Fast Crash Recovery in Distributed File Systems

by

Mary Louise Gray Baker

A.B. (University of California at Berkeley) 1984
M.S. (University of California at Berkeley) 1988

A dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor John Ousterhout, Chair
Professor Randy Katz
Professor Rainer Sachs

1994

Fast Crash Recovery in Distributed File Systems

Abstract

Fast Crash Recovery in Distributed File Systems

by

Mary Louise Gray Baker

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor John Ousterhout, Chair

This thesis presents *fast crash recovery*: a simple, efficient, and inexpensive method for increasing availability in distributed systems. In fast crash recovery we assume that critical resources will fail, and we do not attempt to mask the failures with redundant hardware or software. Instead, we design the system to recover so quickly that there is little downtime. This approach is intended for environments than can tolerate occasional failures and cannot afford the cost and overhead of redundant resources.

In particular, I focus on fast recovery of distributed state. An example of distributed state is the file caching information maintained by servers in most modern file systems. This information describes the state of file caches on client workstations. After a crash, a server must recover this information in order to guarantee the consistency of the caches. Unfortunately, distributed state recovery can be slow and complex. The techniques I have developed reduce state recovery from several minutes to under six seconds for a Sprite file server [Ouster88] with 40 clients.

This thesis evaluates three distributed state recovery techniques based on their speed, complexity, and performance overhead. In *client-driven recovery* clients send their state information to the server after a crash. The server uses this information to regenerate its copy of the distributed state. *Server-driven recovery* is a modification of client-driven recovery that is faster and eliminates cache inconsistencies that can arise during client-driven recovery. The fastest technique is *transparent recovery*, so-called because client workstations do not communicate with the server during recovery. Instead, the server stores its distributed state in a protected area of its own main memory called the *recovery box*. The interface to the recovery box helps detect and prevent corruption of this state information.

To achieve fast overall recovery times, we must also recover other parts of the system quickly. For example, we can eliminate a lengthy file system consistency check by using a log-structured file system that recovers in seconds [Rosenb91]. By combining the improvements described in this thesis, a Sprite file server can reboot in under 30 seconds. This is two orders of magnitude faster than most modern file systems recover.

In addition to evaluating distributed state recovery techniques, this thesis presents some overall guidelines for designing distributed systems that will recover quickly from crashes.

Dedication

To F. F. and the others – I wish you were here.

Table of Contents

List of Figures

List of Tables

Acknowledgments

During my years in graduate school many people have given me their support, guidance, and friendship. Foremost among these people is my advisor, John Ousterhout. He has taught me the value of good design, hard work, and clear thinking. He applies his great creative and analytical skills with patience, encouragement, generosity, and an outrageous sense of humor. I am exceedingly lucky to have had such a conscientious and inspirational advisor.

I would also like to thank the other readers of this thesis, Randy Katz and Rainer Sachs, for their detailed and thoughtful comments. My interactions with Randy Katz in his classes and through his RAID project have always been very rewarding. His enthusiastic and practical approach to research is contagious. I have been fortunate to take courses from Rainer Sachs in both the mathematics and physics departments. Without his knowing it, his excellent teaching was responsible for my decision to come to Berkeley.

The other professors involved with the RAID project deserve my special thanks: Tom Anderson, Dave Patterson, and Mike Stonebraker. They have been tireless with their help and advice. I also thank Alan Smith for his careful answers to several technical questions.

Terry Lessard-Smith, Bob Miller, Kathryn Crabtree, and Heather Brown have all saved me from numerous disasters with quick, cheerful, and expert help. I can't imagine what I'll do without them.

The inhabitants of 477 Evans have been a joy to work with. Antony Ng, Ari Juels, Fred Douglis, Ken Shirriff, and John Hartman energetically debated subjects ranging from the state of the world to the perfect chocolate. Ken's 1 AM conference hot tub cookies have justifiably become world famous, and John has answered endless technical questions with wisdom and patience. I owe a lot to him, including such terms from his sparkling dialect as "bonehead" and "a few beers short of a six-pack."

I also owe a large debt to the other members of the Sprite project: Brent Welch, Mike Nelson, Fred Douglis, Andrew Cherenson, Adam de Boor, Mendel Rosenblum, Bob Bruce, Mike Kupfer, John Hartman, Ken Shirriff, Geoff Voelker, and Jim Mott-Smith. Without them, Sprite wouldn't exist. Brent Welch was particularly generous with his time and ideas as I began working on crash recovery. Mendel Rosenblum is a wonderful friend and mentor as well as a gifted researcher. I have learned a great deal from him and will continue to do so.

Many researchers in industry and at other universities have helped me along the way. I would especially like to thank Mark Weiser, Jeff Mogul, Dan Siewiorek, Anita Borg, Satya, Mike Kazar, Tim Mann, Ken Keller, Barbara Liskov, Greg Nelson, Brian Pawlowski, and Alex Wolf for their contributions to my work and their encouragement. Many thanks to Dan for presenting me with a copy of his infinitely useful book!

My parents and sister continue to offer unconditional love and support. Their combined understanding of art, music, writing, science, and engineering is truly impressive. No matter what my question, they always point me in the right direction. They have also kindly kept to themselves their astonishment that after years of studying computer science I still can't help much when the Macintosh does something weird. I'd also like to thank my grandmother, the rest of my family, and my husband's family for their determined and noisy rooting section!

There are many others who have made these years a pleasure. Annette, Kaz and Sara Wegmuller have often lifted my spirits. Ann Drapeau, Marti Hearst, David Bacon, Mark Sullivan, Dixie Baker, Nick and Mindy Lai, Dave Hitz, Eric Allman, and Greg Couch all steadfastly provided good cheer. Marti Hearst answered many writing questions. Mark Sullivan bravely used the recovery box implementation described in this thesis and gave me valuable feedback. Margo Seltzer's awesome energies have been immeasurable on both technical and confectionery projects. She is a superior researcher, teacher, and chef, and also a devoted friend. Her confidence in me was sometimes the only thing that kept me going. I have benefitted from Keith Bostic's technical know-how, while his friendship and "informative" mailings have given me a whole new perspective on the world. To the folks at Hillegass house: thanks for many fun times, including mud in Ferndale and our delicious Maglite barley wine.

Life here at Berkeley has been much improved by various other resources: Denis Kelly's wine tastings, Tilden Park, the UC Botanical Gardens, the fire trail, Akira Miike's fabulous banquets, the Nefeli Caffè, MST3K, Talk Soup, and Villa-Lobos' Bachianas Brasileiras 2 and 5.

Finally, I owe everything to my husband, Wendell Baker. No matter how busy he is with his own work, he always takes the time to listen, console, encourage, advise, and cook dinner. He never seems to realize that he gives all and expects nothing in return. I hope I can begin to be as supportive of him as he has been of me.

This research has been funded by ARPA grant N00600-93-C-2481, the California MICRO Program, NFS grant CCR-89-00029, and NASA/DARPA grant NAG2-591.

1 Introduction

Over the last ten years, networks of personal computers and workstations have replaced stand-alone timesharing systems in most computing environments. Faster, less expensive processors and improved network technology make distributed systems more cost-effective and flexible than their centralized predecessors. As an organization grows, it is easier to add capacity gradually to a network of computers than it is to a stand-alone system.

This change in technology has required a corresponding improvement in the performance and behavior of operating systems designed for distributed environments. Early network file systems were slow, often because they lacked tools to manage and to take advantage of their distributed resources. For example, Sun's Network File System, NFS [Sandbe85], is slower than more modern systems, because its file servers do not keep distributed state that describes how client workstations use their files. In contrast, modern distributed systems incorporate techniques for managing distributed resources efficiently and transparently. *Distributed state* is a key management tool. Servers and clients keep information about the state of the distributed system and use this information to control sharing of their resources.

Distributed state improves system performance and behavior, because it makes information available locally to hosts in the network. This reduces their need to retrieve data from central servers. For example, client workstations cache files in their main memories so they can access the files quickly without re-reading them from a file server. These cached files are a form of distributed state, because the files' ultimate storage is actually on the server. In turn, most modern file servers keep distributed cache state that describes which clients are caching their files. They use this state to guarantee coherence among the client file caches.

Unfortunately, distributed state also makes crash recovery slower and more complex. When a central resource such as a file server crashes, it loses the state information kept in its volatile main memory. If this information is necessary for correct system behavior, the server must reconstruct its distributed state before it continues to service client requests. For example, to guarantee cache consistency after a failure, the server must reconstruct its distributed cache state. Reconstructing distributed state often requires the server to communicate with each of its clients while handling increased disk I/O and processing loads. State recovery can be one of the most inefficient and stressful of system tasks and reduces system availability by increasing downtime.

This dissertation focuses on *distributed state recovery*, particularly cache and file system state. To provide higher system availability and manage distributed state more reliably we need efficient and well-understood state recovery. The dissertation evaluates three techniques for recovering distributed cache and file system state. The primary goal is to increase recovery speed, but I also consider the reliability, cost, and complexity of the recovery techniques. With good state management and fast recovery techniques, high-performance *stateful* systems (systems that keep state) can provide availability as high as or better than that of lower-performance *stateless* systems.

By combining fast state recovery with other fast recovery techniques, we can significantly improve overall recovery times. In particular, a disk storage manager such as LFS (the Log-structured File System) [Rosenb91] can recover the consistency of the file system on disk in only a few seconds. This contrasts with traditional UNIX file systems that require tens of minutes to ensure file system consistency after a failure. By combining fast state recovery, LFS, and other fast recovery techniques, the Sprite Distributed File System [Ouster88] recovers from crashes in under 30 seconds.

The faster a system recovers, the more available it is. In the limit, as recovery time approaches zero, a system with fast crash recovery is indistinguishable from a system that never crashes at all. I believe we can design systems to recover from the majority of failures so quickly that the downtime is hardly noticed. Therefore, this dissertation also explores optimizing the entire failure recovery sequence and proposes *fast crash recovery* as a new low-cost solution for increasing the availability of servers in high-performance, locally-distributed file systems. The goal of the fast crash recovery approach is to increase availability overall, by recovering quickly from the majority of failures.

Fast crash recovery contrasts with the traditional approach to providing high availability. The traditional approach uses *fault-tolerant techniques* to provide non-stop availability and very high reliability (no loss or corruption of data). To meet such requirements, fault-tolerant computing systems mask failures or prevent crashes entirely, usually with redundant hardware and software. Unfortunately, redundant hardware and software is expensive, and managing the replicated resources can be slow and complex. In contrast, fast crash recovery is inexpensive, since it does not require redundant hardware. It is faster during normal system operation, since it does not need to manage replicated resources beyond those required for normal system operation. And it is simpler, since it does not attempt to mask or prevent failures; it only attempts to recover the system very quickly.

Fast crash recovery is thus a viable technique for providing high availability in some environments, but not in others. It is appropriate for organizations that can tolerate occasional failures, but demand high-performance, low-cost computing systems. An example of such a system is the typical network of UNIX workstations found in many engineering organizations (the office/engineering computing environment). However, fast crash recovery alone is not appropriate in environments that cannot tolerate any downtime. Examples of such environments are on-line transaction processing systems and organizations with life-critical applications. These groups require fault tolerance and are willing to pay vast amounts of money for systems that guarantee non-stop availability. In these environments, the fast recovery approach is insufficient, because it cannot completely eliminate downtime after a failure, and it can only provide fast recovery after the majority of crashes. If a failure results from serious damage to a critical piece of hardware, a system using only fast crash recovery (no redundant hardware) will not be available until that hardware is fixed or replaced. Fortunately, such permanent hardware failures are becoming uncommon, as explained in chapter 2, and the faster we make crash recovery, the more environments will find it an appropriate solution for providing higher availability.

1.1. Measurement Environment and Applicability

I have implemented the fast recovery techniques evaluated in this dissertation in the Sprite distributed file system. Our Sprite system consists of a cluster of about 20 SPARCstation-1, SPARCstation-2, DECstation 3100 and DECstation 5000 workstations, almost all diskless and with 24 to 32 megabytes of main memory apiece. At its largest, the system included 40 client workstations. The cluster has a number of file servers, but most of the traffic is handled by a Sun-4/280 with 128 megabytes of memory and five to six gigabytes of disk space. All of the workstations in the cluster run the Sprite network operating system. Sprite is largely UNIX-compatible, and most of the applications running on the cluster are standard UNIX applications. In addition, Sprite provides process migration [Dougli91] which allows users to off-load jobs easily to idle machines in the cluster.

The Sprite user community has included operating systems researchers, architecture researchers working on the design and simulation of new I/O subsystems, a group of students and faculty working on VLSI circuit design and parallel processing, administrators, and graphics researchers. At various points in the project, our Sprite system simultaneously supported 40 day-to-day users and more than 40 occasional users. It is still the day-to-day computing system for me, for the other members of my research group, and for various other students in the department. The broad set of users and wide variety of applications run on Sprite make it a good system for evaluating file system research. Any problems with crash recovery are bound to be reported, loudly and indignantly.

Sprite provided part of the motivation for this thesis, along with the necessary file system features (such as file caching on client workstations for high performance), but the results of this thesis are more widely applicable. Long recovery times are a problem for many distributed systems, not just Sprite. Many of the techniques described in the thesis, such as reuse of kernel text and initialized data, are applicable to the majority of systems. The techniques for recovering distributed cache state are applicable to current and future file systems [Kazar90][Mann93] that cache file data on clients. In addition, caching is not the only source of distributed state information. The techniques described here are applicable to other varieties of state information such as name and address translation data for network name services and client/server connection information for distributed databases. Finally, frequent server reboots are a problem for many systems, and not just Sprite. Measurements from Internet sites [Long91] indicate that UNIX machines fail on average once every two weeks.

1.2. Outline of Dissertation

This section outlines the thesis and lists its overall results by chapter.

The next chapter of this dissertation gives more motivation and background. First, it describes the distributed cache state recovery problem in more detail. It explains how Sprite client workstations achieve higher file system performance by caching file data, and how the file server uses distributed state information to guarantee cache consistency between the clients. Because the file server keeps this distributed cache state in data structures in its volatile memory, the state information is lost on the server if the server crashes. As the server returns to service after a failure, it must recover this information to continue to guarantee cache consistency. The next section of the second chapter describes a variety of highly-available and fault-tolerant systems to give further motivation for the fast recovery approach to high availability. The last section lists other systems that use main memory to store data across failures, a mechanism described in chapter 5 of this thesis.

Chapters 3 through 5 describe and evaluate three different techniques for recovering distributed state. Chapter 3 covers the first technique, called *client-driven recovery*. In this form of recovery, first designed and implemented by Brent Welch [Welch90], client workstations of the file server keep track of which of the server's files and other objects they have cached or opened. When they detect that the file server has crashed and rebooted, the clients send their state information through to the file server. The server uses this information to reconstruct the main-memory copy of the state information it needs to ensure cache consistency between the clients. Although this recovery technique is the least complex one for Sprite, it also has three disadvantages. First, it leads to congestion on the file server. This congestion exposed flaws in Sprite's communication protocol that allowed cache state recovery to become unstable, sometimes taking ten or fifteen minutes to finish. Second, even with a variety of improvements to prevent instability and server congestion, client-driven recovery is still the slowest state recovery technique, requiring an average of 21 seconds to recover the distributed cache state on a SPARCstation-2 server with ten clients. The server spends much of this time idle, waiting for clients to send it their state information. (Note that these are the times to recover just the distributed state. Overall recovery takes longer, as described below.) A third important problem with client-driven recovery is that it permits cache consistency violations for a short time during and after the server recovers.

The second recovery technique, evaluated in chapter 4, is *server-driven recovery*. This technique is faster than client-driven recovery, requiring about two seconds for a SPARCstation-2 server with ten clients, and scaling linearly to about six seconds for a server with 40 clients. Disk I/O on the file server is the limiting performance factor for server-driven recovery. This technique also eliminates the problems with cache inconsistencies after the server recovers. In server-driven recovery, the server keeps a list in stable storage of clients that have opened or cached its files. For recovery, the server contacts the clients on its list for information about these files. When contacted, the clients send the server their state information. The server then rebuilds its volatile data structures for the distributed cache and other file system state in the same manner used by client-driven recovery. The difference between server-driven and client-driven recovery is that the server has a complete list of the clients it must contact and therefore knows when the recovery process is done. This means it need not wait to see if further clients contact it, and it can hold off granting new services to clients until all clients have recovered. With this advantage the server avoids cache consistency violations and can better control recovery. This chapter also describes problems with server-driven recovery, including server dependence on clients for good recovery speed and increased synchronization complexity in the kernel. Despite these problems, server-driven recovery is likely to be the technique of choice for many systems.

Chapter 5 evaluates the third, and fastest, technique used for recovering distributed state, *transparent recovery*. This form of recovery requires about 1.5 seconds for ten clients, scaling linearly to a little over five seconds for 40 clients. Server disk I/O is still the limiting performance factor, but transparent recovery introduces new possibilities for reducing the amount of I/O. I call the technique *transparent*, because it requires no communication between the clients and the file server for recovery. To avoid gathering the distributed cache state from clients after a failure, the server preserves its state information across failures. However, the state information is updated too frequently during normal operation for the server to maintain it on disk. I have therefore developed a mechanism called the *recovery box*, that allows the server to store the state information in its main memory and treat that main memory as if it were stable storage. The server can then update the information at main memory speeds, while preserving the information in that area of memory across failures. The recovery box itself provides no extra protection against power failures, but it detects and prevents stray memory writes that could corrupt the data preserved there. This chapter includes details on the design and implementation of the recovery box and its applicability to other systems and programs. The recovery box technique is useful outside of cache state recovery, but transparent recovery can be hard to retrofit into existing systems.

Chapter 6 covers other requirements for fast recovery. Besides recovering its distributed cache state, a file server goes through a number of other steps to reboot after a crash. This chapter lists these steps and how I've improved the performance of each, often with the help of others' work. With transparent recovery and the other improvements listed in this chapter, it now takes under 29 seconds overall to boot a SPARCstation-2 file server with 40 clients. This is two orders of magnitude faster than most distributed file systems.

Chapter 7 gives some advice on designing systems to recover quickly from crashes. In particular, the design should include fast recovery from the beginning, because it can be harder to add after the rest of the system has been worked out. A second point is that designers should avoid start-up overhead. A common design trade-off is to do some extra start-up processing if it means the system will perform better during normal operation. This is usually the right trade-off, but the start-up processing adds up quickly and can result in slow recovery. Techniques that use main memory for storing system state, such as the recovery box, make it possible to have good overall performance and faster recovery times. A third point is that designers should build tools to manage and debug distributed state, to make sure it is kept consistent. Finally, recovery of the state information will be much easier if updates to it are easy to locate and isolate in the code.

The concluding chapter gives a side-by-side comparison of the distributed state recovery techniques and sums up the results of the thesis. Recovery is one of the most difficult and stressful activities a system must perform. I hope this dissertation will encourage designers to pay more attention to recovery as a part of system design.

1.3. Contributions

In summary, the main contributions of this thesis are

- The comparison of three distributed state recovery techniques.
- The evaluation of fast recovery as a low-cost solution for providing high availability.
- The evaluation of the recovery box as a mechanism for treating main memory as stable storage across system failures.
- A guide for designing systems that recover quickly.

