Sprite Position Statement:
Use Distributed State for Failure Recovery

Brent Welch
Mary Baker
Fred Douglis
John Hartman
Mendel Rosenblum
John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
spriters@ginger.Berkeley.EDU

Introduction

‘‘Stateless’’ servers have been popularized by NFS [Sandberg85]. The benefit of a stateless server is that the server can crash and reboot and no special recovery action is required. Clients simply retry their operations until they get a response from the server. There are two draw-backs to this approach, however. First, clients can get stuck if they try to contact a server that stays down for a long time. Second, and much more important, a stateless server suffers a performance hit because all changes in the service’s internal state† must be saved on stable storage.

We advocate keeping state in main-memory instead of logging state to disk so that we can implement higher performance services. We are motivated by our distributed file system that uses stateful servers to support a high performance distributed caching system[Nelson88]. For reliability a server’s state is replicated in the main-memory of other hosts so that the system can recover from failure of a server. After a server reboots its clients help it rebuild its internal state. As networks and processors get faster, but disks do not, relying on other hosts will be more efficient than using disks.

Distributed File System Recovery

For the Sprite distributed file system we designed and implemented a recovery system based on replicated state. A file server maintains state that supports distributed caching of file data. Our caching system makes the file system approximately 30% faster than NFS. (Speed-up is for the Andrew file system benchmark [Howard88]. Your mileage

† ‘‘Stateless’’ is not a good term for two reasons. One, the servers do keep state, but on their disk instead of in main-memory. Secondly, neither ‘‘stateless’’ nor ‘‘stateful’’ are in the dictionary!

may vary with actual use.) The goal of our recovery system is to recover the file server's state after it reboots so the currently open (and cached) files on other workstations can continue to be used.

The state that has to be recovered after a file server reboots is the current open file state. The file server keeps this state on a per-client basis and uses it to keep remote caches consistent. It would be complex and costly to log this state to the server's disk. Instead, it is relatively straight-forward to have the operating system kernel on each workstation keep track of how its processes are using files. This means that the open file state is replicated on the server and its clients as shown in Figure 1.

The format of the state is given in Table 1. Note that the state includes a count of active opens, while transient references to a file are not reflected in the state. For example, fork and dup operations increment a local reference count, not the open state, so these operations remain cheap.

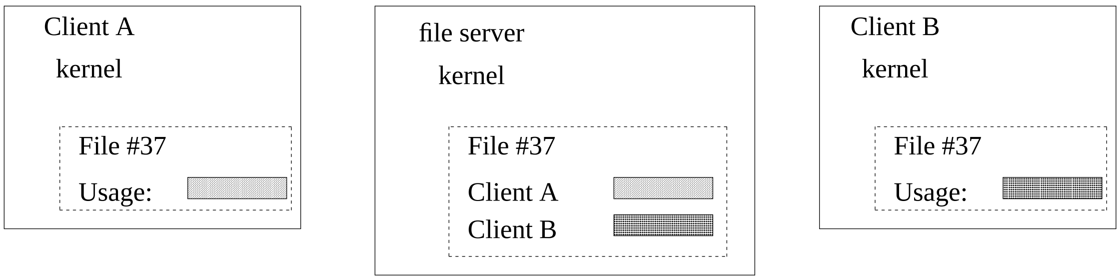

Figure 1. Duplicated state is used to enable recovery after server failure. The server can crash and its file usage state can be rebuilt from the state duplicated on the client hosts.

Per-Client File State	
ClientID	Workstation identifier.
OpenCount	Number of active open() calls.
WriteCount	Number of opens for writing.
ExecCount	Number of opens for execution.
LastWriter	TRUE if client has dirty file data.
LockBit	TRUE if exclusive lock is held.
LockCount	Count of shared locks.

Table 1. State maintained on a per-client basis for each file. The server keeps some additional state about each file on stable storage that includes a version number, ownership, access control information, and the file data itself.

