On the Ubiquity of Logging in Distributed File Systems

M. Satyanarayanan
James J. Kistler
Puneet Kumar
Hank Mashburn

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Logging is a simple yet surprisingly versatile and powerful implementation technique. Since its invention and popularization as a recovery technique for transactional databases [3], it has been used for a variety of other purposes such as updating meta-data in file systems [4], exploiting write-once media [2, 12], maintaining authentication audit trails [10], and increasing the write bandwidth of disks [9].

In this position paper we describe how logging in different guises is used in the Coda File System [5, 11]. Coda is a distributed file system whose goals are to provide highly available, scalable, secure and efficient shared file access in an environment of Unix workstations. High availability is achieved through two complementary mechanisms, *server replication* and *disconnected operation*.

Our experience with logging in Coda is especially valuable because we did not start out with any preconceived notions about the aptness of logging for our system. In fact, we did not think of using logging for any aspect of Coda until well into its detailed design. That logging has proved so useful is hence compelling evidence of its value as an implementation technique.

In the light of our experience, we now hold the position that *logging should be at the forefront of techniques considered by a system designer when implementing a distributed file system.*

Logging is used in at least three distinct ways in the current implementation of Coda. First, value logging forms the basis of the recovery technique for *RVM,* a transactional virtual memory package. Second, operation logging is used in the *replay log* that records update activity made by a client while disconnected from all servers. Third, operation logging is used in *resolution logs* on servers to allow transparent resolution of directory updates made to partitioned server replicas.

1. RVM

RVM, an acronym for *recoverable virtual memory,* is a Unix library that supports local, non-nested transactions on data structures mapped into a process' virtual memory [7]. A unique aspect of RVM is that it allows independent control over the basic transactional properties of atomicity, permanence, and serializability.

Atomicity and permanence of transactions are obtained using a NO-UNDO/REDO value log. The log can be a raw disk partition or a Unix file. If a Unix file is used, true transactional guarantees are achieved only if the `fsync` system call blocks until dirty file buffer cache data has been written to disk. There are two log operations: `flush` and `truncate.` As transactions are committed, new-value records of virtual memory modifications are written to the log device in the `flush` operation. Periodically, the modifications represented by the log records are applied to the committed data image via the `truncate` operation.

Flushing and truncation are usually transparent to applications. But because log management is the strongest determinant of RVM performance, these operations are made visible and an application can use knowledge of its internals to optimize their timing. For example, an application can reduce commit latency by labelling the commit as *no-flush,* thereby avoiding a synchronous write to disk at the expense of persistence. To ensure persistence of no-flush transactions, the application must explicitly flush RVM's write-ahead log from time to time. When used in this manner, RVM provides *bounded persistence,* where the bound is the period between log flushes.

RVM's design is minimalistic. Much of the motivation for building RVM has come from experience with Camelot [1], a general-purpose transaction system used in an early implementation of Coda. While Camelot is more general than RVM, we found that we were deriving most of the benefit of transactions using only a fraction of Camelot's facilities. RVM may thus be viewed as an exercise in discovering the minimum transaction processing functionality useful in building non-database applications such as Unix file servers.

2. The Replay Log

Coda supports disconnected operation at client workstations when no server replica of a volume is accessible. While disconnected, the cache manager, *Venus,* must record sufficient information to replay update activity upon reconnection. It maintains this information in a per-volume operation of mutating operations called a *replay log.* Each log entry contains a copy of the corresponding system call arguments as well as the version state of all objects referenced by the call. Since the replay log is maintained in RVM, we have a situation where an operational logging layer is built upon a value logging layer.

Besides replay logs, Venus maintains other meta-data such as cached directories, symbolic link contents, and status information for cached objects of all types in RVM. The actual contents of cached files are not in RVM, but are stored as local Unix files.

