Availability in the Sprite Distributed File System

Mary Baker
John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
415-642-9669
mgbaker@sprite.Berkeley.EDU
ouster@sprite.Berkeley.EDU

ABSTRACT

In the Sprite environment, tolerating faults means recovering from them quickly. Our position is that performance and availability are the desired features of the typical locally-distributed office/engineering environment, and that very fast server recovery is the most cost-effective way of providing such availability. Mechanisms used for reliability can be inappropriate in systems with the primary goal of performance, and some availability-oriented methods using replicated hardware or processes cost too much for these systems. In contrast, availability via fast recovery need not slow down a system, and our experience in Sprite shows that in some cases the same techniques that provide high performance also provide fast recovery. In our first attempt to reduce file server recovery times to less than 90 seconds, we take advantage of the distributed state already present in our file system, and a high-performance log-structured file system currently under implementation. As a long-term goal, we hope to reduce recovery to 10 seconds or less.

This paper appeared in Operating Systems Review, April 1991.

1. Introduction

The degree of fault-tolerance in a system is related to the amount of money and performance people in that environment are willing to spend on the problem. In highly reliable and available distributed systems, performance is secondary to the security and availability of data. But in the typical locally-distributed office/engineering environment people are most interested in the amount of overall performance for their money, and have not been willing to pay for traditional fault tolerance. Although the costs of reliability and non-stop availability may still be too expensive for this environment, we can provide increased availability without compromising performance by recovering quickly from crashes.

2. The Costs of Fault Tolerance

Systems with high reliability and non-stop availability incur costs of performance, complexity, and resources. Transactions trade performance and some complexity for reliability. Replicated hardware, data and processes trade resources for reliability and availability.

Transactions are necessary in a system such as QuickSilver [1] that provides failure atomicity for distributed activities, but transactions slow performance when compared to a typical office/engineering system such as Sprite [2] that does not provide this reliability. For example, QuickSilver uses transactions as a unified recovery mechanism for both volatile and recoverable resources, and has a low-overhead commit protocol, but its basic operations are slower than Sprite's [3] due to the extra reliability guarantees QuickSilver's operations must provide to the software above [4]. As a general rule, the less a system does, the faster it can do it, and this combined with a desire for simplicity is why the reliability provided by transactions has not yet become a popular part of the office/engineering environment.

Many systems provide availability with replication of processes or files, but this too is not appropriate in a system such as Sprite. [5] shows how process pairs can provide fault-tolerance of applications in an office/engineering environment (Mach), but in Sprite this would not address recovery of the kernel and file system, since for performance reasons, the file system runs in the kernel and not as a user process. The META Operating System [6] is also oriented towards availability and performance, and it uses file replication for availability. But the authors of META state that UNIX caching does not hide all of the performance cost of replication.

3. Fast Recovery In Sprite

There are several advantages to an alternate approach of using fast server recovery for availability in systems such as Sprite. Fast server recovery is simpler to implement and costs less in a number of ways. If a file server crashes and restarts in a matter of seconds, and client workstations can continue where they left off after just a brief pause, then we have good availability without the need to maintain replicated files. We can avoid the cost of replicated updates and the space spent on extra copies and the expense of extra servers. It is also easier to update kernel software by quickly rebooting and recovering rather than by updating the kernel while keeping the machine up and available. And as implemented in Sprite, we can make use of several features already present or planned for in the system.

Availability in Sprite is currently limited by the time required to reboot a file server. During reboot, the server spends the majority of its time recovering state with client workstations and checking disks, and it is these problems we are attacking first, using an improved client recovery mechanism and a high-performance file system that treats the file system as a segmented append-only log. Both of these solutions, the distributed state used for client-server recovery and the log-structured file system, are used for overall performance, and it is convenient that they can also be used for fast recovery.