1 Background

This chapter gives more background and motivation for the distributed cache state recovery problem and for the fast recovery approach to high availability. The first section of the chapter describes distributed file systems that must recover distributed cache state after a failure. It first details the distributed cache state recovery problem in the context of Sprite and then lists other systems with similar recovery problems. The second section describes some of the many highly-available systems and contrasts their methods to the fast recovery approach. The third section describes systems that store state information in main memory for fast crash recovery, as does the recovery box, described in chapter 5.

1.1. The Distributed Cache State Recovery Problem

File systems such as Sprite, Spritely NFS [Mogul92][Sriniv89][Mogul93], DEcorum [Kazar90], and Echo [Hisgen89][Mann93] all share a need to recover distributed cache state. The need arises, because the systems allow file data caching on client workstations and also guarantee consistency amongst the client caches.

The clients cache file data in their main memories to increase file system throughput and reduce the latency of file data accesses. Due to the locality of file accesses, the clients are likely to re-access the cached data soon. A client request satisfied from its local file cache progresses at main-memory speed, avoiding a slower access of the data across the network on a file server. This also reduces the load on the servers themselves. For example, Sprite's file data caches on clients absorb 60% of read accesses and reduce total file traffic between clients and servers by at least 50% [Baker91b].

However, the same file can be stored and modified by different clients, so a cache consistency policy is necessary to ensure that clients all view consistent copies of the data. To implement this policy, a file server keeps information about which clients are caching which of its files. However, the file server can lose this information if it crashes. Recovering this information after a server crash is the *distributed cache state recovery problem*.

The following section explains the problem in more detail, organized as follows. For comparison, I first describe an older file system, NFS [Sandbe85], that has made an opposite set of trade-offs from Sprite's. NFS has no state recovery problem, because it limits client file caching and provides only weak cache consistency. The section then lists the performance benefits Sprite gets from caching file data, including modified (or *dirty*) data, on clients. This is followed by an explanation of why Sprite's cache consistency policy is necessary and how it works. The section then explains the recovery problem that results from Sprite's cache consistency guarantees. The remainder of the section describes some other systems with similar recovery problems.

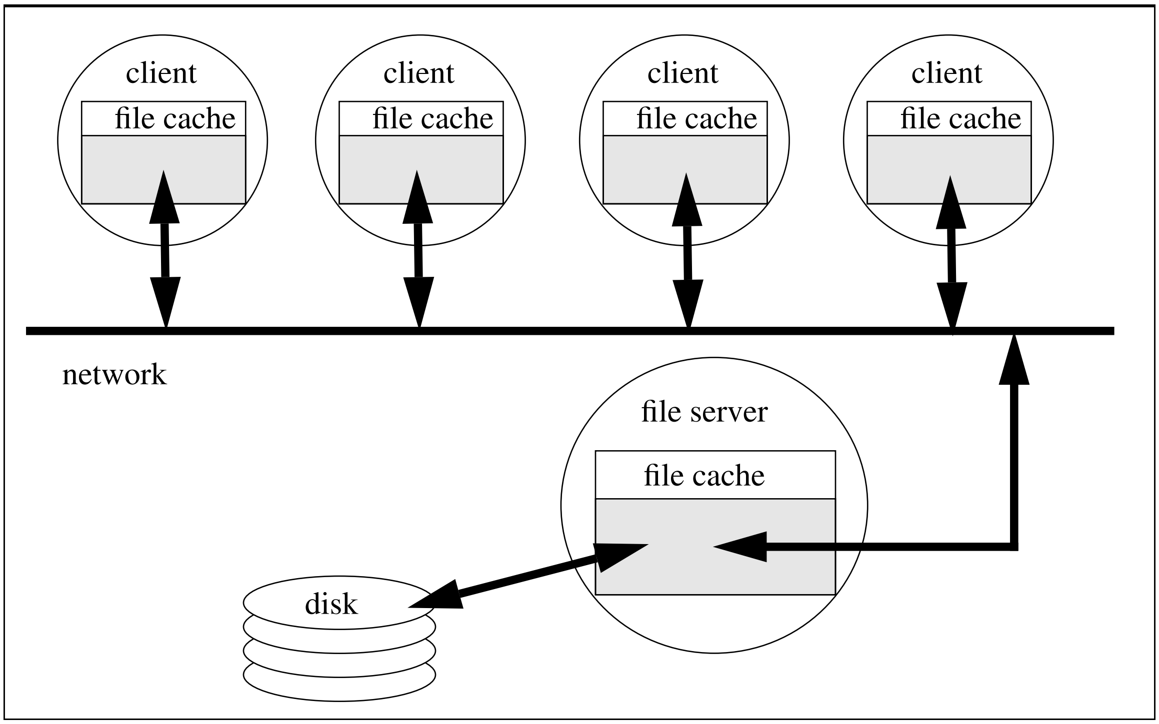

Figure 1-1. Client-server distributed file system.

This figure illustrates a client-server distributed file system, such as NFS. The clients read file data across the network from the file server, unless they find the data already in their main-memory caches. To service a read request, the server first checks its cache for the data. If the data is not cached, the server reads the data from disk and returns it to the clients. Likewise, the clients write data from their caches to the server, which in turn writes data from its cache to its disk.

1.1.1. NFS

NFS is a commercial network file system that has become a standard for workstations on local-area networks. It uses a client-server model and allows clients to cache file data in their main memories. Figure 1-1 shows NFS client workstations, with local caches, connected by a network to a file server with disks and a local cache. To read data from a file, an NFS client looks for the data in its local cache. If the data is in the cache (a *cache hit*), the client does not need to communicate with the server. Otherwise the access is a *cache miss*, and the client requests the data from the server. If the request misses in the server's cache, the server retrieves the data from disk into its cache, and then sends the data to the client, which puts it into its own cache.

NFS is designed primarily to be simple; cache consistency and performance are only secondary goals. To simplify the design, NFS file servers are *stateless*: servers do not keep state in their main memories unless it is also on disk. This means they lose no state information if they crash, and they therefore have no state recovery problem. In particular, NFS servers do not keep state information about which clients are caching which files.

NFS's simple stateless approach leads to consistency problems which in turn lead to performance problems during normal processing. Because an NFS server does not know which clients are caching which files, it cannot inform clients when a file they have cached has been modified by another client. The client caches can thus become inconsistent, and some clients may continue to use the old (or *stale*) data in their caches.

To limit these inconsistencies, clients must poll the server periodically to determine if files they have cached have been updated. Polling puts an extra load on the server and adds latency to some client file accesses. Whenever a client accesses a file, the client checks how long it has been since it last polled the server about that file. If it has been longer than some number of seconds, the client asks the server for status information about the file. This status information includes the date of the last modification for the file. If the file has been modified since the client last polled the server, the client invalidates its cached data for the file. In most NFS implementations, the clients poll the server if a file's status information is older than three to 60 seconds. However, this still leaves at least three seconds during which a client may use stale data. Section 1.1.3 presents data showing the negative impact of this weak cache consistency scheme.

The NFS trade-off in favor of simple recovery and weak cache consistency has even greater performance problems than those mentioned above. To reduce the opportunity for client cache inconsistencies, most NFS implementations use a *write-through-on-close* cache policy. When a client modifies file data in its cache, it sends the new data back to the server when it closes the file. The application's file close operation blocks until the file data has been written all the way through to the server's disk. This makes the new data available quickly for other clients polling the file, but it increases the delay for the client application, and it increases the load on the network and the file server. Most files are only open for a short time [Baker91b], so the write-through-on-close policy means that clients write back their dirty data almost immediately. This ties their write performance to the speed of the server's disk. The following section contains comparative performance measurements between NFS and Sprite, which does not write back dirty data as soon as a file is closed.

1.1.2. Client Caching in Sprite

The Sprite distributed file system uses a client-server model similar to that of NFS, but with an almost opposite set of design decisions. In Sprite, performance and cache consistency are more important than the simplicity of state recovery. Sprite therefore allows more aggressive file caching on clients and guarantees that all clients see a consistent view of the file data.

Unlike NFS, Sprite's client caches delay writing dirty data back to the file server. Sprite allows dirty data to remain in the client's cache for about 30 seconds. This *delayed write-back policy* provides two benefits. First, if the dirty data is either deleted or overwritten while cached, it need never be written across the network through to the file server. This reduces the load on the network and file server. Second, even if the data is written back to the file server after some period of time, the write operation is performed *asynchronously* with respect to the application that issued the write. This means the application does not block until the write is finished, so its performance is not limited by the write operation.

Sprite's delayed write-back policy provides better application performance that the write-through policy used by NFS. Comparisons between NFS and Sprite in [Nelson88] made on Sun-3/50 and Sun-3/75 workstations show that Sprite is at least 30 to 40% faster, due in large part to its writing policies. More recent measurements in [Ouste90a] on faster workstations show that Sprite is now 50 to 100% faster. This is because the penalty for using NFS grows worse as machines get faster; disks and networks have not increased in speed as fast as have CPUs, and Sprite makes fewer disk and network accesses than NFS.

1.1.3. Sprite's Cache Consistency Policy

Permitting client workstations to cache dirty file data improves file system performance, but it also permits potential cache inconsistencies. These cache inconsistencies would violate one of Sprite's goals. Sprite is designed to provide the same view of the file system to clients as would a single time-shared machine, despite file caching. Thus, all clients accessing data from a particular file should see the same, and most recent, view of that data. Sprite's cache consistency policy guarantees this. This section further explains the cache consistency problem and shows how Sprite solves it. It then presents data showing that the cache consistency policy is necessary to prevent clients from accessing inconsistent data. The data also shows that the policy is not invoked often enough to degrade overall file system performance.

Sprite caches can become inconsistent due to *write-sharing*. Figure 1-2 illustrates how this occurs through *sequential* and *concurrent* write-sharing. Sequential write-sharing occurs when clients sequentially access a file, with at least one of them modifying it. The write-sharing is sequential if the file is only open on one client at a time. In the figure, one workstation (called larceny[1]) opens a file for reading and caches it. After larceny closes the file, another machine (called gluttony) opens and writes the file, leaving dirty data in its cache. After gluttony closes the file, larceny opens it again. Without a cache consistency policy, gluttony will read the stale data from its own cache, because it does not know that larceny has modified the file.

Figure 1-2 also illustrates concurrent write-sharing. Concurrent write-sharing occurs if clients have a file open at the same time, with at least one client writing the file. In this case, the client writing the file, gluttony, has the most recent data for it. But without a cache consistency policy, larceny will not know that another client is modifying the file, and it will continue to use the stale data in its own cache.

To ensure that all clients see the most recent data in the presence of write-sharing, Sprite's cache consistency policy does three things. First, the server maintains version numbers on files. The server increments a file's version number whenever the file is opened for writing. Whenever a client opens a file for reading or writing, the server returns the file's version number. The client saves this number for as long as it caches data from the file. If the client opens the file again later, it com-

1. All machine names in this thesis are actual names of hosts in the Sprite network at U.C. Berkeley. Sprite uses a "spices and vices" naming scheme, but there seems to have been rather more interest in vices than in spices.

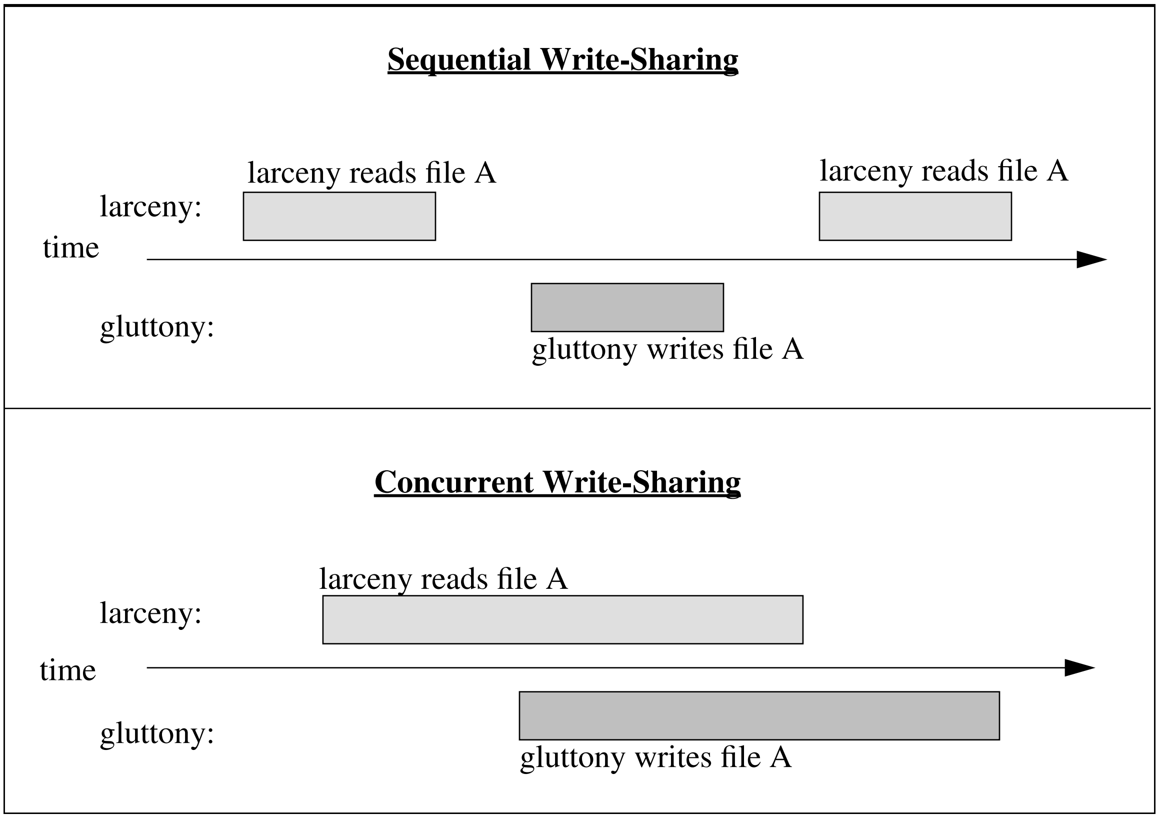

Figure 1-2. Sequential and concurrent write-sharing.

This figure illustrates sequential and concurrent write-sharing using two client workstations called larceny and gluttony. In sequential write-sharing, clients take turns accessing a file, with at least one of them writing it. In this example, larceny first opens file A for reading. It then has a copy of that file data in its local main-memory file cache. After it closes the file, another workstation, gluttony, opens the file for writing. Gluttony now has a modified version of the file data in its local cache. If larceny then opens the file again, without a cache consistency policy, it will read stale data from its local cache, since the new data for the file is in gluttony's cache. In concurrent write-sharing, more than one client opens the file at the same time, with at least one client writing the file. A cache consistency policy is necessary to ensure that any client sharing the file always receives the most up-to-date data.

pares the old and new version numbers. If the version number has changed, the client knows that its cached data for the file is invalid.

Second, the server keeps track of all the clients that are caching its files and the location of the most recent data for each file. The server is able to do this, because the clients send it all their open and close requests. When the server receives an open request for a file from a client, it records in a main memory data structure the fact that the client is caching that file and whether it has opened the file for reading or writing. The file server's list of which clients are caching which files is the *distributed cache state*, and the server is called *stateful* because it maintains this state.

Third, the server enforces cache consistency by telling clients to invalidate their caches, to write back dirty data, or to make a file uncacheable. To do this, Sprite file servers make cache consistency *call-backs* to their clients. When the server receives an open request, it consults its distributed cache state to determine if the request requires any call-back. If so, it calls back to the appropriate clients to tell them what to do. A call-back may be necessary if the open request causes sequential write-sharing. It is always necessary if the request causes concurrent write-sharing.

These call-backs are illustrated in Figure 1-3. In the sequential write-sharing case, a call-back is necessary if a client still has dirty data in its cache when another client opens the file. The server calls back to the client caching the dirty data and tells it to write the data back to the server. The new data is then available on the server before it responds to the new client's open request. In the concurrent write-sharing case, the server calls back to all clients sharing the file and tells them to stop caching the file. If one of the clients has dirty data for the file, it is also told to write that data back to the server. All reads and writes on the file then go directly through to the file server, so the server can make the most recent file data available to all the clients. The file remains uncacheable until all clients have closed it. Turning off client caching during concurrent write-sharing may add latency to individual client read and write requests, but concurrent sharing is so rare that using this simple mechanism does not significantly affect overall system performance [Baker91b].

Given this explanation of Sprite's cache consistency policy, there are still questions to answer. First, is the cache consistency policy really necessary? We can answer this by looking at how often Sprite clients and users would access stale data if we used a weaker cache consistency scheme, such as NFS's. Second, is Sprite's cache consistency policy likely to degrade overall system performance? We can answer this question by looking at how often consistency actions are invoked. Finally, are there cases where the policy does not guarantee a consistent single image of the file system across clients? The rest of this section addresses these issues. I show that the policy is necessary to prevent stale data errors, but that it is not invoked often enough to degrade file system performance, and that it provides a consistent view of the file system except during the unlikely event of a network partition.

The first question is whether a weaker but simpler cache consistency policy would suffice. To estimate the negative impact of a weaker cache consistency scheme than Sprite's, I used Sprite file system traces to simulate a cache consistency mechanism similar to that in an implementation of NFS. In the simulated mechanism, a client considers data in its cache to be valid for a fixed interval of time (the *refresh interval*); on the next access to the file after the expiration of the interval, the client polls the file server and updates its cache if necessary. New data from clients is written through to the server almost immediately in order to make it available to other clients. But if one workstation has cached data for a file while another workstation modifies the file, the first workstation may continue to use its stale cache data until the end of its refresh interval.

To drive the simulation, I used a set of eight 24-hour traces of Sprite file system activity. To obtain these traces, we instrumented our file server kernels to record logical file system events such as file opens and closes. Because Sprite clients send all open and close requests through to the server, we are able to trace much of the cache behavior of individual clients. More information about theses traces can be found in [Baker91b].

