Coping with Conflicts in an Optimistically Replicated File System

Puneet Kumar
School of Computer Science
Carnegie Mellon University

1. Introduction

Coda is a scalable distributed Unix[1] file system that provides high availability through the use of two distinct but complementary mechanisms. One mechanism, *server replication,* stores copies of a file at multiple servers. The other mechanism, *disconnected operation,* is a mode of execution in which a caching site temporarily assumes the role of a replication site. Disconnected operation is particularly useful for supporting portable workstations.

Both mechanisms are forms of *optimistic replication* [1], meaning that they place availability above consistency. Consequently, Coda allows potentially conflicting updates to be made in each network partition during failures. We believe this strategy is acceptable for two reasons. First, sequential write-sharing of data within a short period of time is relatively infrequent in Unix. Second, Coda provides mechanisms to minimize the inconvenience caused by conflicting updates.

This paper focuses on the mechanisms in Coda to cope with conflicts and describes our experience in implementing them. Other aspects of Coda are described in earlier papers [3, 4].

2. Support for Optimistic Replication

In a system that adopts an optimistic replication strategy there must be support to cope with conflicts when they arise. Specifically the system should:

- be able to *detect* a conflict as soon as possible after it arises,
- *contain* the damage due to a conflict, and *preserve* information needed to fix it,
- provide means for *recovering* from the conflict.

In Coda, the *resolution subsystem* provides this support. The following sections describe how Coda satisfies each of these requirements.

3. Detecting Conflicting Updates

Every update operation in Coda is tagged with a unique identifier called the *store id*. Stored along with each replica is the store id of its latest update operation, called the *latest store id (LSID)*. Different LSIDs identify potentially conflicting updates. We use another mechanism *Coda version vectors(CVV)*, to distinguish between genuine conflicts and mere staleness of a replica. CVVs are conceptually similar to version vectors first suggested by Locus [6], but differ in detail. A CVV is an estimate of the length of the update histories of each replica. CVV comparison may indicate a conflict when none exists, but it will never falsely indicate dominance or equality when there is a conflict. Details of how CVVs work are described elsewhere [4].

Although CVVs are useful for differentiating between version staleness and conflicting updates in plain files, they are not useful for directories. This is because a directory's CVV does not capture the update activity of its children. If directory CVV's were to reflect updates of their children, then any modification would require CVV's of all objects along the path from the root to that directory to be updated. Furthermore, directories are amenable to automatic resolution as described below, so version information beyond equality/inequality is superfluous.

[1] Unix is a trademark of AT&T

4. Containing Damage and Preserving Evidence

Damage containment is achieved in Coda by disallowing further mutations on conflicting replicas. At the servers the replicas are "marked in conflict". At the client an obvious way to implement this would be for system calls on the object to return an "Inconsistent Object" error code. Unfortunately, this would require modifications to existing Unix applications to provide intelligent error messages. Instead, the cache manager, called Venus, makes the object appear to be a dangling symbolic link. The contents of the symbolic link are derived by the system from the low level identifiers for the object.

If servers were the only repositories of objects, the number of conflicting replicas would always be limited by the replication factor. But, Coda provides the means for clients to temporarily assume the role of replication sites (disconnected operation), so the number of potentially conflicting replicas for an object is effectively unbounded. To preserve all evidence of the conflicting updates Coda uses a temporary repository per *volume*[2] called a *covolume*. Covolumes are like *lost+found* directories in Unix except that they are not accessible to normal Unix applications. The names of the objects in the covolume are derived from the low level identifiers used by the system. The only way to access objects in the covolume is via a special *repair* tool (q.v.).

5. Resolving Conflicts

Resolving a conflict requires knowledge of the contents of objects. Since files are untyped byte streams there is, in general, no information to automate their resolution. Coda provides a *repair* tool to aid the resolution process. Directory contents, on the other hand, have well defined semantics and their resolution can sometimes be automated, an observation first made by Locus [6]. For example, partitioned creation of files in different replicas of the same directory will make it inconsistent. When the partition heals, the system can automatically resolve the inconsistency by inserting the missing files in the directory replicas that do not already contain it. In the following subsections we describe the repair tool and automated resolution for directories. The repair tool has been implemented and is a part of the Coda prototype. Automated directory resolution is being implemented.

[2] A volume in Coda is like a logical file system in Unix that can be mounted in different areas of the tree.

5.1. Manual Repair Tool

The Coda repair tool allows users to manually resolve inconsistent objects. It uses a special interface to Venus, so that file requests from the tool are distinguishable from normal Unix application file requests. This allows the tool to mutate inconsistent objects (subject to normal access restrictions). The tool allows users to mount volumes with inconsistencies in a scratch area in read-only mode. The volumes are mounted in read-only mode so that the user cannot inadvertantly alter anything. The read-only replicas are themselves not inconsistent so the user can examine their contents using normal Unix applications and determine the cause of the inconsistencies.

To repair an inconsistent file the user specifies a local file whose contents are to be forced over the replicas at all sites. The local file is typically a brand new file created by the user or a manually merged version of the conflicting replicas. Venus sends the file to the servers and in one atomic action the servers force the new data and clear the inconsistency. From then on the object is accessible to all normal Unix applications.