3.1. Distributed State Recovery

In contrast to NFS [7], Sprite uses state on the file server in order to increase performance, but it must recover this state during a file server reboot [8]. For example, some of the state on the file servers is used to keep track of which workstations are caching which files. If a client workstation asks to read a file that is cached for writing on another client, the file server knows to find the most up-to-date copy of the file on the client writing the file. It also knows to turn off client caching if a file is being write-shared between workstations. If a server crashes and reboots, it must recover its knowledge of which clients have which files cached, so that it can enforce cache consistency on future open requests.

The solution in Sprite is to use the file system state replicated on the client workstations. Clients are aware of the set of files they have cached. When a client detects a reboot of a server, it recovers with the server by transferring its pertinent file system state to the server. After all interested clients have done this, the server once again has a consistent view of client caching.

If a server suffers a hardware failure and cannot reboot quickly, distributed state and disk arrays provide recovery possibilities. If the file server's disk is undamaged and can be accessed from another machine, the clients can push the appropriate file system state to the new machine, recovering with it instead. Sprite is also a part of the XPRS [9] project that is building a reliable array of inexpensive disks [10] to address recovery from disk failures.

The problem in distributed state recovery is to make sure its performance scales well as the size of the system scales, and we are currently testing several improvements to reduce the client recovery time. As the number of Sprite client workstations has increased, we've found that the first bottle-neck in recovery is the number of RPC requests that the file sever can handle concurrently. In response, we have added negative acknowledgements to our RPC system so that when a server cannot allocate an RPC server process to a client request, it can inform the client that it is not down, but merely busy. This way, the client knows that it has not lost communication with the server and can back off and try again later. This reduced our client state recovery time with 40 clients from an intolerable 15 minutes (or more) to 100 seconds. With these and several other techniques we believe we can easily reduce client state recovery to about 1/2 second per client, or about 25 seconds with 50 clients.

3.2. Using A Log-Structured File System

Even with fast client state recovery times, we still will not meet our quick recovery goals unless we substantially reduce the amount of time the server spends checking its disks, and to do this we will use new file system technology. Upon reboot, a Sprite file server checks its disks for unreferenced or multiply-referenced blocks in files to repair file system inconsistencies before the system starts running. As with most UNIX file systems, the current Sprite file system is not guaranteed to be in a consistent state after a server crashes. With 6 disks totaling 2 Gigabytes on a Sun-4, checking them in parallel during reboot requires about 15 minutes, and this time increases as the size of the file system increases.

A new file system, called LFS for log-structured file system [11], should give us much faster recovery. LFS, currently implemented and being tested on Sprite, treats the file system as a segmented append-only log. This allows LFS to increase our overall I/O performance by an order of magnitude, since it collects many small writes into one large disk I/O while maintaining fast reads. LFS periodically checkpoints its file system metadata and marks the place in the log where the file system is consistent. To recover, the server needs only to replay the log since the most recent checkpoint. As in most logging/checkpoint systems, the checkpoint intervals can be varied to make tradeoffs between checkpoint overhead and recovery time. We do not yet have our first recovery numbers available from the log-structured file system, but given general performance numbers of the first implementation, log recovery should be possible in under 40 seconds.

3.3. Recovering File System State from Non-Volatile Storage

If our attempts to reduce the recovery period for the file system state prove insufficient, we could instead store the file server's file system state in non-volatile storage. The most important piece of file system state is the handle table, which keeps track of which files are in use and by whom. One possibility is to use LFS to checkpoint and log changes to the handle table onto disk; higher-performance possibilities use varying amounts of non-volatile memory.

There is a logging and recovery interface to LFS that allows the system to log arbitrary data, checkpoint it and recover it, using the raw performance of the file system. This method is similar to the logging/checkpointing method described in [12] for virtual-memory data bases, but can handle a higher rate of updates. The problem is that logging in LFS is asynchronous, and some updates could be lost if a failure such as a power-outage does not allow the data to be flushed to disk before the system crashes. Synchronous updates would severely impair performance.