Table 1-1 presents the results of these simulations. A 60-second refresh interval would have resulted in many uses of stale data each hour, and one-half of all users would have accessed stale data over a 24-hour period. Although a three-second refresh interval reduces errors, about three out

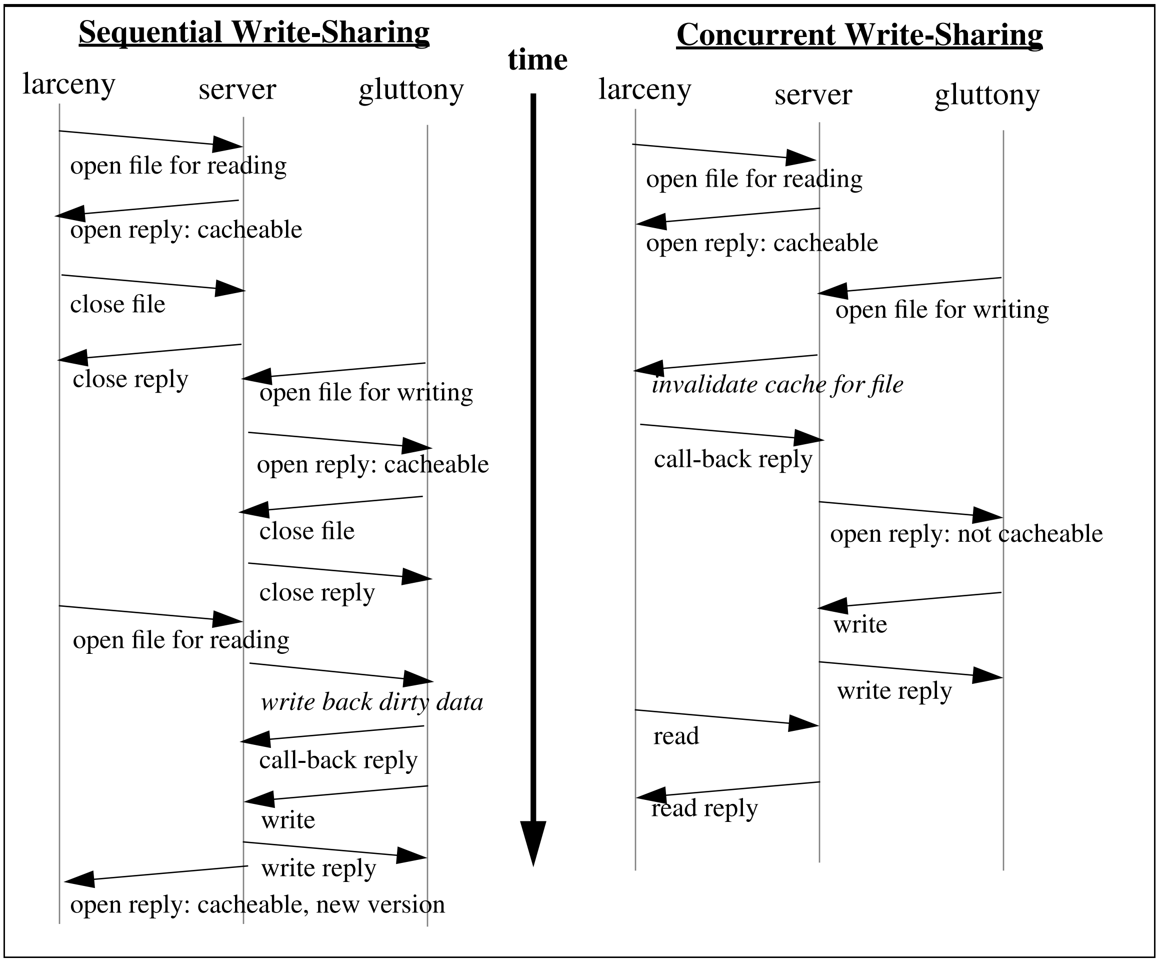

Figure 1-3. Sprite cache consistency call-backs.

This time line illustrates the sequence of calls and cache consistency call-backs that occur in Sprite during sequential and concurrent write-sharing of a file. Consistency callbacks from the server are in italics. For sequential write-sharing, the server responds to larceny's second file open request by first calling back to gluttony to ask that it write back its dirty data for the file. By checking the file's version number in the open reply, larceny learns that another client has updated the file. Larceny then invalidates its own cache for the file. It can then read the most recent data for the file from the server. In concurrent write-sharing, the server responds to gluttony's request to open the file for writing by calling back to larceny to tell it to invalidate its cache and stop caching the file. As part of the reply from the open request, the server informs gluttony that the file is not cacheable. In this way, the server tuns off client caching for the file. During concurrent write-sharing, all read and write requests from the clients go through to the file server.

of 52 users would still have received stale data at some point in an average 24-hour period. This is still a large number of potential file access errors, particularly when compared to other sources of error such as undetected network or disk errors. The level of file sharing in Sprite makes a cache consistency policy necessary.

Measurement	60-second	3-second
Average errors per hour	18 (8-53)	0.59 (0.12-1.8)
Percent of users affected during 24 hours	48 (38-54)	7.1 (4.5-12)
Percent of users affected over all traces	63 (NA)	20 (NA)
Percent of file opens with error	0.34 (0.21-0.93)	0.011 (0.0001-0.032)

Table 1-1. Stale data errors.

This table lists results from simulations of a weaker cache consistency mechanism based on polling: clients refresh their caches by checking the server for newer data at intervals of 60 seconds or three seconds. Errors are defined as potential uses of stale cache data. Effected users are those whose processes suffered errors. The numbers in parentheses give the minimum and maximum values among the eight traces. NA means not applicable.

Sprite is not the only environment in which file write-sharing occurs. A year-long trace of file activity in AFS [Satyan90], a distributed system with over 400 users, also shows some sharing. In AFS, the chances of a file being modified by two different users in less than a day are between 0.09% and 0.72%, depending upon the type of file [Kistle91]. While the measurements show less sharing than in Sprite, they still indicate a significant percentage of possible consistency errors in the absence of an effective cache consistency policy. Also, these measurements do not include possible consistency violations due to the same user modifying a file on more than one machine.

The Sprite environment, however, encourages users to share files between machines. In Sprite, a user may edit, compile, and run the same program on different machines and trust that the cache consistency policy will handle conflicts. Sprite also allows processes to migrate transparently to idle nodes in the network. As a process migrates from one machine to another, its data accesses can cause sequential sharing across those machines. As a result, same-user file modifications in a system such as Sprite can require a cache consistency policy.

The same trace data from Sprite helps answer the second question: whether consistency actions are invoked often enough to degrade overall performance. Table 1-2 presents this data. About one in every 300 file opens causes a file to be opened for reading and writing on more than one machine. In addition, for about one in every 60 opens, the server recalls dirty data from another client’s cache in order to obtain the most recent version of a file. This is an upper bound on the required number of server recalls, because the Sprite server actually issues more call-backs than necessary. It sometimes fails to keep track of whether a client has already finished flushing its dirty data back to the server via the delayed write mechanism of the cache. Our conclusion from this data is that Sprite’s cache consistency actions are not invoked often enough to have a large impact on overall performance.

The last question is whether there are cases for which Sprite’s cache consistency policy fails to guarantee a consistent image of the file system across clients. Sprite’s cache consistency policy works well unless a network partition occurs between the server and a client. If a server loses contact with a client for a significant period of time (explained in chapter 3), the server assumes that

Type of action	File opens (%)
Concurrent write-sharing	0.34 (0.18-0.56)
Server recall	1.7 (0.79-3.35)

Table 1-2. Consistency action frequency.

This table gives the frequency of various consistency actions, measured as a percent of all file opens, excluding directories. Concurrent write-sharing refers to opens that result in a file being open on multiple machines and open for writing on at least one machine. Server recall refers to opens for which the file's current data resides on another client and the server must retrieve it. The numbers in parentheses give the minimum and maximum values among the eight traces.

---

the client has crashed. The server will then clean up the cache state associated with the client, so that it can grant requests from other clients to access files that were cached by the lost client. If the lost client has not actually crashed, though, it will still have the files cached. During the network partition, it will not see other clients' modifications to its files, thus violating the single-system image of the file system. Also, when the lost client is able to contact the server again, it will not be permitted to write its dirty data back to the server for any file that has been modified during the partition by another client. In this sense, the isolated client is treated as if it crashed and lost its dirty data. The Echo system, described later in this section, guarantees a single-system image across clients even during a network partition, but it too can lose dirty data on clients separated from their server. Fortunately, on Sprite's locally-distributed network, we almost never experience network partitions. This could be a more severe problem in a wider-area network.

1.1.4. Sprite's Distributed Cache State Recovery Problem

Keeping distributed cache state in main memory data structures on the file server solves Sprite's cache consistency problem, but it creates a crash recovery problem. The cache state information can be lost when the file server crashes and must be recovered before the server can continue to satisfy client requests while maintaining cache consistency.

Originally, Sprite did not attempt to recover distributed cache state. To avoid cache inconsistencies after a server failure, users were forced to reboot their client workstations. While this now sounds like a bad decision, it did not seem so before we acquired much experience with distributed computing. After all, in a central time-shared system, terminal users must all login again after a machine crash. But workstation users keep more state information locally than terminal users; window set-ups and on-going processes are inconvenient to restart. The situation soon became intolerable. Clearly, Sprite needed a mechanism to recover distributed cache state that would allow client workstations to continue to operate seamlessly after a server reboot, without losing any information.

This thesis evaluates three such methods for recovering distributed state, all implemented in Sprite. The first method, described in chapter 3, is client-driven recovery, in which clients initiate recovery with the server. The central idea for this technique is that each client workstation keeps track of the files it has open or cached. Thus each workstation has a copy of its own state information. To recover, the clients send their state information to the server, and the server uses this information to rebuild its distributed cache state. The second method, described in chapter 4 is server-driven recovery. This is a modification of client-driven recovery in which the server initiates contact with the clients to request their state information. The third method is transparent recovery, described in chapter 5. For transparent recovery the server keeps the distributed state information in stable storage, preserving it across failures. After a failure the server retrieves the information from its stable storage, so that it does not need to communicate with clients to recover.

This thesis concentrates on recovery in Sprite, but many other systems have the same task. Like Sprite, these systems cache file data on clients for performance but keep distributed state to guarantee client cache consistency. The following sections describe how three other file systems recover their distributed cache state. The choice of recovery mechanism in the first two systems was influenced by Sprite. Spritely NFS [Mogul92][Mogul93][Sriniv89] uses server-driven recovery, and DEcorum [Kazar90] uses client-driven recovery. The third system, Echo [Hisgen89][-Mann93], instead maintains a backup server with a copy of the distributed state. This technique is similar to transparent recovery, except that the stable storage for the server's state is actually the main memory of another server.

Unfortunately, the following sections provide only limited recovery performance data. Spritely NFS is the only one of the three systems to publish such data so far. While other aspects of system design and performance receive much attention in the literature, recovery usually does not.

1.1.5. Spritely NFS

Spritely NFS addresses the cache consistency and performance problems of NFS by adding Sprite's cache consistency protocol to NFS. A goal of Spritely NFS is to show that a stateful system can provide strong cache consistency guarantees and high performance without a complex recovery implementation.

To guarantee cache consistency, Spritely NFS turns an NFS stateless server into stateful server using protocols similar to those in Sprite. A Spritely NFS server maintains distributed cache state that it uses to enforce cache consistency among its clients. To maintain the distributed cache state, Spritely NFS modifies the NSF protocol to add new client-to-server open and close calls. As in Sprite, the server can then monitor which clients are caching which files by tracking their file open and close requests.

The stateful server also addresses the performance problems of standard NFS. Because the server can enforce cache consistency, it is safe for clients to delay writing back dirty data from their caches. Using this delayed write-back policy, Spritely NFS clients gain improved performance over standard NFS clients which must use a write-through-on-close policy to reduce possible cache consistency violations.

To recover the cache state information after a server crash, Spritely NFS uses an implementation of the server-driven recovery described in chapter 4. On disk, the server keeps track of which clients it must contact for recovery. After rebooting it asks those clients to send it their cache state information. The server regenerates its cache state information by combining the state sent by the clients. Server state recovery takes about two seconds for a single client with 393 open files. Thus recovery seems to take about five to seven milliseconds per open file.

Spritely NFS makes a strong argument in favor of stateful systems. It shows that it is possible to make a few changes to a stateless protocol to turn it into a stateful one and gain performance and cache consistency advantages without introducing great complexity.

1.1.6. DEcorum

Transarc's DEcorum [Kazar90] file system for the Distributed Computing Environment is based on a previous file system called AFS [Howard88][Satyan90] but has made several modifications to provide higher performance and stricter cache consistency guarantees. While AFS only allows clients to cache file data on their local disks, DEcorum allows diskless clients to cache file data in their main memories for higher performance. DEcorum also uses a delayed write-back policy for cached dirty data on clients, rather than AFS's write-through-on close policy. This provides higher performance and better cache consistency guarantees. With these features, DEcorum's use of client caching is very similar to Sprite's.

DEcorum also uses a cache consistency policy similar to Sprite's. DEcorum provides a consistent single-image of the file system across clients, but it implements its cache consistency policy using *tokens*. A token represents permission to access a file or other object in some manner. The token manager on the file server grants tokens in response to client requests to open, read, write or otherwise access files. A client must not read or write cached file data until it has received the appropriate token from the file server. As in Sprite, the server's token manager intercepts all file open and close requests (and all other client calls through the Vnode [Kleima86] interface). The token manager keeps track of which clients have which type of tokens for which files. If a new token request could cause a cache consistency conflict, the manager calls back to clients to revoke the conflicting tokens, thereby notifying the clients that their cached data will no longer be valid. When the manager revokes a write token, the client must also write back any modified file data before returning the token. Thus DEcorum prevents cache inconsistencies during write-sharing by ensuring that only one write token exists for the shared data. This allows only one client at a time to modifiy the shared data. Before that client is granted the write-token, the server revokes all read tokens. This prevents other clients from reading stale shared data.

DEcorum's cache consistency policy differs from Sprite's in a few ways. DEcorum clients cache file status information as well as file data. The file status, or *attributes*, includes such information as file size, ownership, and last modification date. Caching this information makes updates to the file attributes faster. Also, DEcorum can associate tokens with byte ranges on files, rather than just whole files. This allows for finer-grained file sharing. For example, one client can obtain a write token for the beginning of a file, even if another client currently has a token for the end of the file. Thus one client can cache and modify the beginning of a file while another client caches the end of the file. This may be of limited benefit, though, unless the level of file sharing increases in distributed systems. Finally, in order to handle network partitions, servers wait two minutes before beginning to grant token requests that conflict with those held by a client they assume to have crashed. This allows the isolated client more of an opportunity to rejoin the system before its cached data can become stale. However, two minutes is a long time to delay the new token requests from other clients.

DEcorum's recovery state problem is similar to Sprite's. If a DEcorum server crashes, it loses its token information and must recover it. DEcorum uses an implementation of the client-driven recovery technique described in chapter 3, but with more attention paid to recovery after network partitions [Kazar93]. Clients detect when servers return after a crash or network partition. They then make calls to the server, and the server responds with an indication of whether it truly crashed and lost the token information. For token recovery, each client scans a table looking for files that are either currently open or have cached dirty data. For each such file, the client then calls the server to re-establish the token with it. Currently, no performance data is available concerning DEcorum's token recovery protocol.

1.1.7. Echo

Echo [Hisgen89][Mann93] is interesting in the context of this dissertation for at least two reasons: Echo provides high availability through replicated file server CPUs and disks, a method that contrasts with Sprite's approach, and it keeps distributed cache state information similar to Sprite's. This section describes both aspects of Echo, because Echo's use of replication affects how it performs distributed cache state recovery.

For high availability, Echo allows both file servers and disks to be replicated. Figure 1-4 illustrates an Echo system. With this replication scheme, Echo remains available despite the crash of a server CPU or a disk. When a primary server fails, its backup takes over the primary's duties and has access to the same disks. Likewise, if a disk fails, its replica is still available to the servers.

This replication scheme provides better availability than Sprite, but it is not simple and it costs more, because it uses extra hardware. Besides the cost of the extra servers and disks, the scheme uses special disk hardware and a high-bandwidth, low-latency network called Autonet [Rodehe91]. Echo disks are equipped with hardware that recognizes at most one server at a time as the disk's owner. This is necessary, because it is possible for a backup server to try to take over for a primary that is merely slow but not actually down. The extra hardware prevents the servers from both trying to access the same disk in this situation. Autonet makes it possible to detect crashed nodes in under a second. This allows a backup server to take over for a primary very quickly. Also, as described below, a low-latency network reduces overhead when a primary server CPU communicates with its backup to update the replicated main-memory cache state.

Like Sprite, Echo clients cache file data for high performance, but Echo's client caching is even more aggressive than Sprite's in at least three ways. First, most of the client workstations dedicate more than 16 megabytes to the cache. Sprite clients generally use about one-fourth of their main memory for the file cache, for an average of seven out of 28 megabytes. Second, Echo clients cache directories as well as files, so clients see very good performance when listing directory contents. Sprite clients do not cache directories. Third, Echo allows clients to cache modified file attributes as well as dirty file data. This speeds up operations that examine the status of files.

To guarantee cache consistency, Echo uses a token scheme with *leases* [Gray89]. A lease is a time-out period associated with a token and agreed upon by both the server and the client. Token managers on Echo file servers grant and revoke tokens to maintain cache consistency in the same way DEcorum token managers do. However, tokens granted by Echo servers are valid only for the duration of the lease; they must be refreshed by the client before the time-out deadline in order for the client to continue to cache the associated file. The advantage of leases is that they simplify and improve handling of network partitions. If a server loses contact with a client, the server waits to

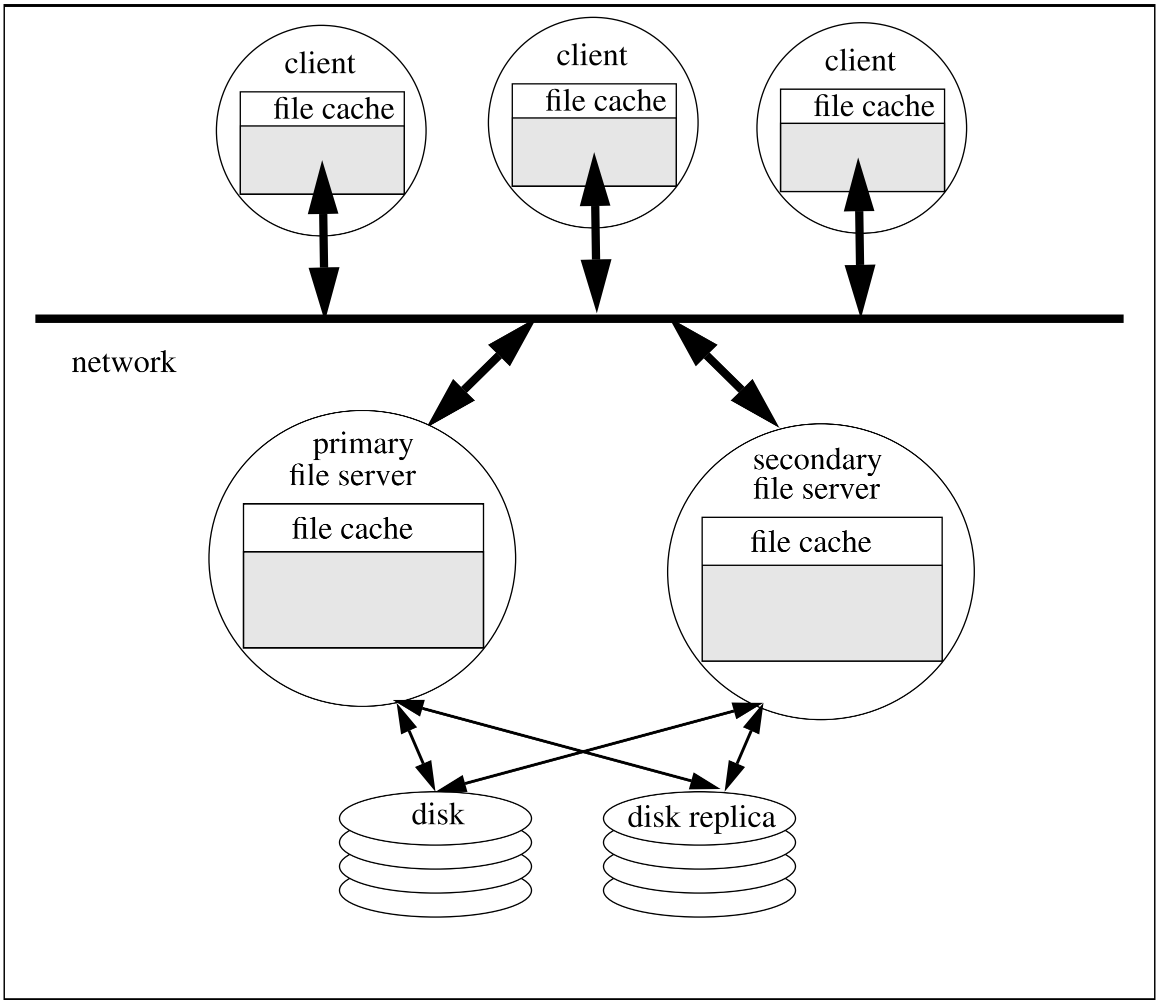

Figure 1-4. The Echo system.

