, but with an underlying assumption that the network is fast and reliable (i.e., a LAN). This paper presents *large granularity callbacks* as a way to reduce the client-server communication necessary to maintain file cache coherence. Large granularity callbacks trade off precision of invalidation for speed of validation after connectivity changes.

1 Motivation

To function in a weakly connected environment, distributed file systems must cope with low bandwidth and intermittence of network connections. In such an environment, reducing client-server communication is critical for good performance. Caching reduces client-server communication considerably, however, the network must be used to maintain cache coherence.

File systems such as NFS [7] and Sprite [5] validate cached data at file open time. File systems such as AFS [3] and Coda [8] use *callbacks* to reduce client-server communication and thereby server load. When a client caches a file from a server, the server promises to notify the client if the file changes. This is called a *callback promise*, or just a *callback*. An invalidation message is called a *callback break*. If a client receives a callback break for an object, it discards its copy of the object and refetches it when it is next referenced. If a network or server failure occurs, the callback is dropped. In that case, the object is *suspect*; it may or may not still be valid. The client must validate such an object with the server when it is next referenced.

The design of AFS assumed a LAN-based environment, in which failures are rare. In this environment one would expect callbacks to be broken because of remote mutations rather than dropped because of connectivity changes. In environments with mobile computers and wireless networks, connectivity changes are frequent. This suggests the opposite – that servers will be contacted more often to validate objects that are suspect than to refetch objects that are known to be stale. In the extreme case, clients may degenerate to checking cached files on every open.

We propose to reduce the cost of validation by increasing the *granularity* of the operation. Consider the following extreme example: suppose a client could request a single callback for the entire file system. In addition to the objects it caches, it also caches a version number for the file system. The presence of a file system callback is a strong statement – it means every cached object is valid. After a partition, a single RPC with a small amount of data (a single version number) determines whether anything changed. If the version number is current, all of the client's cached data is still valid. Unfortunately, a callback break for the file system isn't very helpful – any cached object could be invalid.

This example has efficient validation but costly recovery from invalidation. Object callbacks are the other extreme – invalidation is precise, but validation requires communication for every object. The granularity

of the callback should balance the costs of validation and invalidation. Ideally, the granularity of callback would be variable. We conjecture that *large granularity callbacks* would be useful for collections owned by the primary user of a client, and collections that don't change frequently or change *en masse* (e.g., system binaries). Fine granularity callbacks would be appropriate for objects in shared areas or owned by other users. Since access patterns and network conditions are dynamic, clients should adapt to the granularity that will minimize client-server communication.

2 Detailed Design

In this section we present a design for large granularity callbacks in the Coda File System [8]. Coda is a descendant of AFS 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. A user-level process called *Venus* manages a file cache on the local disk of each client.

Below we discuss which granularities we support, how large granularity callbacks are established and broken, and how Venus handles references to cached objects. We have a prototype implementation of this design, and are awaiting usage experience.

2.1 Choice of Granularity

An obvious way to determine granularity is to exploit the hierarchical structure of the name space. However, the same practical considerations that led to the use of low-level file identifiers in AFS limit the set of granularities we can support in practice. For efficiency and scalability, the collection should have a concise, unique identifier understood by both client and server. To implement renaming in the presence of callbacks, the identifier must be unique across name bindings. These considerations preclude the use of pathname prefixes to identify the units of validation.

The level of granularity we have chosen is the *volume*, which is the next larger convenient abstraction in Coda after objects. A volume forms a partial subtree in the name space. Volumes are glued to the rest of the name space at *mount points*, which are transparent to users, and may vary in size dynamically. Typically, volumes are created for individual users or projects, so they contain collections of objects that are related. They are represented by fixed-length identifiers known to both client and server, and the identifiers are invariant across changes in mount points.

Volume callbacks (VCBs) may be maintained in addition to or instead of object callbacks. It is important to allow a VCB to be a substitute for an object callback, to amortize the cost of validation over objects cached from that volume.