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

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

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

3. The Resolution Log

Coda also supports optimistic replication of data at servers. A key problem in optimistic replication is detecting when replicas of an object have been updated simultaneously in multiple partitions and using the object's semantics to merge the concurrent partitioned updates. We refer to the process of examining replicas of a directory, deducing the set of partitioned updates and merging them using Unix semantics as *directory resolution.* Concurrent partitioned updates that violate Unix semantics when merged are called conflicting updates.

The basic data structure used for directory resolution is a *resolution log* associated with each replica. Each resolution log entry contains the arguments of the directory operation it represents. Atomicity of the log and directory mutation is guaranteed by by placing both log and directory contents in RVM and mutating them within the same transaction.

This is another instance of operational logging being layered on value logging. There is one physical resolution log associated with each Coda volume. However, log entries for each directory are linked together for faster lookup since each directory is resolved separately.

By examining the logs from a set of replicas, it is possible to unambiguously determine the updates missed by each replica, and to detect conflicting updates. A four-phase protocol, with one server acting as coordinator, is used to perform directory resolution. In the first phase all the replicas participating in the resolution are locked. In the second phase, the coordinator collects the resolution logs of the other replicas. In the third phase, a merged log is shipped to the replicas, and each computes and replays missed updates. In the final phase, all the replicas are unlocked.

A log entry's space can be reclaimed as soon as the corresponding directory update has been propagated to all other replicas. Such information is usually made available to a server during the second phase of the update protocol (Coda Optimistic Protocol [11]). Therefore, in the absence of failures a log entry is reclaimed shortly after it is allocated. However, during partitions a log on a server can become full. In that case the server can unilaterally reclaim old log entries. Correctness is not violated by this, although the ability to transparently resolve non-conflicting updates is lost.

Using a log-based strategy for directory resolution provides an easily-understood tradeoff between resource usage and usability: the larger a log, the lower the likelihood of loss of transparency. Further, it is substantially easier to implement, and is more flexible and efficient than the purely inferential scheme used in the Ficus file system [8].

Measurements from the implementation show that the time for resolution is typically within 10% of the time for performing the original set of partitioned updates. Analysis based on file traces from our environment indicate that a log size of 2 MB per hour of partition should be ample for typical servers [6].

4. Conclusion

In the near future, we see at least one additional use of logging in Coda. This is a generalized logging facility for "conflict-smart" applications to perform transparent file resolution. We plan to implement such a logging facility on top of RVM. We expect that applications will use the facility for operational logging, though nothing precludes its use for value logging.

Overall, our strategy of using a general-purpose value logging layer at the bottom, with special-purpose operation logging above has worked out very well. It combines the well-known strengths of these two forms of logging in a highly convenient and usable manner.

References

[1] Eppinger, J.L., Mummert, L.B., Spector, A.Z.
Camelot and Avalon.
Morgan Kaufmann, 1991.

[2] Finlayson, R., Cheriton, D.
Log Files: An Extended File Service Exploiting Write-Once Storage.
In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. November, 1987.

[3] Gray, J.
Notes on Data Base Operating Systems.
Technical Report RJ2188, IBM Research Laboratory, San Jose,, February, 1978.

[4] Hagmann, R.
Reimplementing the Cedar File System Using Logging and Group Commit.
In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. November, 1987.

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

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

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

[8] Popek, G.J., Guy, R.G., Page, T.W., Heidemann, J.S.
Replication in Ficus Distributed File Systems.
In *Proceedings of the IEEE Workshop on Management of Replicated Data*. November, 1990.

[9] Rosenblum, M., Ousterhout, J.K.
The Design and Implementation of a Log-Structured File System.
ACM Transactions on Computer Systems 10(1), February, 1992.

[10] Satyanarayanan, M.
Integrating Security in a Large Distributed System.
ACM Transactions on Computer Systems 7(3), August, 1989.

[11] 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.

[12] Svobodova, L.
A Reliable Object-Oriented Repository for a Distributed Computer System.
In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*. December, 1981.

Table of Contents