This figure illustrates replication of file servers and disks in the Echo distributed file system. If the primary server fails, the secondary can take over, because it has access to the same disks. If a disk fails, its replica is available to the servers.

revoke the client's tokens until they time-out. The client knows that its tokens are no longer valid if it has been unable to refresh them with the server. Unlike Sprite, this allows Echo to maintain a consistent single-system image across clients even during network partition. Unfortunately, leases offer no further advantages if the client has dirty data in its cache. Any dirty data still on the client during a network partition may have to be discarded when the client's tokens time-out. This is true even if the application that created the dirty data has already exited assuming it wrote the data successfully.

Echo's cache consistency policy differs from Sprite's by allowing more caching during write-sharing. In Sprite, a file undergoing concurrent write-sharing is marked uncacheable. In Echo, at least one client is always allowed to cache a file, even during concurrent write-sharing. Echo makes this increased sharing possible by granting cache tokens for read and write requests, rather than handling cache consistency based only on file open and close requests. If two clients have a file open for concurrent write-sharing, tokens for the file will pass back and forth between them as they each issue read and write requests. This is an advantage unless the tokens pass back and forth frequently, in which case the token approach will generate just as much traffic as Sprite: the caches will be flushed continuously by token recalls, whole cache blocks will be reread many times, and there will be additional overhead for issuing and recalling tokens.

Like Sprite, Echo must recover its distributed cache state after a server failure, but Echo uses server replication to accomplish this. The token information constitutes Echo's distributed cache state. Unlike Sprite, Echo is able to take advantage of its replicated file servers for token recovery. Echo stores the token information in main memory on both the primary server and its backup. Before granting any token request to a client, the primary server sends a message to the backup causing it to update its copy of the token information. Updating the token information in the memory of the backup server is faster than updating it on disk. And because Echo already uses replicated servers for high availability, this is a natural solution for protecting the token information from server failures. When the primary server fails, the token information is available on the backup that takes over the duties of the failed server.

If both servers crash, though, Echo has no way to regain the token information. In this case, Echo users must reboot their workstations as was done originally in Sprite. Echo's designers chose not to implement a recovery mechanism such as Sprite's that would allow the servers to retrieve the token information from the clients. One of the reasons is that Echo authenticates client requests more carefully than Sprite [Mann93]; it would have to authenticate the token information received from clients after a crash. However, Echo's designers believe that a mechanism such as Sprite's would have been useful: "Weighing the advantages and disadvantages, we prefer token replication as the first line of defense against server crashes. However, we encountered enough double server crashes in Echo to convince us that token recovery would have been useful as a second line of defense. It would have considerably reduced the disruption to users in these cases [Mann93]."

1.2. Providing High Availability

This section describes traditional fault-tolerant techniques used to provide very high (non-stop) availability. The categories of techniques are error repair, redundant (replicated) hardware and software, and transactions. For each category, I describe a few systems that use the technique. However, these divisions are somewhat clumsy, because the techniques are not entirely independent and many systems use more than one.

There is a further category of research that attempts to provide high availability through software verification. The idea behind this work is to build systems that avoid crashes altogether, because they are flawless and can be proved so. I do not describe work in this area, because nobody has yet formally verified a distributed system the size and complexity of Sprite.

The goal of this section is to explore the trade-offs between fault tolerance and the fast recovery approach to high availability. If a system must guarantee non-stop availability, then it must use the fault-tolerant techniques described in this section. In particular, it must use some form of redundant hardware and software to provide continuous operation in the event of a failure. However, as described below, these fault-tolerant techniques also have disadvantages. They can be difficult to generalize (error repair), slow (transaction-based systems) or some combination of expensive, complex, and slow (redundant hardware and software). In contrast, the fast recovery approach does not provide continuous operation after a failure. It merely attempts to reduce downtime by recovering quickly. But because the fast recovery approach is simpler and less expensive, environments with less stringent availability requirements may find it more attractive.

1.2.1. Error Repair

To provide high availability, some systems attempt to detect and repair errors on-line. One name for this technique is *error repair.* It is also called *forward error recovery.* Tandem's Integrity S2 [Jewett91] and AT&T's 5ESS (electronic switching system) [Smith81] [Toy92a] are examples of systems that detect their own faults and repair themselves on-line, as described below. The appeal of error repair is that it does not incur the performance overhead of redundant hardware and software. Also, the technique is mandatory in real-time environments that cannot stop processing or back up and redo operations. However, there are several disadvantages of the technique. First, it often requires help from special hardware which adds to the cost of the system. Second, it is hard to generalize to different systems, because it requires ad-hoc recovery code for each different type of error. Finally, if the failing component of the system is unavailable while undergoing error repair, this technique does not provide truly continuous operation. For this last reason, most systems, including those described below, combine error repair with redundant hardware and software.

1.2.1.1. Integrity S2

A goal of Tandem's Integrity S2 is to provide high availability and good performance for a standard UNIX operating system with source-level application portability. Integrity S2 accomplishes this goal with highly redundant and specialized hardware and the ability of portions of the system to locate and fix hardware and software errors. Integrity S2's hardware architecture allows on-line removal of failed components including CPUs. Faults that occur in the processor memory complex are reported to the processors using a high-priority interrupt. The operating system can then execute instructions to repair the system, but it first identifies the faulty component and executes any needed recovery code. Faults in the I/O system or peripheral devices are handled by an I/O fault handling layer.

Integrity S2 provides error repair capabilities in the software by modifying the standard UNIX implementation of internal consistency checks. In a standard UNIX kernel, a detected failure in either the hardware or the kernel software will cause a *panic*, which in turn causes the operating system to cease processing. In contrast, Integrity S2 uses a *forward recovery model*. The kernel still self-checks itself for faults, but instead of allowing the consistency checks to cause a panic, the kernel executes a forward recovery routine specific to the type of fault detected. The recovery routine attempts to fix the problem and whatever damage it may have caused. For example, consistency checks are executed by audit routines to maintain the correctness and consistency of kernel data structures. This approach is not easily generalizable, because for any given system it requires an arbitrary number of consistency checks with special recovery code for each. It can be difficult to determine if all possible errors have been assigned recovery code. It can also be difficult to determine if the recovery code is correct.

1.2.1.2. 5ESS

AT&T's 5ESS [Smith81][Toy92a] [Toy92b] is another example of a system that combines error repair with redundant hardware and software. 5ESS detects and repairs faults in both its hardware and software, and can even make hardware and software upgrades, all without interruption in service. For hardware error repair, the 3B20D processor includes the ability to detect faults and execute self-correcting instruction sequences. The system also logs error rates. If a certain rate of error is exceeded, the system switches to a new component and automatically removes the failing component from service. Specialized hardware and software and a highly redundant hardware architecture make this possible.

The software also includes self-check and repair capabilities. 5ESS uses the UNIX RTR operating system, intended to handle both real-time and time-shared functions. All the components of the software audit their important data structures, either regularly or upon request. If the audits find problems with the data, they can either attempt to correct the data, or they can execute other error recovery mechanisms – sometimes causing initialization of data structures or even removal of hardware components. As a technique for providing high availability, the UNIX RTR software has clearly been effective; the telephone switching system long functioned within its goal of a total of only a few minutes of downtime each year. However, recent interruptions in service show that as the system becomes more complex, it has the same downside as the software for Integrity S2; the system of checks and recovery actions is complicated and hard to verify.

Besides checking and repairing faults, 5ESS uses redundant hardware and software components, in part so that it can upgrade hardware and software without interrupting service. There are three main factors involved in retrofitting a 5ESS switch this way. New hardware and software are added to the system at different times, so new software must be compatible with the old hardware, and new hardware must be compatible with the old software. Second, to prevent loss of service, not all the software modules can be changed at once. Therefore, messages from new software modules to old ones must be compatible with the old modules. Third, the databases within the 5ESS switch (for routing and subscriber information and data about the hardware and peripheral equipment) must evolve from a format compatible with the old software to a format compatible with the new. The method used to accomplish these tasks is called *generic retrofit*. It takes advantage of the redundancy in the switch so that one part can be changed while another still operates. The redundancy is costly, but the alternatives are worse: no upgrades to 5ESS, or loss of service to customers. More detailed descriptions of systems using redundancy for fault tolerance follow in the next section.

1.2.2. Redundancy for Masking Faults

Most systems that provide high availability use some variety of redundancy, in the hardware, the software, or both to allow the system to mask failures. For example, if a system has backup file servers, then the failure of a primary file server is masked if the backup file server can take over for the failed primary.

To provide non-stop processing it is absolutely necessary to use this sort of redundancy, but it does not come for free. Extra hardware costs money and can cause increased architectural complexity and performance degradation. Performance degradation occurs if backup systems must be kept up to date with primary systems. Keeping them up to date requires more message traffic as well as added complexity. If the backup systems are designed to handle a full load when the primary has failed, then there is wasted extra capacity in normal operation. Otherwise the system runs slower when in backup mode.

While the systems described in this section all use redundancy to tolerate faults, they differ in at least three important ways. First, they are designed to tolerate different types of faults. For example, some systems tolerate hardware errors but not software errors. Second, in some systems the existence of replicated resources is transparent to both users and software, while other systems require sophisticated knowledge to write fault-tolerant programs. Third, the systems differ in the extent to which replicated resources are wasted in the absence of failures. For example, some systems use their replicated processors only if a primary processor fails. In other systems, the extra processors run different jobs and therefore contribute to the overall processing power.

In this section I describe Tandem [Bartle81][Bartle90], Stratus [Stratu89][Webber92], TARGON/32 [Borg83][Borg89], Zebra [Hartma93], and ISIS [Birman84][Birman89]. Some other systems using replication are Andrew [Howard88], Cedar [Giffor88], Eden [Pu86], Grapevine [Birrel82], LOCUS [Walker83], and SWALLOW [Reed81]. There is also some work on replicated recoverable processes [Babaog90] on Mach [Accett86].

1.2.2.1. Tandem NonStop Systems

Tandem's NonStop Systems are some of the best-known highly available systems and use hardware and software redundancy to tolerate both hardware and software failures. Their main market is on-line transaction processing. Through a highly redundant architecture, Tandem systems are able to tolerate the failure of any single hardware component.

Tandem's hardware redundancy is designed to help prevent *error propagation*, in which an error causing one system component to fail causes other parts to fail as they take over. For example, Tandem developed a loosely-coupled multiple computer architecture using primary and backup processors. If one processor fails, a backup is ready to take over, but the backup is usually passive and only periodically updated by a checkpoint from the primary. Because the backup processor is updated by checkpoint, rather than by executing exactly what the primary executes, it is less likely to suffer the same error as the primary. Other hardware redundancy in the system includes fully replicated (mirrored) disks to provide full backup of the database, a separate I/O bus for each processor, and dual-ported controllers connected to the I/O busses.

Tandem's software fault tolerance is also designed to avoid error propagation. Tandem uses *process pairs* supported by the Guardian operating system kernel. A process pair functions logically as a single process, but it is composed of a primary process and a backup process. Tandem's process pairs are similar to the primary/backup scheme used in their hardware; the backup process is updated by checkpoint, rather than by executing everything the primary executes, so it is less likely to suffer the same error as the primary. In Tandem's case, the primary process does all the work while the backup remains passive and prepared to take over if the primary suffers a fault. At critical points, the primary sends checkpoint messages to the backup. The backup applies the changes in the checkpoint messages to its own state and data structures. Immediately after a checkpoint, the primary and backup processes are identical. The checkpoint messages usually occur before or after an important I/O or file operation, since these operations change the state of the primary process with respect to the outside system. After a fault in the primary, the backup becomes the new primary process and begins processing at the point of the last checkpoint operation.

Sequence numbers are associated with requests so that if a new primary process duplicates a request sent by the old primary before it died, the request is recognizable as a duplicate.

Tandem's process pair scheme is particularly suited to avoiding failures from *transient* software errors – the most pernicious sort of software error. A transient error is one that occurs infrequently and is hard to reproduce. They are often timing-dependent. For this reason they are also called *nondeterministic* errors. The backup process executes what the primary executes only after a failure, when it begins processing from the point of the last checkpoint. An infrequent timing-dependent error is thus unlikely to strike both the primary and then the backup as it recovers from the last checkpoint. Software bugs that are not transient will likely be debugged and removed before long, but transient problems can be very difficult to find and fix. It is Tandem's opinion that software faults are inevitable and also more common than hardware faults. Therefore it is necessary to guard against transient software faults to provide high availability.

The main problem with process pairs as implemented by Tandem is that they are not easy to use. Choosing when to checkpoint primary process state to the backup requires experience and expertise from the programmer for good performance. A typical checkpoint might indicate that a file has been opened or closed. Depending on the importance, even an update to a file could require a checkpoint, causing significant overhead. Since the checkpoint operations are expensive, it is not reasonable to insert them naively in applications that need to be fast.

Tandem is one of the most successful vendors in the on-line transaction processing market. Depending on how much customers are willing to pay, they can buy as much fault-tolerance as they want from Tandem. Tandem can even replicate entire geographic sites to prevent major disasters from bringing companies to a halt. However, Tandem systems can cost tens of millions of dollars [Tandem90], and it is not easy to write fault-tolerant applications using their process pairs directly. In many computing environments the cost of this approach is too high.

1.2.2.2. TARGON/32

TARGON/32 is a UNIX-based system intended for an on-line transaction processing environment. The goal of the system is to ensure that all executing processes will survive any single hardware failure and some nondeterministic software failures. To achieve this goal TARGON uses redundant and specialized hardware and software, including three-way atomic message delivery and process pairs.

For hardware fault tolerance TARGON uses a special setup of redundant multiprocessors, as illustrated in Figure 1-5. They connect two to 16 *clusters* using a fast replicated bus. Each cluster is actually a shared-memory multiprocessor with three processors. Two of the processors run user processes and most of the system code. One of the processors executes special kernel code for message handling and recovery of backup processes, as described below. The replicated clusters in the TARGON system are not used solely as backups. They are available for productive use in the absence of failures. All peripheral devices are dual-ported so that they remain available even if one cluster fails. Finally, disks can be mirrored to keep data available in the event of a disk crash.

To provide non-stop availability, TARGON uses a process pair scheme similar to Tandem's in many ways. Like Tandem, TARGON's process pairs avoid error propagation and failures due to some transient software failures. Each process pair is composed of an active primary process and its inactive backup process. The primary and backup run on different clusters, to provide fault tol-

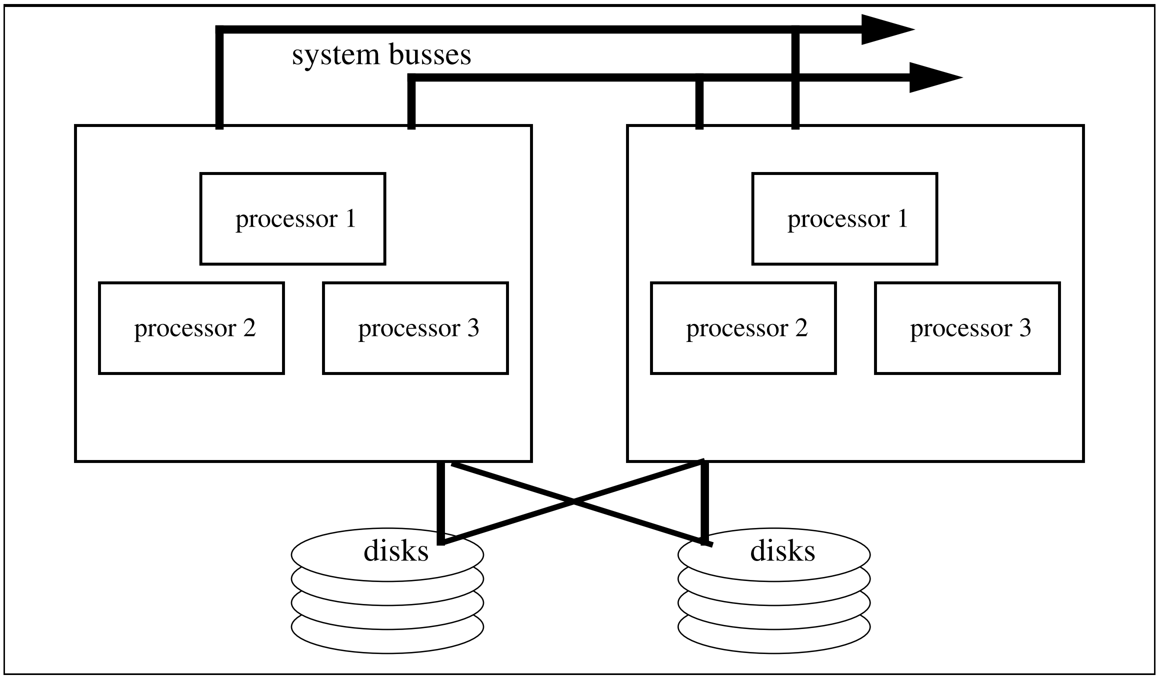

Figure 1-5. TARGON system architecture.

This figure is a simplified illustration of a two-cluster TARGON system. Each cluster is composed of a multi-processor with three processors. A replicated bus connects the clusters. Both clusters can access the disks, which may be mirrored.

erance in the event of any single processor failure. During normal execution, the system logs all input to the primary in a queue for the backup process. Periodically, the state of the primary and backup are synchronized, and the input in the backup's queue is deleted. In the event of a failure, the backup process rolls forward by processing all the input in its queue since the last synchronization point. This brings its state up to date with that of the primary at the time it failed.

TARGON's process pair implementation differs from Tandem's, because TARGON's focuses on ease of use. TARGON's approach thus requires that the use of process pairs be entirely automatic and transparent to users of the system and user-level software. Primary processes in TARGON do not require explicit checkpoints to bring the backups up to date. Instead, the processes are synchronized automatically by the system using a message-based strategy.