2.2 Establishing VCBs

Since VCBs may be used instead of object callbacks, a pre-condition for establishing a VCB is that all cached state from the volume must be valid. Therefore, this operation can be expensive. Venus must employ an adaptive strategy, using volume callbacks only when conditions favor them. (The policy for doing this is discussed in section 3.) Coda file servers maintain version stamps on volumes, which are updated whenever an object in a volume is modified. To establish a VCB, Venus caches the version stamp for the volume.

2.3 Referencing Cached Objects

When a cached object f in volume V is referenced, a callback on f or a VCB on V constitutes proof that f is valid. If neither callback is present, and conditions favor establishing a VCB, Venus will do so at this point. If it has a version stamp for V, it checks if V has changed by presenting the server with its copy of V's version stamp. If it matches the server's version stamp for V, callback is established on V and f is

deemed valid. Otherwise, Venus must validate all of the cached objects from V. If conditions do not favor establishing a VCB, Venus will fetch f and establish a callback for it alone.

2.4 Breaking VCBs

When an object f in volume V changes, the servers send a callback break for f to all clients with callbacks on f or V. Clients interpret the callback break for f as an implicit VCB break for V. Note that if the client holds a callback on V, it will get a callback break even if f is not in its cache. This is necessary because our design allows clients to hold any combination of volume and object callbacks. After a partition a client may re-establish the callback on V (because it didn't change), but not on any of the objects cached from V. In this case, the server does not know which objects the client has cached, so it must be conservative when f changes. It is not sufficient to merely send the identifier of f on the callback break for V and allow the client to re-establish the VCB after discarding f (if cached), because other objects in V may change between the time the volume callback is broken and the time the client attempts to re-establish it.

2.5 Persistence of VCBs

We treat volume callback breaks like object callback breaks, namely, once the callback is broken, the server has no further obligation to the client for that object. An alternative is to allow the server to continue sending callback breaks for the volume, listing on each message all of the identifiers of objects that have changed. The client would receive this running commentary until it requests otherwise. This scheme has the advantage that re-establishing volume callback would be relatively inexpensive. We have rejected this approach because mutation traffic exhibits considerable temporal locality [2, 6]. If the object being changed is cached, the client should drop volume callbacks until the mutation storm passes. If the object being changed is not cached, the client should drop volume callbacks and use object callbacks instead.

3 Analysis

In this section we consider a simple calculation to obtain insight into the tradeoff between the costs of validation and invalidation. This model will be used as a basis for Venus to determine when it would be advantageous to use volume callbacks. Although we make a few simplifying assumptions, we believe the model illustrates when volume callbacks will result in less communication between client and server.

For some volume V, let

$$
\begin{array}{rcl}
f & = & \text{the number of objects in } V \quad (f \geq 1) \\
c & = & \text{the number of objects cached from } V \quad (1 \leq c \leq f) \\
p & = & \text{the rate of connectivity changes (partitions) for } V \\
m & = & \text{the rate of mutations on objects in } V \text{ from other clients} \\
P & = & \text{the probability of a mutation on an object in } V \text{ from another client during a partition}
\end{array}
$$

Assume for tractability that mutations are made uniformly over objects in V [1]. The probability of partitioned mutations depends on m and the duration of the partition. For each of volume and object callbacks, we count the number of RPCs required to keep the cache and server copies consistent over some period of time t. The analysis is worst-case, i.e., after a mutation the mutated object is always referenced, and after partitions the entire cache is referenced. This is not too severe a departure from reality, since Coda's *hoard* subsystem periodically references all cached objects in its database [4].

[1] Given that mutations are localized, m is not uniform over objects in V. This change affects only the analysis in section 3.1 since the probability that the modified object was cached is no longer $\frac{c}{f}$.