Using non-volatile memory to store the handle table or part of the recovery log is an attractive possibility, given recent improvements in price and performance of non-volatile memory. Some stable storage boards can offer performance only twice as slow as normal RAM, but with the advantages of write atomicity [13]. We could access the entire handle table directly in non-volatile memory, or we could put the tail of the LFS log in non-volatile memory to avoid the cost of synchronous writes to disk for file system metadata. The use of logging and a small amount of non-volatile memory to provide instant recovery is not new. This is how a database system such as Postgres [14] provides instant recovery.

4. The Disadvantages of Fast Recovery for Fault Tolerance

Despite the performance and simplicity advantages of fast recovery in the Sprite system, this technique provides only a small amount of fault tolerance. Without failure atomicity, recovering quickly from crashes does nothing to prevent data loss or inconsistency. Currently, if a Sprite file server is unable to flush its cache to disk before crashing, it will lose data that users may consider safe. Nor does fast recovery provide non-stop availability. While the file server is down, even if it is only for moments, there is no backup file server available.

Given the performance and cost goals of a locally-distributed office/engineering environment, reliability and non-stop availability may be impossible to provide. People in this environment have traditionally been willing to sacrifice reliability for speed. For instance, most UNIX file systems will allow 30 seconds worth of data to be lost in order to gain performance by batching together data transfers to the disk, a choice not permitted in on-line transaction processing environments. In addition, people in the typical office/engineering environment are not as willing to spend money for on-line replication of hardware, processes or files. Although fast recovery provides only minimal fault tolerance, it may be the most this environment can afford.

5. Conclusion

As distributed systems become larger and more popular, availability becomes more important in environments that traditionally are not willing to spend money on the problem. In these environments, tolerating faults means recovering from them quickly. We can provide cost-effective availability in these environments by making our systems recover in a matter of seconds.

6. References

[1] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. *Recovery Management in QuickSilver*. ACM Transactions of Computer Systems, vol. 6, No. 1, Feb. 1988. pp. 82-108.

[2] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. *The Sprite Network Operating System*. Computer, vol. 21, no. 2, February 1988. pp. 23-36.

[3] John Ousterhout. *Why Aren't Operating Systems Getting Faster As Fast As Hardware?* Proceedings of the Summer 1990 USENIX Technical Conference, June 1990. pp. 247-256.

[4] Roger Haskin. Personal Communication. September 30, 1990.

[5] Ozalp Babaoglu. *Fault-Tolerant Computing Based on Mach*. Operating Systems Review, ACM Press, January 1990. pp. 27-39.

[6] Kenneth Birman and Keith Marzullo. *The ISIS Distributed Programming Toolkit and The META Distributed Operating System*. Sun Technology, 2, 1 (Summer 1989). pp. 90-104.

[7] R. Sandberg, et. al. *Design and Implementation of the Sun Network Filesystem*. Proceedings of the Summer 1985 USENIX Technical Conference, June 1985. pp. 119-130.

[8] Brent B. Welch *Naming, State Management, and User-Level Extensions in the Sprite Distributed File System*. Technical Report Number: UCB/CSD 90/567, University of California at Berkeley. PhD Thesis, May 1990.

[9] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. *The Design of XPRS*. Proceedings of the 1988 VLDB, Los Angeles, CA, 1988.

[10] D. Patterson, et. al. *RAID: Redundant Arrays of Inexpensive Disks*. Proceedings of the 1988 ACM-SIGMOD Conference on Management of Data, Chicago, Illinois, June 1988.

[11] Mendel Rosenblum and John K. Ousterhout. *The LFS Storage Manager*. Proceedings of the Summer 1990 USENIX Technical Conference, June 1990. pp. 315-324.

[12] Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber. *A Simple and Efficient Implementation of a Small Database*. Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, 1987. pp. 149-154.

[13] J. P. Banatre, M. Banatre, G. Muller. *Architecture of Fault-Tolerant Multiprocessor Workstations*. Proceedings of the Second Workshop on Workstation Operating Systems, IEEE Computer Society Press, Sept. 1989. pp. 20-24.

[14] Michael Stonebraker. *The Design of the Postgres Storage System*. Readings 13th International on Very Large Databases, Brighton, England, 1987.