TARGON's message-passing software and hardware provides much of the system's fault tolerance. All input to and output from processes is in the form of messages, so the backup's input queue is actually a message queue. For fault tolerance, the backup process must receive all messages received by its primary, but no additional messages. This requires three-way atomic message delivery; whenever one primary process sends a message to another primary process, the message must also be noted by the backup of the sending process and queued at the backup of the destination process. This three-way atomic message system is implemented with special-purpose bus hardware and low-level software in the device driver.

TARGON is an attractive approach to fault-tolerance in the on-line processing environment, but it has three disadvantages for a typical office/engineering environment. First, the replicated and special-purpose hardware makes the system more costly. Second, TARGON's fault-tolerance reduces its performance compared to a standard UNIX system. Third, the system implementation is complex.

Extra communication is a large part of the performance overhead in TARGON. On a single-machine system, without any fault-tolerance, the overhead of communication with server processes is about ten percent, ranging from three to 20% for different system calls. Running on a two-machine system, the overhead for a single process becomes 15%. However, the capacity of the system increases with the additional machine. Adding fault tolerance (backup processes) to the system reduces performance by another ten percent.

One example of implementation complexity in TARGON is that different types of processes require different backup and recovery schemes. In TARGON, the kernel does not survive crashes, so important operating system functions must run as recoverable server processes. However, the implementation of backup processes for user processes is different than that for kernel server processes and peripheral devices. For efficiency, the kernel servers must violate some of the assumptions made of user processes. Even the implementation of servers for block-special (disk-like) devices differs from that of character-special (tty-like) severs. Another example of complexity is that TARGON's process pair scheme requires cooperation between the message system and the paging mechanism. Any changes in the address space of the primary since the last synchronization must be stored so that they are available to the backup process in the event of a failure.

1.2.2.3. Stratus

Stratus also provides hardware and software redundancy for fault tolerance, but it uses *lock-step* synchronization for its replicated processors rather than Tandem's or TARGON's primary/backup scheme. A pair of processors or other boards in a hardware module perform the same actions together, in lock-step. If the self-check on one of the boards fails, then the other board is able to continue functioning with no delay. The advantages of this approach are that hardware errors are detected immediately, and no checkpoints are necessary, since both boards in a pair have the same state at all times. The disadvantage is the cost/performance trade-off. Since the same software runs on the synchronized processors, the processing capabilities of at least one processor are wasted during normal operation of the system. Thus the cost of the system is higher than its processing capacity warrants.

1.2.2.4. Zebra

Zebra is an example of a high-performance network file system that uses replication for availability but wastes very little system capacity. To do this, Zebra uses two techniques from RAID [Patter88]: *striping* for fast file access, and *parity* for high availability of file data. These techniques are illustrated in Figure 1-6. In Zebra, file striping means that different blocks of a file are stored on different storage servers. Striping provides faster access to large files, because different portions of the file can be read simultaneously from the different servers. For example, in the figure the first three blocks of a file are striped across the first three servers. Parity blocks provide high availability of file data, because they make it possible to retrieve the contents of an entire file in the event of any single server failure. In the figure, the fourth server stores the parity block for the first stripe. This parity block is the exclusive-or of the first three blocks. If any server crashes, we can construct the missing block of the stripe from the other blocks combined with the parity block. The next stripe is composed of the next three file blocks along with their parity block. To spread out the load, the parity blocks for the different stripes are located on different servers.

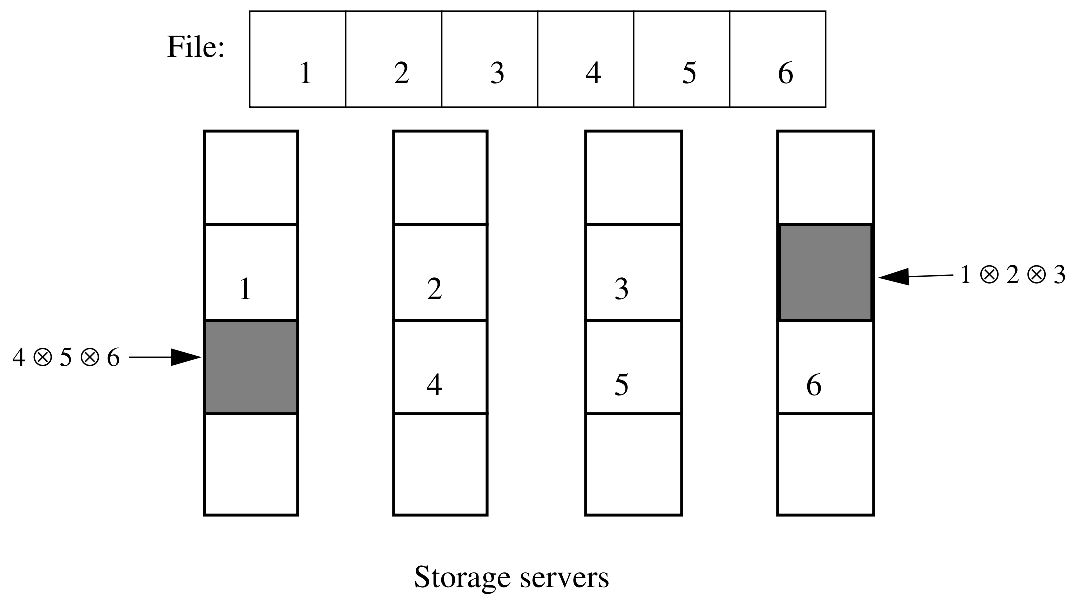

Figure 1-6. File striping and parity blocks in Zebra.

This figure shows how a file with six blocks is striped across four storage servers. The first stripe consists of block numbers one to three with a parity block stored on the fourth server. The parity block is the exclusive-or of the other three blocks in the stripe. The second stripe consists of blocks four to six. To balance the load amongst servers, the parity blocks for different stripes are spread amongst different servers.

With sufficient servers, Zebra wastes very little system capacity. The extra hardware cost is effectively one block's worth of disk space for each data stripe. This extra block holds the stripe's parity. Since the size of a stripe depends on the number of servers in the system, the percentage of lost capacity due to parity blocks decreases as the number of servers increases. In terms of processing capacity, no storage server is wasted during normal processing, since no server stores only parity blocks. However, extra processing is required during a failure to access data or reconstruct file data from the surviving data blocks and their parity.

While Zebra can continue operation despite the failure of a storage server, it does have a single point of failure. One server, called the file manager, must also keep track of which parts of a file are on which storage servers. All lookup operations go through the file manager, and the file manager maintains the file system metadata. Unless it is replicated, the file manager is a single point of failure, because clients will not be able to open new files if it crashes.

Zebra's file manager may be a good target for some of the fast recovery techniques described in this thesis. The faster the file manager restarts, the sooner Zebra can return to service. Fast start-up for the file manager is particularly appropriate, because the Zebra designers believe that it should be possible to store the file manager's metadata on the storage servers, rather than on a local disk on the file manager. Thus, if the file manager breaks, any other server can take over for it quickly, using the metadata on the storage servers.

1.2.2.5. ISIS

The ISIS project provides high-level support for writing fault-tolerant distributed programs by replicating code and data on different machines. ISIS runs on top of the operating system. Its specification language allows programmers to replicate data structures as many times as desired, so programs can survive the failure of an arbitrary number of sites. Programs access the data as if it weren't distributed. A front-end system inserts calls to a run-time system that handles replication, consistency, failure detection, and recovery. To manage synchronization and concurrency, ISIS provides special commit protocols, all-or-nothing broadcast protocols, atomic broadcast protocols (all sites see messages in the same order) and variations with less strict ordering but higher performance. ISIS also has an automatic distributed checkpoint mechanism.

Although many applications have been built with ISIS, the most interesting in the context of this thesis is the Deceit file system [Siegel89]. Deceit is an NFS-compatible distributed file system that provides the user with per-file control over reliability and performance. For example, a user may choose higher reliability with less performance for some critical files, and higher performance with less reliability for often-accessed files.

These capabilities, however, do not come for free. Performance measurements with only one client and an unreplicated server show that many Deceit operations are significantly slower than the corresponding NFS operations. A NULL RPC is 26% slower; an eight-kilobyte read is 19% slower; an eight-kilobyte write is 24% slower; a GETATTR RPC is 20% slower; and a remove is more than three times slower. A few operations, however, are faster: lookup, rename, and mkdir. Replicating the servers to provide availability further decreases performance. An eight-kilobyte write to a server with one replica is about twice as slow as an eight-kilobyte write to an NFS server.

1.2.3. Transactional Systems

The concept of *transactions* makes it easier to implement fault-tolerant systems. This is because transactions make it easier to handle redundancy and failure recovery. Operations executed as transactions are atomic (either all the changes are made or none is made), durable (the changes survive failures), isolated (serializable), and consistent (the transformation is correct) [Gray88]. Transactions ease handling of system redundancy, because they can ensure that the replicated portions remain consistent. For example, a system that uses replicated file servers for availability may update both servers simultaneously as a single transaction. Because transactions are atomic, the state on both servers will remain identical. Transactions also make recovery actions easier to define. For example, a client might perform a set of related file operations as a transaction. If the transaction finishes, the client knows that the server completed all the operations. If the server fails before the transaction finishes, the client can assume that the server executed none of its operations. Thus there is no question about which operations the client must re-execute after the failure.

While transactions make it easier to implement fault-tolerant systems, they do not by themselves provide non-stop availability. Redundancy is still necessary. If some system component fails, transactions cannot provide continuous operation unless another component can take over. Thus fault-tolerant systems that use transactions do so in conjunction with redundancy.

Transactions do have a disadvantage – performance overhead. As a general rule, the less a system must do, the faster it can do it, and transactional systems must do extra work. In order to provide atomicity, every operation performed in a transaction must be undoable in case the transaction fails. An easy way to make operations undoable is to copy the state of an object before performing any transaction on it. If the transaction fails, the original copy can be restored. But copying an object for each transaction can be costly. Another way to make operations undoable is to log changes to an object, and periodically checkpoint the object by applying the log changes to it. This avoids copying the object for every transaction, but requires extra work to maintain the log and perform checkpoints. For this reason, systems such as Sprite that care primarily about performance are not built on top of transactional systems. Instead, these systems craft individual operations to be atomic as necessary.

In this section I describe two systems that use transactions in different ways to provide recoverable file systems. LOCUS [Muelle83][Popek85][Walker83] uses transactions to keep data consistent across replicas, while QuickSilver [Haskin88] uses transactions simply to provide clean recovery of non-replicated resources. While LOCUS and QuickSilver provide transactions as a part of the operating system (*embedded transaction support*), some systems provide transaction services in an environment or language built on top of the operating system. Transactional systems that I do not describe include Camelot [Bruell88], Argus [Liskov88], TABS [Specto85], Cedar [Giffor88], Clouds [Dasgup88], Eden [Pu86], SWALLOW [Reed81], and V [Cherit84]. Tandem and Stratus, described previously, are also transactional systems. Further comparison of embedded transaction systems can be found in [Gray93] and [Seltz93a].

1.2.3.1. LOCUS

LOCUS provides a UNIX-compatible, transparently distributed file system with automatic file replication for high availability. Transparent handling of replicated files is a major goal in LOCUS. Users can choose the desired degree of replication for files. To make file replication transparent, the system automatically propagates updates to a file to all its replicas and maintains the consistency of these replicas. Sets of changes to a file and its replicas are applied atomically, with a *commit* operation to end the transaction. Closing a file automatically commits any changes. Recognizing that updates to a set of files might be related, LOCUS even allows users to specify groups of files amongst which updates should be handled atomically. To make this possible, LOCUS implements *nested transactions*. A nested transaction is a transaction performed as part of a larger transaction. Its results are visible as soon as it completes, before the larger transaction completes, but can be undone if the larger transaction fails.

For the highest availability, LOCUS designers originally chose to make both a file and its replicas available even when they are separated during a network partition. Under these circumstances, the different copies can be concurrently updated, each from within its portion of the partition. Thus the file and its replica could become inconsistent during a partition. When the partition ended, LOCUS attempted to merge the file's conflicting updates transparently. This led to a lot of special-case protocols for merging conflicting copies of different types of files or directories. In a more recent version of the system, the designers have decided to eliminate this complexity by allowing only the primary copy of a file to be modified during a partition. Thus only the changes from the primary copy need to be applied to the replicas to merge partitions.

Besides complexity, a disadvantage of LOCUS is that it is slow [Lai89]. Maintaining consistency between file replicas is one source of performance overhead. The system enforces file consistency guarantees across machines with distributed transactions implemented with a two-phase commit protocol. Thus an update to a file requires messages to and from any hosts storing replicas. Another slow operation in LOCUS is failure recovery, because it is optimized for the unlikely occurrence of network partitions. The LOCUS recovery protocol requires communication between every pair of hosts in the network. Each host communicates with each other host to ensure that the hosts form the largest possible fully-connected partitions.

1.2.3.2. QuickSilver

The designers of QuickSilver took the approach that software and data replication for recoverability requires too much performance overhead, so the QuickSilver operating system uses transactions as its single system-wide recovery paradigm. QuickSilver guarantees that the file system is recoverable to a consistent state after any failure, because all file modifications are made transactionally.

QuickSilver is structured as a small kernel with system services implemented as server processes. Communication between client and server processes is implemented with an IPC (interprocess communication) mechanism with transaction support. The Log Manager provides a general-purpose low-level interface for transactional logging. Using this interface, each server process chooses the type of logging and commit protocol appropriate for its task. Thus, the servers are able to optimize transaction performance as much as possible. After a crash, the servers drive their own recovery from the log.

QuickSilver is probably the transactional file system that has placed the highest priority on performance, but embedded transactions do cause some overhead. For example, a remote one-kilobyte IPC message on QuickSilver is at least twice as slow as one on Sprite, due in part to the transaction support the QuickSilver IPC system must make to higher-level software. Every IPC message belongs to a uniquely identified transaction. The IPC mechanism must keep track of all servers receiving messages belonging to any particular transaction so that QuickSilver's Transaction Manager can include them in the transaction's commit protocol.

1.3. Recovery Using State Information Stored in Main Memory

Traditionally, the contents of main memory are not considered reliable after a crash. Even if the memory is battery-backed (non-volatile) so that its contents are preserved across power outages, many systems do not consider the data trust-worthy after a failure. This is because the failure itself may in some way have disturbed the data.

Instead, systems use magnetic disk or tape or other stable storage device to store data that must be preserved across failures. While a failure can also corrupt data stored on these devices, systems designers have accumulated techniques to help prevent, detect and fix such corruption. Unfortunately, accessing data on these devices is orders of magnitude slower than accessing data in main memory. Preserving and retrieving data would be much faster if we could safely store it in main memory.

This thesis develops a method to treat main memory as stable storage. The idea is to structure the contents of an area of main memory (called the *recovery box*) so that we can avoid corruption of its contents. The recovery box also allows us to detect any memory corruption after a failure. If corruption occurs, we must discard the recovery box contents, but hardware write protection and other techniques described in chapter 5 make this corruption unlikely. If the contents are okay, we have very fast access to the preserved data after a failure. There is also less performance overhead to maintain the data in main memory during normal system processing than to maintain it on disk. Chapter 5 explains how Sprite uses the recovery box to preserve distributed state information across file server crashes.

The largest body of work similar to the recovery box consists of the many examples of main-memory databases or database models [Birrel87][DeWitt84][Garcia92][Hagman86][Salem86][Stoneb87]. However, most of these systems eventually push the database log to stable disk storage and recover only the tail of the log from memory, which is usually non-volatile. This enables the database to recover a consistent copy of the database even if the tail of the log becomes corrupted. For example, the original design of the POSTGRES storage system assumed the existence of some non-volatile main memory that could be implemented with battery backup and error correction techniques. The design proposed storing the tail of the transaction log in the non-volatile memory. However, the actual implementation of POSTGRES pushes the log and data pages to disk on every transaction commit [Sulli93b].

Another similar technique is Recoverable Virtual Memory (RVM) [Satyan93]. RVM is a very lightweight transactional facility designed for UNIX applications with persistent data structures whose updates must be fault-tolerant. To avoid severe performance overhead, RVM values simplicity over generality; for instance, it does not provide nested or distributed transactions, but it does provide permanence in the event of power failures by pushing log changes to disk. The recovery box takes an even more extreme approach. The recovery box does not by itself protect data from power failures, so it does not need to write data to disk. This makes it very lightweight, but it also limits its suitability to applications that can occasionally suffer failures.

Despite this list of similar techniques, there seem to be few systems that depend primarily on main memory for stable storage of any of their data, and in particular, very few file systems that do this. Besides Sprite's use of the recovery box, I am aware of only two other file systems that have considered this technique: the Phoenix In-Memory File System [Gait90], and the Harp file system [Liskov91]. However, Harp currently does not actually implement the technique [Johnso93]. I describe these two systems in this section.

1.3.1. Phoenix

Phoenix is an in-memory file system intended mainly for diskless computers with battery-powered memory. To protect the file system, Phoenix uses a copy-on-write and checkpoint scheme. The system starts with a time-stamped and write-protected copy of the file system called the reserve file system. As pages of the file system change, the new pages are added to a new version of the file system, leaving the reserve copy unchanged. At some interval the system performs a checkpoint, garbage collecting the replaced pages in the reserve file system. Then the process continues with the newly time-stamped version used as the reserve file system. With this use of memory, Phoenix is probably not appropriate for large highly-active file systems. Its designer, however, feels it might be appropriate as a cache for large high-performance file systems.

There is one big difference between Phoenix's use of main memory and the recovery box's use. After a crash Phoenix may lose all changes to the file system since the last checkpoint. While the memory containing the changes may be non-volatile, there seem to be no checksums or other techniques to ensure its contents are uncorrupted. Checksums may be too expensive to protect the Phoenix file system from corruption.

1.3.2. Harp

Harp is relevant to this thesis for at least two reasons: it stores state information in main memory, and it uses replicated file servers for high availability. Harp is a replicated UNIX file system accessed via NFS. Each primary Harp file server has a backup file server capable of serving the same files. The backup can also be a primary file server for a different set of files, to avoid wasting capacity in the system. Clients send their requests to a primary file server. The primary server logs any resulting file system changes to its main memory. It later applies these changes to the file system on disk asynchronously. Before responding to the client, the primary server also forwards the request information to the backup server, which logs the file system modifications in its own main memory. The backup then responds to the primary before the primary responds to the client. Updating this information in the main memory of the backup server is faster than synchronously updating the file system on disk. In this way, Harp replicates its file system change log in the memory of a backup machine in the same way Echo replicates its token information.

Harp designers considered using main memory to store information to help the system recover after failures that cause both the primary and backup server to crash. If both servers crash, Harp loses the file system changes logged only in main memory. To handle multiple server crashes Harp considered modifying UNIX so that a portion of its volatile memory would survive a soft crash. They could then store some system state, such as the log and information about the commit state, in this part of memory and use it to restart the system quickly. The designers have so far not implemented the technique, because they do a complete reboot after a crash, and their computers (a collection of MicroVax 3500's) perform a memory check that overwrites the contents of memory [Johnso93].