3.1 Object Callbacks

When a cached file is modified at another client, the server makes one RPC to break the callback, and the client will make one RPC next time the file is referenced. After a partition, the client must contact the server on the first reference to each cached object to ensure that the cached copy is still valid. The number of RPCs over time t is then

$$\begin{aligned} RPCs(t) &= \text{callbacks on objects in the cache} + \\ &\quad \text{references after callbacks} + \text{references after partitions} \\ &= mt\left(\frac{c}{f}\right) + mt\left(\frac{c}{f}\right) + ptc \\ &= 2mt\left(\frac{c}{f}\right) + ptc \end{aligned}$$

3.2 Volume Callbacks

Suppose a client has volume callbacks only. If an object in the volume is modified remotely, the client receives a callback break for that volume. To re-establish the VCB, the client must validate all cached state from that volume. After a partition, the client sends the server its copy of V's volume version stamp. If nothing has changed, then the cached copies of everything in V are valid. Otherwise, the client must recover from the partition as in the object callback case, by validating each cached object on the next reference. The number of RPCs over time t in this case is

$$\begin{aligned} RPCs(t) &= \text{callbacks on objects in the volume} + \text{validating state after callback} + \\ &\quad \text{volume check after partitions} + \text{references after partitions} \\ &= mt + mtc + pt + ptc \cdot P \end{aligned}$$

Volume callbacks are advantageous when

$$2mt\left(\frac{c}{f}\right) + ptc > mt + mtc + pt + ptc \cdot P$$

$$p(c - c \cdot P - 1) > m\left(1 + c - 2\left(\frac{c}{f}\right)\right)$$

$$\frac{p}{m} > \frac{1 + c - 2(\frac{c}{f})}{c - c \cdot P - 1}$$

This inequality yields a few observations. First, p must be strictly greater than m to offset the cost of recovering from a broken volume callback. This is the most important factor. Second, the more objects cached from a volume, the lower the probability of false invalidation.

Now let us examine a few specific cases. First consider a volume owned by the primary user of a wireless mobile computer. In this case, we expect $p \gg m$, and since m is nearly zero, $P = 0$. In this case as long as the client has cached more than one file, volume callbacks will be a win. Second, consider the volume of the primary user on a fully connected workstation. In this case, we expect p to be low, but

non-zero because of server failures. If the user accesses his volume from only this client, we expect m to be nearly zero. The amount of savings from volume callbacks will depend on the number of files cached from the volume – the more, the better. Third, consider a system source volume. In this volume, m is non-zero because it is shared by a group of developers who occasionally install new files. If accessed from a fully connected client, p and m are likely to be close. Volume callbacks will be helpful only if a large percentage of the volume's files are cached.

3.3 Implementation Considerations

Most of the logic for volume callbacks is on the client. Venus must collect enough information on the parameters above to determine when it should use volume callbacks. Venus can easily keep counts of c and p. To estimate m, Venus can count the number of callbacks received per volume. The count may be an underestimate if Venus has held only object callbacks, since it is not notified of changes elsewhere in the volume. To establish volume callbacks Venus can require $p > km$, where k is supplied at start time, or set with a reasonable default. Venus can obtain f from the server, or it can require a minimum on c. To estimate the probability of partitioned mutations, Venus can simply count the number of times a volume validation failed.

On the server, an additional callback table is needed for hosts holding volume callbacks. Since there are many fewer volumes than objects, this is at most a small amount of additional state.

4 Related Work

For reasons quite different from ours, Wang and Anderson have recognized independently the value of maintaining cache coherence at a large granularity [9]. They propose maintaining coherence on clusters of files, such as subtrees. Their primary motivation is to reduce server state, rather than client-server communication. There is no discussion in their paper of how the choice of granularity is made. To the best of our knowledge no working implementation of this proposal exists yet [1].

5 Conclusions

In this paper we have introduced large granularity callbacks as a way to reduce the client-server communication necessary for cache coherence. We believe they will be beneficial in any environment in which validations are frequent or expensive, and can be used in combination with other techniques such as batching to improve performance. We have presented a design for volume callbacks in the Coda File System, and analyzed factors affecting their performance. In our design, the client determines based on these factors whether object or volume callbacks would be best for a particular volume, and adapts to the appropriate level dynamically. We have implemented volume callbacks for the Coda File System, and expect to gain direct experience with them in the coming months.

References

[1] Thomas E. Anderson. Personal communication, October 1993.

[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] 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.

[4] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25. Association for Computing Machinery SIGOPS, October 1991.

[5] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134 – 154, February 1988.

[6] 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.

[7] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network File System. In *USENIX Summer Conference Proceedings*. USENIX Association, June 1985.

[8] 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.

[9] 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.