A server reconstructs its open file state through a *re-open* protocol. Workstations contact the server and re-open their files by presenting the server with their part of the open file state. The re-open protocol is an idempotent protocol that attempts to reconcile the server's file usage state with the client's state. A client workstation initiates the re-open protocol any time it determines that the server's state might be out-of-date. Typically this happens after a server reboot, or after a network partition ends.

If a client's state for some file is not compatible with the server's existing state, then the client's re-open of that file is denied. An incompatibility can arise after a network partition that isolates an active writer from the server. The server might conclude that the original writer died and allow another writer to generate a conflicting version of the file. The file servers use their regular cache consistency algorithm to detect this conflict.

Our recovery system provides the same behavior that proponents of stateless servers desire. Applications are blocked if they access a server that is down, and they continue automatically after it recovers. However, our recovery system also allows these operations to be aborted at the user's request. This is possible because recovery is handled at a high level by the file system, instead of at a low-level in the communication protocol.

Recovery Costs

There are two contributions to the cost of a state-based recovery system: crash detection, and the state itself. The Sprite kernel includes a simple module that tracks the state of other hosts and triggers actions when other hosts crash or reboot. A distinction between crashing and rebooting is made because services need to clean up after clients crash, while clients need to take recovery action after servers reboot. Crash and reboot detection is done by monitoring the kernel-to-kernel network RPC protocol. Timeouts indicate a host failure, and hosts are checked periodically to make sure they are up. An explicit check is avoided if there has been recent message traffic from a host. A generation number in the RPC header is used to detect reboots, and this requires monitoring arriving messages. Currently, only a small percentage of each CPU is consumed by host monitoring, although this does increase as the size of the system increases.

One thing we did not anticipate was the large amount of state maintained by our file system. In a small system of about 15 clients and 2 servers, each server has about a thousand memory-resident file descriptors, and each client may have a few hundred. The large number is due to caching. If a file is cached anywhere, then the server has to keep a descriptor for it. The number of open files is not limited, so the memory usage of the file servers will continue to grow as the system gets larger. The large number of descriptors also affects the server's recovery time. Currently the clients will initiate recovery during a 30 second period, and each client takes 1 to 4 seconds to recover. The client recovery time depends on overlap with other clients, and any background processing that the server may still be doing as part of bootstrap.

Conclusion

Our experiences with recovery in a distributed system indicate that solutions based on replicated state are quite feasible. Whether or not this sort of recovery is more efficient than a disk-based approach depends on the relative costs. However, as the performance of hosts and networks accelerates relative to disk performance, using other hosts for state backup will be more and more efficient than using local storage. In the Sprite file system, for example, it was not expensive for the clients to maintain a copy of the server's open file state. Clients already kept some state associated with local access. Replicating the server's state was achieved by making the client's bookkeeping match the server's. We found this simpler than logging the server's open file state to its disk, which would also slow down open and close operations.

Research in this area includes further study of our existing system, and application of the recovery principal to other areas of the system. We need to measure the per-client cost of system so we can estimate the scalability of our approach. There is both a memory-usage cost to the server, as well as a background CPU load involved with host monitoring. We would also like to extend our system so that applications can benefit from the host-monitoring sub-system of the kernel. With this in place we can experiment with different recovery systems in user-level applications.

References

Howard88. J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham and M. J. West, ''Scale and Performance in a Distributed File System'', *Trans. Computer Systems 6*, 1 (Feb. 1988), 51-81.

Nelson88. M. Nelson, B. Welch and J. Ousterhout, ''Caching in the Sprite Network File System'', *Trans. Computer Systems 6*, 1 (Feb. 1988), 134-154.

Ousterhout88. J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, ''The Sprite Network Operating System'', *IEEE Computer 21*, 2 (Feb. 1988), 23-36.

Sandberg85. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, ''Design and Implementation of the Sun Network Filesystem'', *USENIX Conference Proceedings*, June 1985, 119-130.