1.4. Summary

An important part of recovery in distributed file systems is the recovery of distributed cache state information. Many modern distributed file systems cache file data on client workstations and provide cache consistency by maintaining distributed cache state information the file server.

Recovering this state information quickly after a file server failure is part of the challenge for fast crash recovery and is the subject of most of this dissertation.

Fast crash recovery is an availability technique intended for environments that require low cost and high performance and that benefit only secondarily from high availability. The fast recovery approach is not fault-tolerant. It does not mask or repair faults. It merely recovers from them quickly enough to provide good availability.

In contrast, traditional systems provide high availability through the use of fault-tolerant techniques – techniques designed to allow the system to operate correctly and continuously despite the presence of faults. These techniques are necessary for non-stop operation, but they can be expensive, slow and complex. Most of these systems use replicated hardware and software that enables them to mask faults. Other systems also use error repair to fix the damage associated with faults and continue operating. Some systems make all file system modifications transactionally. This ensures that their file systems are recoverable to a consistent state. Figure 1-7 qualitatively illustrates these differences between the fault-tolerant and fast recovery approaches.

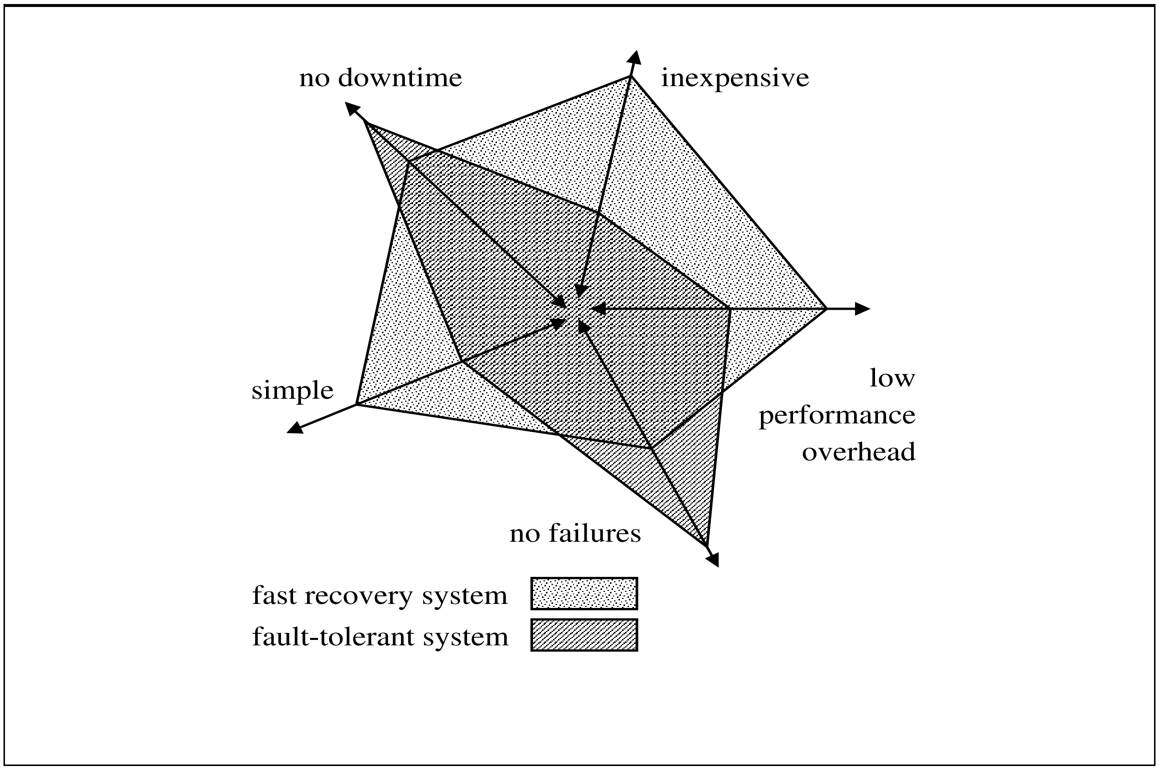

Figure 1-7. Comparison of fault-tolerant and fast recovery approaches.

This figure qualitatively illustrates the difference between the fault-tolerant and fast recovery approaches to providing high availability. The figure is a Kiviat graph. There are five axes, each showing the value of a system parameter. For example, the upper axis shows the cost of the system. A point far away from the center of the figure indicates low cost, while a point towards the center indicates high cost. On all axes, a point farther away from the center is better. The fast recovery approach meets the axes at points indicating low cost, simplicity, low performance overhead, fairly fast recovery, and some number of failures. The fault tolerant approach guarantees the fastest recovery time and no complete failures, since it masks faults entirely. However, it is more expensive due to its replicated hardware, it is more complex, and it has higher performance overhead.

3 Client-Driven Recovery

This chapter describes client-driven recovery of distributed cache and file system state. In client-driven recovery, clients detect file server failures and initiate recovery with the server when it reboots. To recover, the clients send the server their file system state, including information about which files they have cached locally. From this information, the server rebuilds the distributed cache state that it uses to guarantee cache consistency.

The advantage of client-driven recovery is that it is the most natural and least complex form of state recovery in Sprite. The clients control recovery and request recovery of the file server in the same way they request all other services. Client-driven recovery thus requires the least amount of special-purpose code of all the techniques described in this thesis. This reduces the probability that the recovery code is itself incorrect and likely to cause a failure.

However, allowing the clients to drive recovery also has several disadvantages, including a tendency to cause resource contention on the file server. For example, as the Sprite system grew to include more client workstations, the increased amount of recovery traffic caused server congestion and instability during recovery. Recovery of the distributed cache state sometimes required ten to fifteen minutes to complete, and occasionally failed to finish until users rebooted many of the client workstations.

Fortunately, the server congestion problem is solvable. With the congestion control mechanisms and other performance improvements described in this chapter, it now takes an average of 21 seconds to recover the cache state on a SPARCstation-2 server in a test network with ten clients. There is a lot of variance in this recovery time, because as described later in this chapter, ten clients may easily take 30 seconds to recover. Many more clients may be added to the system without increasing recovery time beyond 30 seconds, because the server spends over 90% of the recovery time idle, waiting for clients to send it their state to recover.

Other problems with client-driven recovery are cache consistency violations after a server crash and limited recovery speed. Unlike the congestion problems, these problems cannot be solved in the client-driven recovery approach. The cache consistency violations arise, because the server does not have control over when recovery finishes and other client requests begin. Client-driven recovery is slower than the other techniques, because the server must wait for clients to initiate recovery. These problems are addressed by server-driven recovery, which is described in the next chapter.

This chapter is organized as follows. It first explains the process of client-driven recovery and then details the amount and type of state recovered by clients. Then it lists the problems that cause congestion on the file server during recovery, and how to solve them. Finally, it describes the problems that can be solved only by switching to server-driven or transparent recovery: cache consistency violations after a server failure and limited recovery speed. Brent Welch designed and implemented the original version of client-driven recovery, but the analysis of its problems and the solutions presented here are my own work.

3.1. How Client-Driven Recovery Works

While Sprite's distributed caches create a crash recovery problem, their distributed nature is also the solution to this problem. This observation was made by Brent Welch and Mike Nelson: although the server loses its distributed cache state information after a crash, each client retains its own information about which files it has open or cached. To regenerate the information on the server, it is necessary only to gather that information from the clients. In client-driven recovery the clients detect that the server has crashed and rebooted, and they send the server their cache and file system state via an idempotent recovery protocol. From this state, the server reconstructs the volatile data structures it uses to guarantee cache consistency and to control access to its resources.

The following sections describe this mechanism in more detail. The first section explains how clients detect server crashes. The next section describes how clients recover with the server. The third section describes recovery on the server side, namely, how the server uses the state information it receives from clients.

3.1.1. Detecting Server Crashes and Reboots

To initiate recovery with a server that has crashed and rebooted, the clients must be able to detect the crash and reboot. Sprite clients do this using a low-level mechanism that monitors message traffic. Hosts communicate with each other in Sprite using kernel-to-kernel remote procedure calls (RPCs) [Welch86]. A client issues a request of the file server by sending it an RPC. One of several RPC server processes in the server's kernel handles the request. The server then sends the results of the request back to the client in a response to the RPC. The clients and server keep track of hosts from which they've received an RPC request or reply. If they've received a message from a host recently, they label the host as being alive. But if an RPC to a server *times out* (receives no response), the client considers the server to have crashed. The RPC that times out will *hang*, waiting for the server to become available again. To detect when the server returns to service, the client periodically *ping*s the dead server. A ping is a simple RPC that requests nothing more than an acknowledgment from the receiving host. When the client finally receives an acknowledgment from a server that crashed, it means the server has rebooted and is available again. The clients can then recover with it.

To detect server reboots, clients also monitor the server's *boot ID*. A host changes its boot ID whenever it reboots, and it includes this boot ID in the header of every RPC request or reply it issues. If a server crashes and reboots between RPCs from a client, the client will detect a change in the bootID during its next RPC. With the improvements in recovery speed described in this thesis, it is possible for a server to crash and reboot quickly enough that a client would not otherwise detect the event with a hung RPC.

The boot ID also helps distinguish between a true crash and a network partition. In a network partition, the server remains operational but is unreachable due to network problems. If a client marks a server as dead and later receives an acknowledgment to a ping, it checks the boot ID in the server's acknowledgment. If the boot ID has changed, the client knows that the server truly crashed and rebooted. If the boot ID has not changed, then the server's inability to communicate was due to a network partition.

Unfortunately, differentiating between crashes and network partitions is not very useful in the context of Sprite's state recovery. Even if the communication lapse was due to a network partition and not a crash, the client must still go through recovery with the server. This is because servers also monitor the state of clients. If the server cannot communicate with a client, it assumes the client has crashed and it garbage collects the client's state information. When the client and server are again able to communicate, the client must assume that the server has cleaned up its state information. The client therefore treats the network partition as if the server crashed and rebooted, so the client must send the server its state information again.

3.1.2. Client State Recovery

When a client detects that a server has crashed and rebooted, it initiates recovery action with that server. This section describes the first of the two basic steps a client goes through to recover from a server crash. In the first step, the client sends its cache state and other file system state to the server. This is called *recovering* the state information. This section details the amount and type of state the clients recover, and explains how the clients recover it. The second client recovery step, restarting hung RPCs, is described in the next section.

There are three basic types of state information, and clients recover these in separate phases: *prefixes* first, then *I/O handles*, and finally *streams*, as described below. I provide the most detail for recovery of I/O handles for files. This is because files are the only cacheable objects in Sprite. File I/O handles thus form the distributed cache state that accounts for the bulk of file system state to recover.

Clients first recover their prefixes. A prefix is a reference to the root of a file system, and it contains information about that file system domain. A client cannot access an object in a file system without first obtaining a prefix handle to that file system. Thus prefixes must be recovered before other types of information.

Clients next recover their I/O handles. I/O handles store information about objects in the file system, such as files and directories, devices, pipes, and *pseudo device*s. (A pseudo-device is a named object in the file system that is actually implemented by a user-level software process [Welch88].) An I/O handle is similar to a UNIX inode, but also includes information such as the number of open references (*streams*) that the client has for an object, and whether the streams are open for reading or writing. For files, it includes other cache state information such as whether the client has cached pages from the file. Some file I/O handles reference directories, because Sprite implements directories as uncacheable files. An object does not have to be open to have an I/O handle. For instance, clients will have file I/O handles for files they have closed, but for which they still have cached pages.

Finally, clients recover their streams (sometimes called stream handles). These are the open references to objects in the file system and are similar to UNIX open file table entries. A stream keeps state information about the open reference, such as the offset into a file for the next read or write operation. Each stream also points to a corresponding I/O handle for the object it references. Streams are reopened last, because the server must already have information about the I/O handles referenced by the streams.

To recover a prefix, I/O handle, or stream, a client *reopens* the object by issuing an RPC to the file server. The reopen RPC contains information the server needs to generate its own set of I/O handles and streams.These I/O handles and streams form the server's distributed file system state.

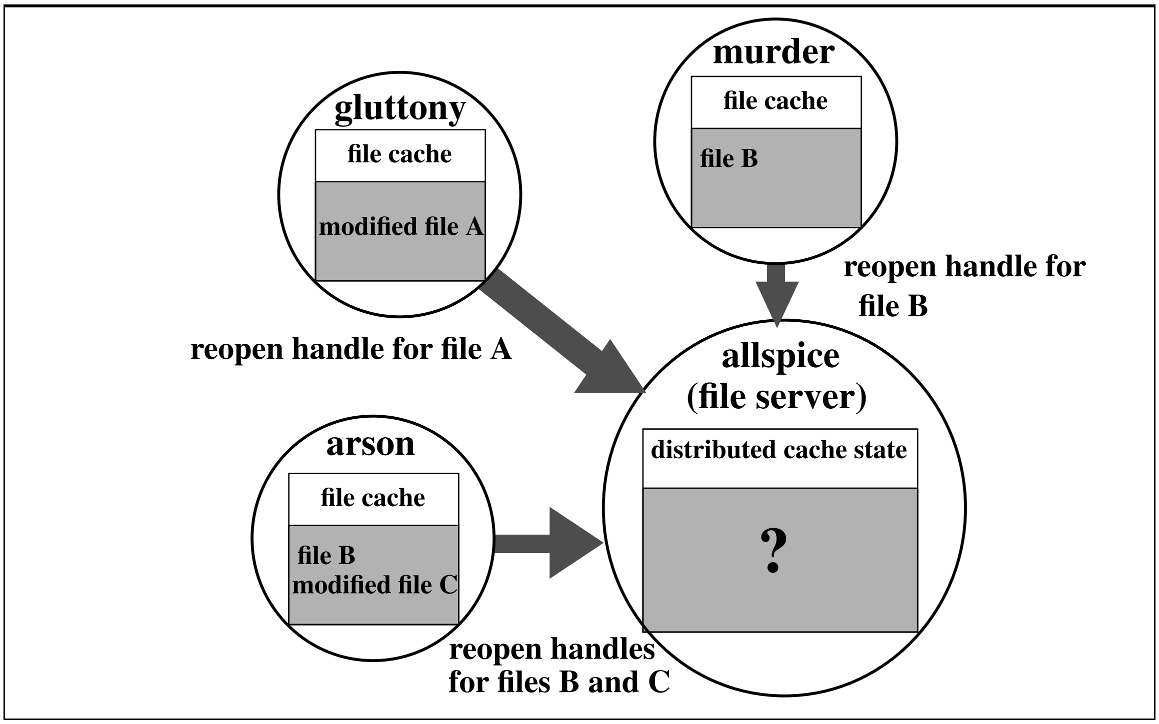

Figure 3-1. Client-driven recovery in Sprite.

This figure illustrates how the Sprite clients refresh the server's distributed cache state information after a server failure. After a crash, the server has lost this information, because the information was stored in its main memory. However, each client has information about which files it has open or cached. The clients reopen these files on the server, and the server uses the information in the reopen requests to rebuild its distributed cache state.

Figure 3-1 and Figure 3-2 illustrate the client recovery process for files. Figure 3-1 shows the clients sending the server their recovery information, and Figure 3-2 shows the server once again in normal operation, with its distributed cache state complete. A client indicates that it has finished reopening its file system state by sending a final *end recovery* RPC to the server.

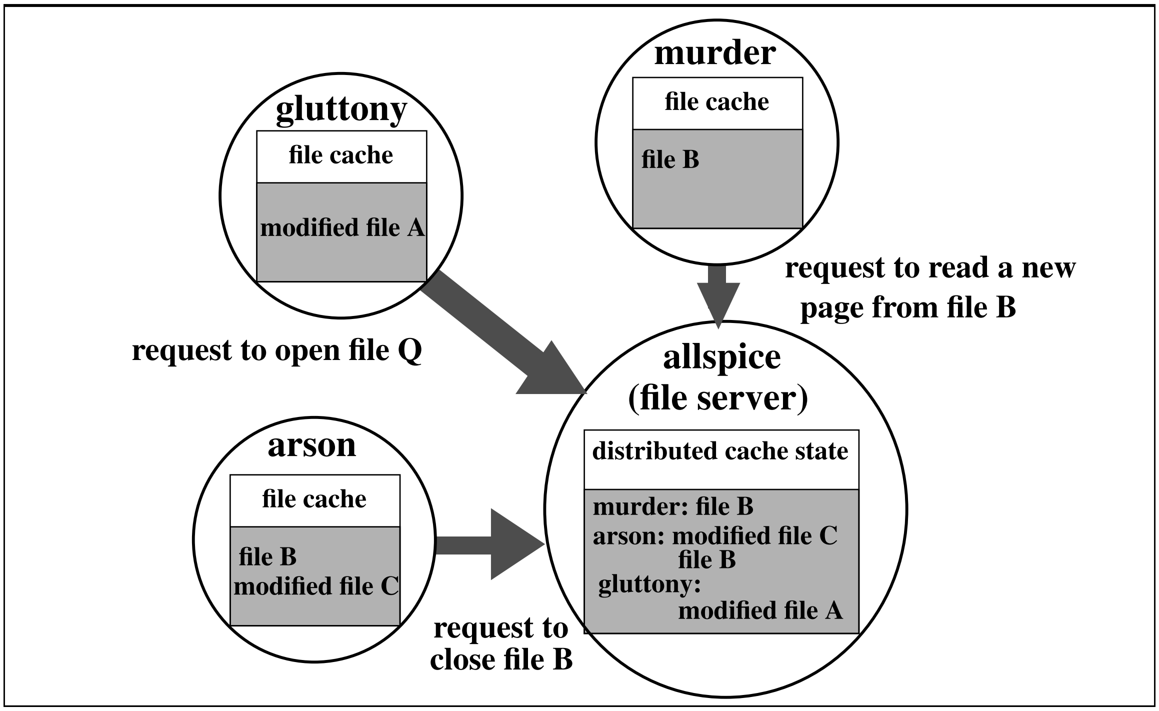

Figure 3-2. The system after recovery.

This figure shows the Sprite system after recovery. It is now in normal operation again, because the server has reconstructed its distributed cache state.

Table 3-1 shows the state information a client sends to the server in its reopen requests for different types of file system objects. In the context of this dissertation, the file I/O handles are the most important objects reopened, since they form the server's distributed cache state. Recovery for the other types of I/O handles is straight-forward. The state sent to the server for file I/O handles includes five things. The file ID is a unique identifier for the file object on disk. The prefix ID is the file ID for the file system storing the file. The use structure gives the number of open reference streams to the file, sometimes called *use counts* or *reference counts*. This includes a count of how many references are for writing or executing the file. The flags field indicates whether there are dirty blocks for the file in the client's cache. The version number of a file is updated every time the file is opened for writing. The server stores a copy of the version number on disk. During recovery, the server compares the version number from the client to the one on its disk to detect whether the file has changed. If so, the server can respond to the client telling it to invalidate its cached copy.