To repair a directory the user specifies a list of compensating operations for each replica. The operations are listed in a local file which is sent to the servers during the repair. Each server parses the file and performs the compensating operations on its replica. The inconsistency is not cleared. The user must verify that the versions are indeed identical.

At the server, the set of compensating operations are performed via a transaction. Coda servers store directory contents and file meta information in *recoverable virtual memory (RVM)* [5]. With the aid of transactions and RVM the servers can guarantee the atomicity of the compensating operations. This considerably simplifies the user's management of manual repair in the event of server or network failures.

5.2. Automating Directory Resolution

In the taxonomy put forth by Guy [2], all but three kinds of conflicting directory operations can be automatically resolved in a Unix system. These are *remove/update* conflicts (involving an update of a replica in one partition and removal in another), *update/update* conflicts (exemplified by status modifications to partitioned replicas of a directory), and *name/name* conflicts (objects with identical names are created in partitioned replicas of a directory). Name/name and update/update conflicts can be detected by inspecting the contents of directory replicas. To detect remove/update conflicts, the state of the replica just before removal should be known.

Our original implementation strategy consisted of creating a *ghost* when an object is deleted during partitioning. The ghost contains the status of the object just before deletion. A ghost is invisible to the user but is accessible to the resolution subsystem via the object's parent. Except for directories, ghosts contain no data. Directory ghosts may contain a list of child ghosts. Thus, recursive tree removals during partitions may result in trees of ghosts. The implementation of resolution with ghosts is complicated. Directory contents have to be examined to recreate the update histories of replicas and compose the list of compensating actions. A garbage collection mechanism is needed to reclaim ghost objects when network partitions disappear. The advantage of ghosts is that they can exist indefinitely, so remove/update conflicts will be detected eventually. However, during long partitions a lot of garbage will be generated in the system which could paralyze it.

We are implementing a new design for the resolution subsystem. Instead of creating ghosts, the status of an object being deleted is stored in its parent's *log*. The directory data structure is augmented to accomodate a circular log which contains the list of operations performed on the directory and the status of any affected child. It is invisible to the user. A directory log is a linked list allocated from a finite circular buffer in the volume. Each log record contains the id of the operation and the id(s) and some status bits of the affected child(ren). When the log fills up, old records are reused. Directory operations are logged only during partitions. Logging simplifies the implementation of directory resolution. Resolving two directories now entails finding the last common log entry and replaying the rest of the log at the other site. *Genuine* conflicts are automatically detected when some operations cannot be performed at a site. The finiteness of the log ensures that old log entries are automatically garbage collected by the system. The disadvantage is that the last common log entry between two replicas may be overwritten due to a wrap around. In this case, even if there is no genuine conflict, manual repair is required.

We believe that logging directory operations is a better design. In the absence of failures there is no overhead. In the event of short network partitions there is a small overhead of logging when objects are created or deleted. Resolution can be automated during long partitions provided there is no write-sharing (even when some useful logging information is overwritten due to a buffer wrap around). One replica will be strictly dominant and its contents can be forced at all other replication sites. This situation can be detected by storing the last log entry before the failure separate from the log. Resorting to manual repair in the event of write sharing during long network partitions is an acceptable compromise.

We have augmented the repair tool as a first step to automating directory resolution. Given an inconsistent directory the tool compares its replicas and produces a list of compensating operations, that will make them identical. When it encounters a genuine conflict it prompts the user for help. Completely automated resolution is similar except that it is invoked automatically whenever more than one version of a directory is seen by the client. All genuine conflicts will be marked with a special flag indicating the type of conflict.

5.3. Automating Resolution of Genuine Conflicts

No matter how convenient the repair tool is, manual repairs are an annoyance to the user and their frequency should be minimized. We are exploring the feasibility of attaching *resolution policies* to user volumes. Policies are "heuristics" which specify what action to take for genuine conflicts. For example, there may be a policy for remove/update conflicts specifying that the remove should be backed out and the update forced. Policy rules could be specified in a variety of ways (e.g., like Unix "make" rules). We are still exploring this and will be able to report more by the time of the workshop.

6. Conclusions

Conflict detection, confinement of damage and resolution are indispensable functions of a distributed file system that uses optimistic replication to provide availability. Version vectors are useful for detecting conflicts in files but not for directories. Since directories are usually small and have a fixed set of operations, it is possible to log directory operations thus simplifying the resolution algorithms considerably. A finite length log is sufficient for automated resolution except when conflicting updates are made during lengthy partitions. The added complexity of managing directory logs is minimal in a system that provides transaction and recoverable virtual memory support at the servers.

References

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

[2] Guy, R. A Replicated Filesystem Design for a Distributed Unix System. Master's thesis, Department of Computer Science, University of California, Los Angeles, 1987.

[3] Satyanarayanan, M. Scaleable, Secure, and Highly Available Distributed File Access. *Computer 23*, 5 (May 1990).

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

[5] Spector, A., and Swedlow, K.R., e. *The Guide to the Camelot Distributed Transaction Facility: Release 1*, 0.98(51) ed. Carnegie Mellon University, May 1988.

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