Transparent Logging as a Technique for Debugging Complex Distributed Systems

M. Satyanarayanan
David C. Steere
Masashi Kudo
Hank Mashburn

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

April 17, 1992

As any battle-scarred veteran will testify, debugging a distributed system in production use is an enterprise fraught with great difficulty and frustration. By the time the system is released for production use, most of the easy bugs have been found and fixed. The remaining bugs are typically non-deterministic in nature, and will only manifest themselves under conditions of heavy use. Although rare, such bugs cannot be ignored because they often have serious consequences.

In this position paper, we put forth the thesis that *logging is a flexible, powerful, and convenient tool for debugging complex distributed systems.* We substantiate this thesis in three steps. First, we argue that logging is particularly well suited for debugging distributed systems. Next, we observe that logging is already used in distributed systems for reasons independent of debugging. Finally, we show that the latter uses of logging can be transparently extended to support debugging.

1. Debugging Complex Distributed Systems

Much of what we have learned about debugging distributed systems is drawn from our experience in building the Andrew and Coda File Systems and supporting significant user communities on them[6]. The evolution of such systems can be described by the following cycle. At each stage, bugs are found and fixed making the system more stable. This attracts more users, resulting in heavier usage. This in turn triggers more subtle bugs, which are typically caused by increasingly pathological interactions between components of the system. Attempts to reproduce the bugs offline are rarely successful, since it is extremely difficult to precisely re-create the sequences of events inducing the bugs. Ignoring such bugs (for example, by restarting a server) may not be acceptable, since corruption of non-volatile storage may be involved.

How does one find and fix such elusive bugs? Traditional debugging techniques, such as the use of debuggers, adding debugging statements, or code walk-throughs are usually ineffective. First, debuggers or the addition of output statements can affect the timing of a system, causing race conditions to change or disappear. Second, modifying the code causes relinking, which can change the layout of data structures in memory. This may remove symptoms of a memory corruption bug, but the bug may manifest itself in the future in other ways. Third, given the size and complexity of most distributed systems, reading the code is seldom insightful enough to locate the bug. Finally, the debugging cycle of recompiling, relinking, and regression testing becomes increasingly onerous as the size of the system and users' reliance on it increases. Hence the ability to rapidly generate and test debugging hypotheses is lost.

Logging provides exactly the right functionality for debugging these types of problems. It can correctly capture the precise sequence of events preceding a failure. Since the corruption of non-volatile data may not be detected until long after it occurs, the history stored in the log is indispensable in finding these bugs. Another benefit of logging is that it is versatile. One can store a variety of data, from byte-level modifications to invocations of important operations. Logs can be cached in VM, using write-through or write-back strategies, depending on the need for reliability.

2. Logging as a Paradigm for Building Distributed Systems

Independent of its value for debugging, logging is valuable in building distributed systems[7]. For example, logging provides two benefits in Coda: it simplifies recovery code, and it provides a history of high-level operations for reconciling modifications made during network partitions.

Recovery in Coda is simplified by the use of *recoverable virtual memory (RVM)*[3]. RVM is a lightweight transaction facility which supports local non-nested transactions on non-volatile data structures that have been mapped into a server's address space. RVM simplifies the design of Coda because it encapsulates all concerns pertaining to recovery. Maintaining internal invariants only requires that transactions take data structures from one consistent state to another in the absence of failures.

Coda uses logging in two ways for coping with network partitions. Clients keep track of modifications made while disconnected from servers in their *replay logs*. These logs are then used to propagate updates to the servers upon reconnection. Servers keep track of directory modifications made during network partitions in *resolution logs*. Resolution logs allow servers to automatically resolve conflicting updates using knowledge of Unix semantics.

The use of logging in systems predates Coda. For example, logging was originally used as a recovery technique for transactional databases[1]. Since then it has been adapted for a wide variety of uses, such as maintaining authentication audit trails[5], updating meta-data in file systems[2], and increasing the write-bandwidth of disks[4].

3. Transparently Exploiting Logging for Debugging

A key insight that we have gained through our work with Andrew and Coda is that the logging needed for debugging can be transparently incorporated into the system. As a result, logging for debugging can be obtained *without modifying server code and with minimal perturbation of system behaviour*.

As Coda evolved, we uncovered bugs that were taking many weeks to locate. We realized that the information in the RVM log was exactly what we needed to locate the source of corruption to Coda's non-volatile store. To determine the precise sequence of modifications to non-volatile data structures we only had to develop a tool for analyzing and displaying the log data. This technique has enabled us to find several memory corruption bugs with minimal effort. We plan to extend RVM to tag transactions in a manner that will allow us to more quickly identify the code responsible for a specific modification.

Another form of logging occurs in the RPC stub code. An extension to our RPC package generates stub code to trace RPC activity and to store the data in a circular log in VM. This tracing function can be turned on or off dynamically. To gain permanence, data from the VM log can spooled and asynchronously written to disk. Thus this technique captures recent RPC history with no modifications to the server code.

4. Conclusion

In closing, we would like to reemphasize our position that logging is a powerful and versatile tool for debugging mature distributed systems. The problems that arise in these systems are often non-deterministic and difficult to reproduce. Logging proves to be a valuable technique for finding such problems. Our experience is that such logging can be incorporated with little or no programmer-visible modifications, and with minimal perturbation of system behaviour.

References

[1] J. Gray. Notes on data base operating systems. Technical Report RJ2188, IBM Research Laboratory, San Jose, February 1978.

[2] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November 1987.

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

[4] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[5] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3), August 1989.

[6] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5), May 1990.

[7] M. Satyanarayanan, J. Kistler, P. Kumar, and M. Mashburn. On the ubiquity of logging in distributed file systems. In *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992.