3.1.3. Restarting Hung Client Processes

After recovering its file system state with a failed file server, a client executes its second recovery step: awakening all processes that are hung waiting for an RPC to complete with the server. When a server crashes, all client RPCs in progress with the server will hang until the server is again available. When the client detects that the server is again available, it recovers is state infor-

Type	State information	Purpose
Prefix	prefix name	Server checks if it still services this domain
File I/O handle	file ID	Unique ID for file
	prefix ID	Unique ID for file system
	use	Number of read/write/execute references
	flags	Whether there are cached dirty blocks
	version number	Identifies version of the file
Device I/O handle	file ID	Unique ID for device object in file system
	use	Number of read/write/execute references
Pipe I/O handle	file ID	Unique ID for pipe
	use	Number of read/write references
Pseudo-device I/O handle	file ID	Unique ID of control handle
	server ID	ID of host running the pseudo-device code
	seed	Used to create the unique file ID
Stream	stream ID	Unique ID for stream
	I/O handle ID	file ID of I/O handle stream references
	flags	Flags passed from open request
	offset	File access position for stream

Table 3-1. State information sent from clients to the server for recovery.

For each type of handle, this table gives the information sent by clients to the server for crash recovery. Important fields are described in the text. Pseudo-devices are devices implemented by user-level processes. Please see [Welch88] for further details.

mation and then wakes up its hung processes. The processes automatically resend their RPCs to the server.

3.1.4. File System State Recovery on the Server

So far this chapter has described the client side of state recovery; we now look at the server's side. Its two basic activities are processing the cache state sent to it by clients and retrieving further information from disk.

To regenerate most of its file system state, the server processes the state information it receives from clients. When the server reboots, it turns on its RPC service and is then available for clients to send it their state. For each I/O handle that a client reopens, the server creates a corresponding I/O handle that records the client's use of a file or other object. Likewise, for each stream a client reopens, the server creates a corresponding stream that records the client's open reference to a file system object.

In particular, the purpose of the server's file I/O handles is to maintain cache consistency among the clients. The server must know which clients have open references or cached dirty blocks for files. During recovery, the server builds a client list for every file that is reopened. The server adds to this list each client that indicates it has an open reference or cached blocks for the file.

Although the server receives its cache state information from the clients, it also needs information from the *file descriptor* for each reopened I/O handle. The file descriptor includes an objects's last access and modify times, size, access permissions, and other data that the server needs before clients can access the object. Clients do not send descriptor information to the server, because they do not necessarily have the most recent copy of it. The descriptors are stored on the server's disk. The server also caches copies of these file descriptors in main memory, to avoid going to disk every time it refers to a descriptor. During recovery the server reads the file descriptor for each reopened I/O handle from disk into memory. If more than one client reopens the object, the server only performs the disk read for the first client and uses the cached descriptor thereafter.

Fortunately, the server can overlap these disk I/O operations with the processing necessary for reopening other handles. The server is able to do this because reopen requests from different clients can be handled by different RPC server processes. When one RPC server process blocks for I/O, another can perform the processing necessary for its reopen request.

3.2. Client Cache and File System State

This section gives more detail about the cache and file system state maintained by clients. The type and amount of state information determines the amount of work performed during recovery. Understanding what is recovered enables us to create realistic recovery benchmarks. In this section I show three things. First, file I/O handles and streams to files account for most of the client state. Second, the clients only need to tell the server about a small part of their state. Third, the server does not need to perform descriptor reads for all of the file I/O handles reopened, because about a third of the file handles are to files shared on more than one client. The measurements in this section were made on Sprite's main Sun-4/280 file server. The data was gathered on four separate days, spanning a month, across all available Sprite clients, for a total of 50 state measurements, each of a single client.

Table 3-2 demonstrates that file I/O handles and streams to files together are responsible for the majority of state to recover. The table shows the average number of file I/O handles, streams to files, and other file system objects served by Sprite's main file server per client. File I/O handles and streams to files constitute about 97% of all handles. For this reason, the benchmarks in this thesis use only I/O handles and streams for files.

Table 3-3 shows that clients only need to reopen a small part of their total state information. The table gives the average number of files per client that are neither open nor have dirty cache blocks to write back to the server. A client does not need to recover these files. Eighty-eight percent of file I/O handles fall into this category, thus a client only needs to reopen about 12% of its file I/O handles. Counting streams and other I/O handles as well, this means a client only needs to reopen about one quarter of the state it maintains.

Table 3-3 also shows the number of files per client with dirty cache blocks. In any given snapshot of client state, a client is unlikely to have dirty blocks for many files in its cache. On average, there are no dirty files per client. This information is useful for constructing realistic benchmarks.

Handle type	Average number per client	Percent of total
File I/O handles	636 (68-2103)	85.5
File streams	77 (32-135)	10.3
Other handles and streams	31 (14-80)	4.2
Total	744	100.0

Table 3-2. Amount of state information on clients.

This table shows the average number of handles and streams referencing objects on Sprite's main Sun-4/280 file server for each client. The data was gathered on four separate days spanning a month across all available workstations, excluding the server, giving 50 measurements, each of a single client. Numbers in parentheses give the minimum and maximum values recorded.

Status of file I/O handle	Average number	Percent of file I/O handles	Percent of total state
Unopen file I/O handles with no dirty blocks	560 (36-1968)	88.0	75.3
File I/O handles with dirty blocks	0 (0-2)	0.0	0.0
Unopen file I/O handles with clean blocks	362 (6-1702)	57.0	48.7

Table 3-3. Status of file I/O handles on Sprite clients

This table gives the status of the file I/O handles on clients referencing files on Sprite's main Sun-4/280 file server. The first category shows files that are neither open nor have dirty cached blocks on the client. The second category shows file I/O handles for which the client does have dirty cache blocks. The last category shows closed files for which the client has clean cached blocks. Eliminating the recovery of the first category of file I/O handles reduces the number of file I/O handles recovered by 88%, and the total state recovered by more than three-quarters. Numbers in parentheses show the minimum and maximum values recorded.

However, if a server is down for a long time, more clients will accumulate dirty blocks that they need to write back to the server.

Note that it is not necessary for a client to reopen files that are closed, but for which it still has clean cached blocks. The last entry in Table 3-3 shows the number of files per client in this category. The server does not need to know about these files to maintain cache consistency, because the client will find out of its own accord the next time it opens such a file whether or not its cache blocks are still valid. It does so by comparing the new file version number against the one it has saved in the file I/O handle. But to save the file's version number, the client must keep around the

Category	References (%)	Minimum (%)	Maximum (%)
References to open shared files	37.1	35.0	56.0
References to unopen shared files	44.4	33.2	51.9

Table 3-4. Amount of file sharing on clients.

This table shows the amount of file sharing amongst the clients of Sprite's main Sun-4/280 file server. This includes files that are shared for reading or for writing. *References to open shared files* gives the percentage of file streams for files that are open on more than one client. *References to unopen shared files* gives the percentage of unopen file I/O handles on a client that are referenced by more than one client. The data was gathered on four separate days, spanning a month, across all available workstations, excluding the server. The percentage is an average across all the days' data. The last two columns pro-

file I/O handle, even if the file is closed. The client can only garbage collect file I/O handles for files that are closed and have no blocks left in the cache.

There is one other factor that determines how much work is performed during file I/O handle recovery, and that is the degree of file sharing between clients. The server performs setup work for a file the first time it is reopened, and it can skip this work when other clients reopen the same file. Reading a handle's file descriptor is the most costly example of such setup work. Table 3-4 lists the amount of file sharing (both read- and write-sharing) amongst clients for files on Sprite's main file server. Over a third of references to open files are to files that are shared by another client. This means that the server will not need to perform a descriptor read for at least a third of the file I/O handles it reopens.

The table also includes this information for files that are not currently open on a client, but for which the client still has handles because it referenced the files in the past. (The files may be open on other clients, however.) Over 40% of these unopen references are to files referenced by more than one client. I include this information for use with later measurements; as explained later, clients in the past recovered even these unopen handles.

3.3. Measurement Setup

This section describes the test environments and methodology used for the measurements presented in this and the following chapters. There are three setups, because Sprite has changed over time, and some earlier measurements cannot be reproduced easily on the current Sprite system. I call the setups *old Sun-3*, *production system*, and *testbed*.

Several of the measurements in this chapter are from an old Sun-3 setup. This was a past Sprite system that used a Sun-3 with about five gigabytes of disk space for the main file server. There were about 40 clients consisting of Sun-3 and DECstation 3100 workstations.

The production system setup is an updated Sprite system. In this system, both the server and clients have been replaced with faster processors. The server is a Sun-4/280 with about six gigabytes of disk space. The clients consist of SPARCstation-1's, SPARCstation-2's, and DECstation 5000's. Although the system now has only about 20 clients, most of the measurements in this dissertation were taken when the production system had 40 clients. As necessary, I indicate the number of clients with the measurement results. The previous measurements of per-client state information in this chapter and a few measurements in later chapters are from this setup.

Most of the measurements in this thesis are from a testbed with its own server. This testbed makes recovery experiments much less painful for our users, since my experiments require crashing and rebooting a server. Also, I can control the resources on the server and the state on the clients, so the testbed provides a much more controlled measurement environment. The testbed includes a SPARCstation-2 file server (40 megahertz, 20 integer SPECmarks) with a WREN IV disk (29.5 millisecond average access delay, 1.3 Megabyte/second maximum transfer rate) and ten clients consisting of SPARCstation-1's, SPARCstation-2's and DECstation 5000's. The testbed is actually a part of the production Sprite system, because all the hosts are on the same network and use the root file server from the production system. But for experiments using the testbed, the clients only recover state from the testbed server, because it is the only server I rebooted.

Most of the measurements from the testbed system, including those in later chapters, use what I call the *basic* state setup. When measurements in the testbed use a variation of the basic state setup, I describe the differences explicitly. Table 3-5 gives more detail about the file system state for this setup. I chose the number and types of streams and file handles, and the degree of sharing and other file system activities, with the following goals in mind. The setup should be easy to control, realistic, and at least as aggressive per client as the averages reported in Table 3-2, Table 3-3, and Table 3-4. Each client has state for 790 file I/O handles. Ninety of the files are open, and one-third of these open file references are to files shared by other clients. The rest of the files are unopened and have no dirty cache blocks that need to be recovered. Of these unopen files, about 40% are file I/O handles also shared by another client. In addition, two clients each have the same file open for writing. The clients truncate and rewrite the file every two seconds. This helps to test for cache inconsistencies during recovery. To simulate the on-going activity of file system clients, one client repeatedly creates and unlinks a file every second. The create and unlink requests will hang when the server fails, but will start up again as soon as the client recovers with the server. Finally, all the clients have an I/O handle for the prefix of the file system served by the testbed server.

For most of the timings in this thesis, I provide only five or six data points. There are three reasons why this is a sufficient number of tests. First, the variation in the results is small – only a few percent. Second, I am not measuring truly random processes. The experiments are controlled, so that the results should be the same for each repeated experiment, within some positive perturbation. The error in the results is always positive, because the tests cannot run any faster than their minimum value. By averaging the results of the tests, I report how long one should expect crash recovery to take, and not the minimum amount of time it could possibly take. Third, I measure how the results scale with different numbers of clients and different amounts of state information. This means that the five or six data points from each experiment are actually a part of a continuum of other data points that support the results.

Handle State	For each client	Additional for one client	Additional for two clients
Unshared unopen file I/O handles	420	0	0
Shared unopen file I/O handles	280	0	0
Unshared file I/O handles open for reading	60	0	0
Shared file I/O handles open for reading	30	0	0
File I/O handles open for writing (shared)	0	0	1
Associated stream handles	90	0	1
Files created and unlinked	0	1	0
Prefix handle	1	0	0

Table 3-5. Basic testbed recovery state setup.

This table shows the per-client experimental setup for the basic testbed measurements. The first column indicates a category of state. The second column gives the amount of state in that category per client. The third column indicates if any single client has extra state in that category. The last column indicates if two clients have extra state in that category.

3.4. Sources of and Solutions to File Server Contention

This section now turns to problems with client-driven recovery and some solutions to those problems. Recovery is one of the most stressful of a system's activities. It places a very high load on the system and often requires executing special-purpose code that is never exercised during normal system operation. It is hardly surprising then that many system bugs, design limitations, and other flaws are likely to be discovered during crash recovery.

There are three general problems for client-driven recovery: contention for resources on the server, cache consistency violations, and limited recovery speed. This chapter includes solutions for the contention problems and several recovery protocol optimizations. These improvements reduce recovery times from as much as ten to fifteen minutes, to about 20 to 30 seconds. However, fixing the cache consistency violations and further improving recovery speed require switching to server-driven or transparent recovery, the subjects of following chapters.

3.4.1. Server Contention

As the Sprite network grew from fewer than ten client workstations to more than 40, recovery was the first part of the system that ceased to work. The increased recovery load on the server exposed a flaw in Sprite's RPC system: a lack of contention control that caused clients to re-initiate recovery unnecessarily. Congestion control is a problem during any period of high load, but in Sprite it first surfaced during client-driven recovery. This section describes the problem.

The contention problem arose because the server has limited processing power and the clients did not reduce their demands as the server became overloaded. As a result, the server could not process all the reopen requests from clients in a timely manner. Some client reopen requests arrived at the server and were dropped in its low-level network interrupt routine, because there were no more available RPC server processes to pick up the requests. Other requests might be picked up by server processes, but they could not be handled quickly, because the server was already processing as much as it could. Creating new RPC server processes to absorb the extra reopen requests did not help beyond a certain point, because the extra processing load just made the server even slower.

These dropped RPC requests caused the system to become unstable. Client workstations that received no response or acknowledgment of their request within the RPC time-out period assumed that the server had crashed again and therefore lost their distributed cache state. These clients then reopened their file system state all over again. Under these circumstances, recovery of file system state often took ten or fifteen minutes in the old Sun-3 system. In some cases, recovery did not finish at all. We had no choice but to reboot some of the client workstations manually to reduce the load on the server so that the other workstations could recover. I call this sort of recovery session a *recovery storm*, as represented in Figure 3-3.

Figure 3-4 shows the unstable behavior of the system in a particular recovery storm. To begin with, clients are all in the *unstarted* state, waiting for the server to become available. When a client detects that the server is available, it enters the *recovering* state, during which it sends the server its reopen requests. For some clients, the reopen RPCs time out, and the clients enter the *timing out* state, until the server responds to their next ping requests. When the server responds to a client's ping request, the client re-enters the recovering state and starts sending the server its recovery information again. Clients may alternate between recovering and timing out indefinitely. The number of clients recovering oscillates up and down during the recovery storm. Finally, some clients manage to complete recovery for the last time enter the *finished* state. This reduces the load on the server and enables other clients to complete recovery. These measurements were taken in the old Sun-3 setup with 41 clients. In this recovery storm, distributed cache state recovery takes over 500 seconds to complete. This recovery storm may seem severe, but it was not an unusual example.

3.4.2. Solving the Contention Problem

To solve the contention problem, clients must reduce their demands on the server during periods of high load. All distributed systems need this ability, unless they have a large amount of wasted capacity during normal processing. Contention control is useful during any period of high server demand, and not just during recovery. This section describes two ways to reduce contention in Sprite and explains why I chose the second. It then shows the improved system behavior during recovery as a result of reducing server contention.

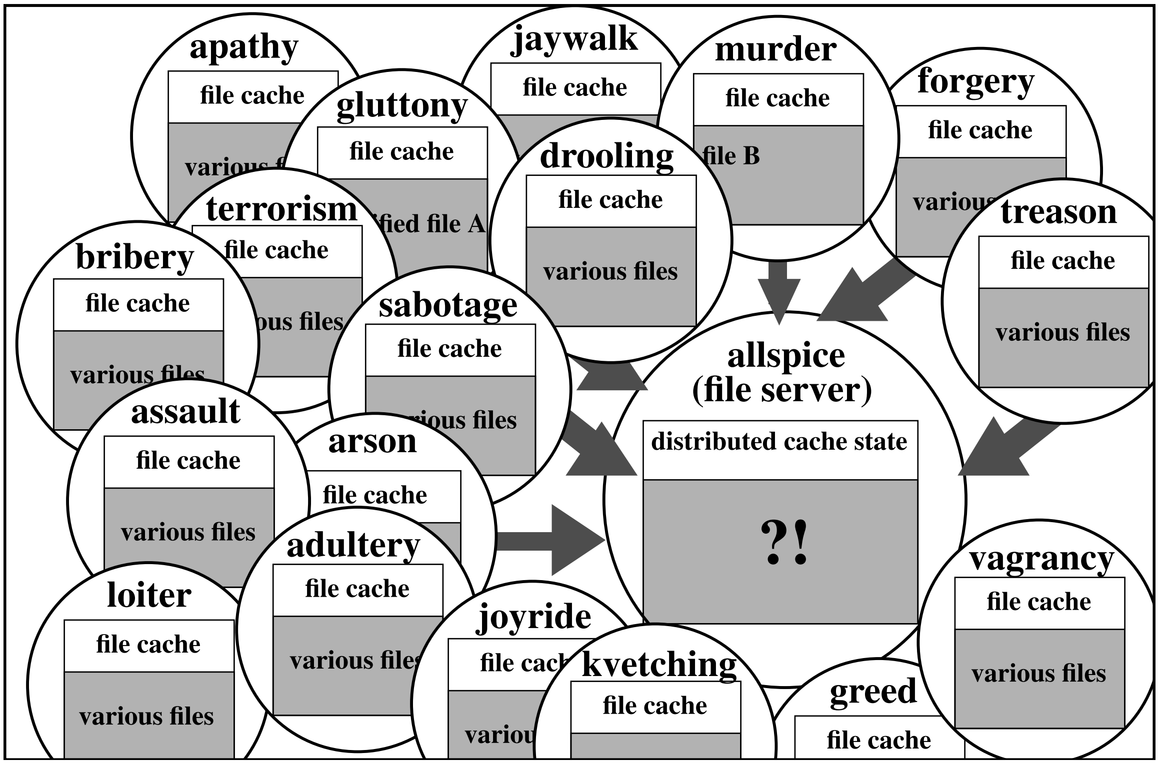

Figure 3-3. Recovery storm in Sprite.

In a recovery storm, the server is overwhelmed by reopen traffic from recovering clients.

One way to reduce client demands during periods of high load is to reduce the number of concurrent requests each individual client can make of the server. We can do this by decreasing the number of *RPC channels* per client. An RPC channel is a set of data structures that keeps track of the state of an RPC request. A client process that wishes to communicate with the server is assigned an RPC channel. Each client has a set number of RPC channels which permit that number of RPCs to progress concurrently. A process usually reuses the same channel for subsequent RPCs, but the channel may be reassigned if there are more processes than there are channels. To reduce the load on the server, we could ramp down the number of channels per client. For example, the client could automatically reduce its channels if the server fails to respond to some number of RPC requests.

I discarded this solution for two reasons. First, even if each client has only one channel, the clients can still overload the server if they each make a single request at the same time. In fact, a client uses only one channel to issue its recovery requests, because only a single process handles the client's recovery activities. This means that reducing the number of channels to one per client will not reduce the load due to reopen requests. Second, this solution is complicated. If all of a client's channels are in use, this solution requires shutting down active channels and redirecting the processes associated with them to wait for another channel.

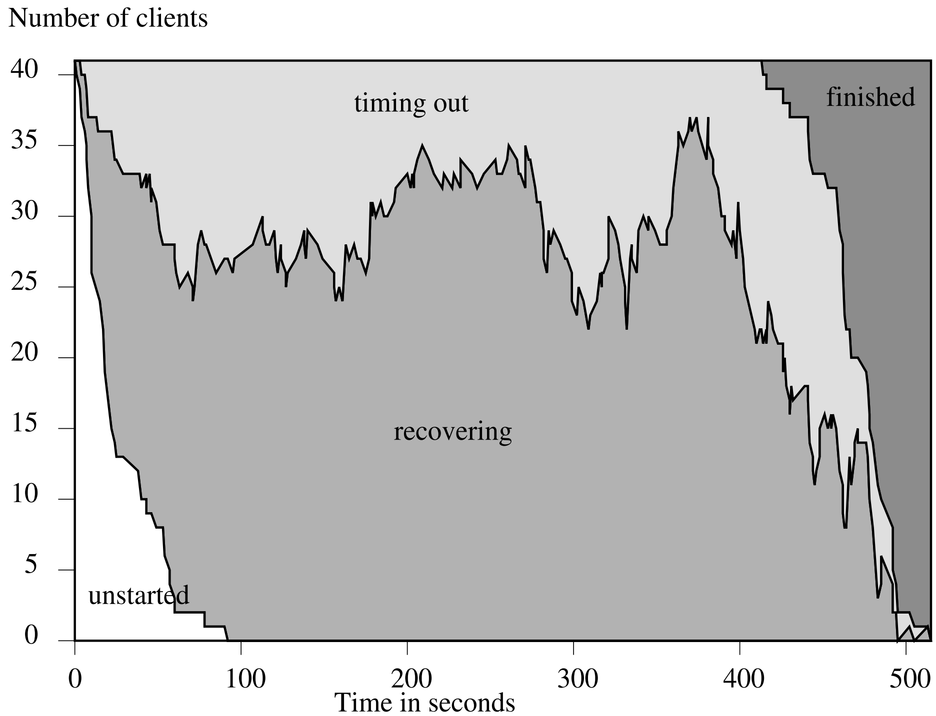

Figure 3-4. Recovery storm behavior.

This figure illustrates the states clients pass through during a recovery storm. The X axis shows time progressing. The Y axis gives the number of clients. The different regions represent different states. To read the graph, take a vertical slice at any point in time. The height of the different regions along that slice gives the number of clients in each of those states at that time. For example, at time 0 the clients all begin in the *unstarted* state. At 200 seconds, 33 clients are recovering, eight are timing out and none has finished. This recovery storm was measured in the old Sun-3 setup with 41 clients and took 515 seconds to complete.

Instead, I chose a solution that both reduces the client load and eliminates instability in the system. I added congestion control in the form of *negative acknowledgments* to Sprite's RPC protocol. A negative acknowledgment is an immediate response from the server to a client RPC request. The response merely informs the client that the server is still alive but too busy to handle the request. The server detects it is overloaded when no RPC server processes are available. Instead of ignoring a client request under this circumstance, the server responds with a negative acknowledgment (NACK).

NACKs allow the server to shed load without causing instability, because they tell clients that the server has not crashed – it is just too busy. The clients can then back off for a period of time (using an exponential back-off scheme) and continue their requests later, reducing the load on the server. NACKs also eliminate the source of instability, because they indicate to clients that the server has not lost their state again, since it has not crashed again. The clients then know it is not necessary to reissue all their reopen requests.

Negative acknowledgments are a general mechanism for congestion control. They are useful during normal processing and not just during recovery. Any time the server becomes overloaded with client requests, negative acknowledgments allow it to shed load without destabilizing the system. For example, NACKS provide congestion control during large system compiles, when many clients simultaneously run migrated compilation processes. Even during periods of lighter server loads, we often see more than ten NACKS issued by the server in one day.

It is important, though, that the congestion control mechanism use very little processing on the server, or else it will make the server loading problem even worse. NACKs are a good solution, because they do not add much overhead on the server. To respond with a NACK, the server does not need to interpret the client's request in any way. An RPC server process is not needed to issue the NACK. The NACK can thus be executed at a very low level (at interrupt level in most systems), without using much CPU processing or other server resources.

Figure 3-5 shows the changed recovery behavior of the system with a server that can generate negative acknowledgments. This measurement is also from the old Sun-3 setup with 41 clients. The first feature to note is that no clients enter a timing-out phase during which they believe the server is down. Instead, the clients progress directly from reopening their handles to the finished state. The recovery phase may still take a while, especially if the clients receive many negative acknowledgments and must wait for the server, but the clients always make forward progress recovering their cache state. The second feature to note is that recovery is shorter. Clients do not make repeated recovery attempts, so the overall recovery period takes only about 110 seconds rather than the previous 500.

However, this speedup is also due to staggering client recovery requests, as described in section 3.4.4. The following experiment measures the benefit due alone to NACKS, without any other variables. In the more controlled environment of the testbed setup, I measure recovery times with and without NACKS. To initiate the recovery storm with only ten available clients I cause each client to recover more state than in the basic state setup, and I restrict the number of RPC server processes on the file server to three, thus simulating a server with less processing power. I chose the parameters to this experiment to ensure a high enough load on the server to cause a recovery storm; they are otherwise arbitrary. Each client recovers 3020 handles: 1000 file I/O handles for unopened files, 1010 file I/O handles opened for reading and the corresponding 1010 stream handles to those open files. Ten of the open files are shared between all the clients, but the other files are all unique. In addition, two clients create and unlink three files every second, and one client truncates and overwrites a file every second. In the first set of measurements, the server could not generate negative acknowledgments, while in the second it could. The setup for the two measurements is otherwise identical.

The following figures show the results of these experiments. While not as chaotic as the previous recovery storm, Figure 3-6 shows similar behavior. Clients alternate several times between the recovering and timing-out states. Overall, the recovery storm requires over four minutes (250 seconds) to complete. Figure 3-7 shows the same scenario but with a server that generates negative acknowledgments. There is no longer a phase during which client RPCs time-out, because they receive negative acknowledgments instead. Eliminating the instability cuts recovery time in half, to two minutes. These measurements show that negative acknowledgments alone are a successful form of congestion control.

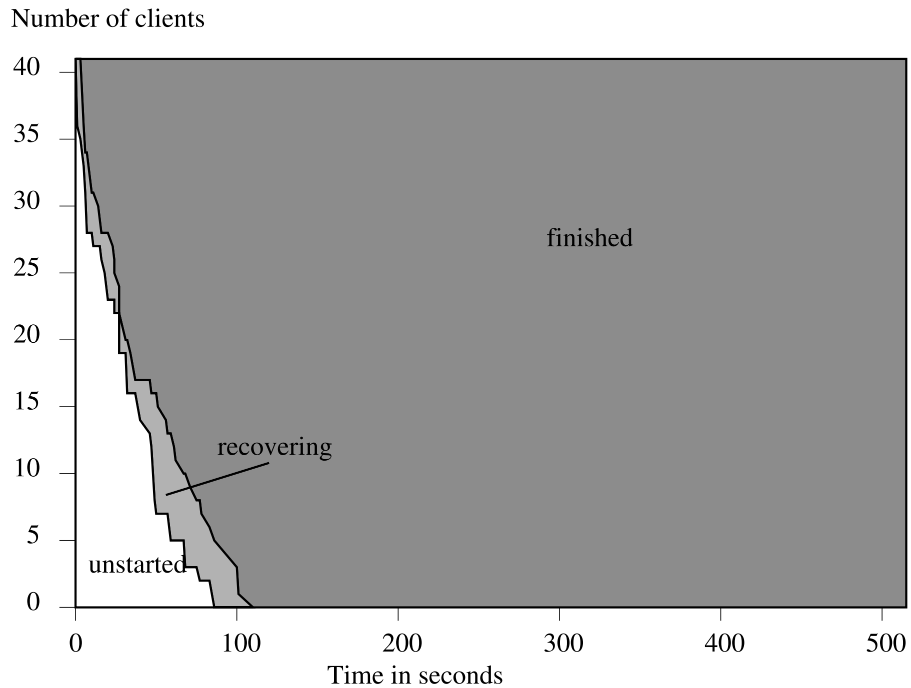

Figure 3-5. Recovery with server NACKs.

This figure illustrates the states clients pass through during recovery with a server that can respond with negative acknowledgments. The X axis shows time progressing. The Y axis gives the number of clients. The different regions represent different states. To read the graph, take a vertical slice at any point in time. The height of the different regions along that slice gives the number of clients in each of those states at that time. For example, at time 0 the clients all begin in the *unstarted* state. At 40 seconds, 14 clients still have not started recovery, three are recovering, and 24 have finished recovery. This recovery session was measured in the old Sun-3 setup with 41 clients and took 110 seconds to complete.

3.4.3. Synchronized Client Recovery Requests

A second problem that contributes to recovery storms is that many clients may initiate recovery with the server simultaneously, rather than staggering their recovery requests. This puts a sudden recovery load on the server. I found that the reason for this behavior is that file server reboots synchronize the clients over time and cause them to initiate recovery together. This is an interesting example of a phenomenon seen elsewhere in distributed systems: global events (in this case the server reboots) can cause unexpected and undesirable synchronization of different elements of a distributed system. The results are similar to those found in the *convoy phenomenon* [Blasge79].

Figure 3-8 shows this recovery synchronization problem. At 83 seconds through recovery, 15 clients initiate recovery with the server during the same second. At 84 seconds, another 15 clients initiate recovery with the server. These clients place a sudden load on the server when they all ini-

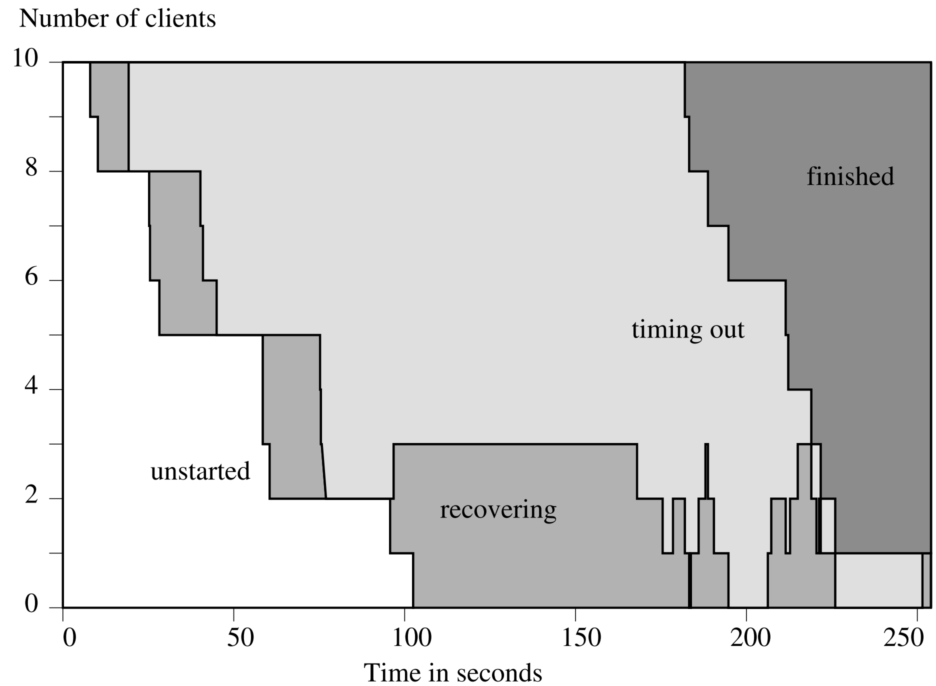

Figure 3-6. Recovery storm in testbed setup.

This figure illustrates the states clients pass through during a recovery storm. The X axis shows time progressing. The Y axis gives the number of clients, using the same approach as used in Figure 3-4. This recovery storm was measured in the testbed setup and took about 254 seconds to complete.

---

tiate recovery together. Unfortunately, once a group of clients caravans this way, the caravans do not break up; after the next server crash, the same clients will initiate recovery together again.

This synchronization was caused by the implementation of the reboot detection mechanism. To detect when a server reboots, each client pings the server every 30 seconds, as described in section 3.1.1. While the server is still unavailable, the ping RPCs time out, because the server cannot acknowledge them. But before the client's RPC system actually considers the RPC to have timed out, it re-issues it several times, in case the first attempts failed due to a temporary network problem. In Sprite, the RPC is retried six times over the course of five seconds. If any of the attempts receives a response from the server, this means the server is available and the RPC need not be further retried. Because clients ping a crashed server every 30 seconds, and spend five seconds pinging it on each attempt, there is a one-sixth probability that a client will be in the midst of pinging when the server finally becomes available and responds to the client. Because of this, one-sixth of the clients are likely to detect the server reboot within the same second. This phenomenon is further illustrated in Figure 3-9.

Once a set of clients becomes synchronized this way, the group does not split up. This is because clients reset their ping timers when they detect the server reboot. While the server remains avail-

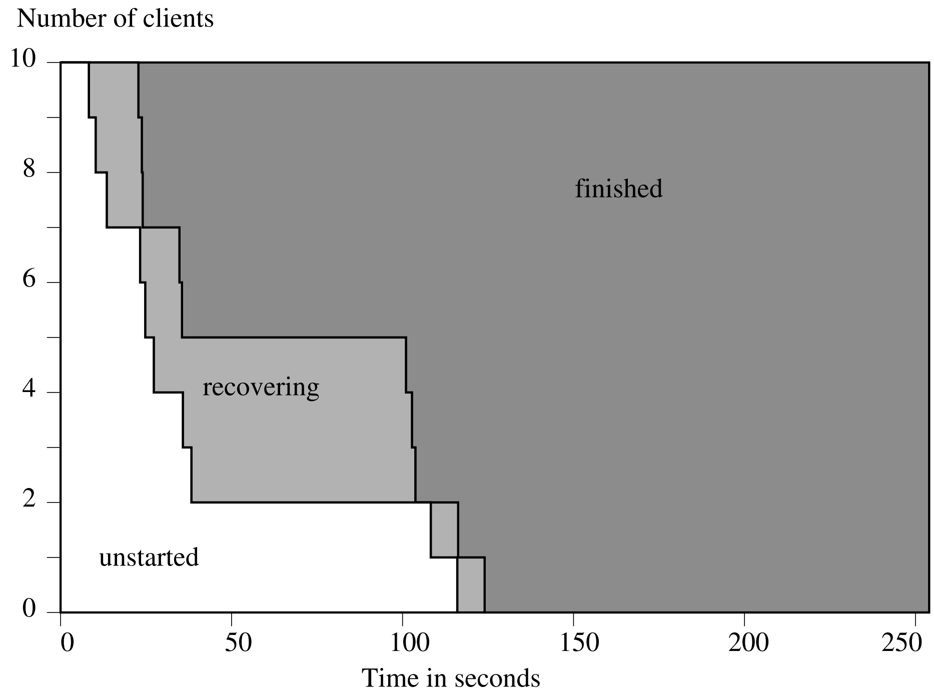

Figure 3-7. Recovery with NACKS in testbed setup.

This figure illustrates the states clients pass through during recovery with a server that can respond with negative acknowledgments. The X axis shows time progressing. The Y axis gives the number of clients, using the same approach as was used in Figure 3-5. This recovery session was measured in the testbed setup and took 124 seconds to complete.

able, the pings are skipped, but they are still rescheduled at 30-second intervals. Because one sixth of the clients detect the server reboot together, one sixth of them set their timers to re-ping the server together. From this point on, the clients' reboot detection mechanism is synchronized, so they will detect a server reboot simultaneously and will initiate recovery together. Only rebooting a client will desynchronize its ping timer, until it again joins a caravan.

3.4.4. Staggering Client Recovery Requests

To fix the synchronization problem it is necessary to stagger the clients' ping RPCs. An easy way to do this is to set the client pings to occur at 30-second intervals after a random event, rather than after the global event of the server reboot. I use a client's own reboot time for the event. While not really random, individual client reboots are well-enough staggered that it has solved the synchronization problem, as seen in Figure 3-10. There are now no groups of more than three clients that detect the server reboot simultaneously, so there are no more than three clients that begin recovery within the same second.

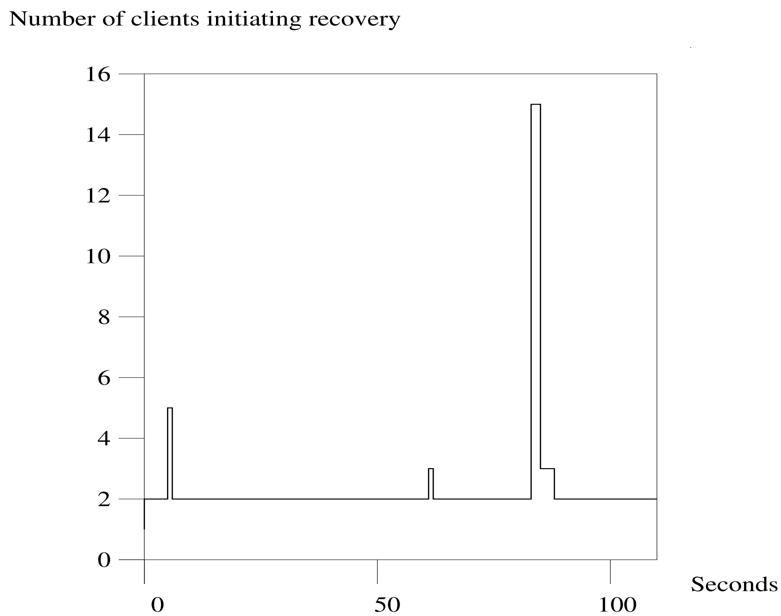

Figure 3-8. Synchronization of client recovery.

This figure illustrates the grouping of client recovery attempts. The X axis shows time through recovery. The Y axis shows the number of clients initiating recovery with the server during a one-second interval. For example, at 83 seconds 15 clients initiate recovery. Fifteen clients also initiate recovery at 84 seconds. Many clients initiate recovery more than once, due to timing out during the recovery storm. This recovery session was measured in the old Sun-3 setup.

The combination of staggering client recovery and implementing negative acknowledgments in the RPC system significantly improves recovery performance. In the old Sun-3 setup, these improvements reduced client-driven recovery from many minutes to about two minutes. In the testbed setup, negative acknowledgments alone reduced an artificial recovery storm from 254 seconds to 124 seconds.

Finally, I include a further measurement as a base against which to compare later performance improvements in this chapter. This test uses the basic state setup in the testbed setup, with no extra state to cause a recovery storm. The test also includes NACKS and staggered client recovery attempts. With this setup, client-driven recovery requires 25 seconds, on average. This measurement was taken four times, with a standard deviation of 1.4 seconds.

3.5. Other Performance Problems and Solutions

Adding negative acknowledgments and staggering client recovery eliminates recovery storms by controlling server congestion, but there are two other performance problems I discovered while examining client-driven recovery. The first is that it is easy to recover more state than necessary.

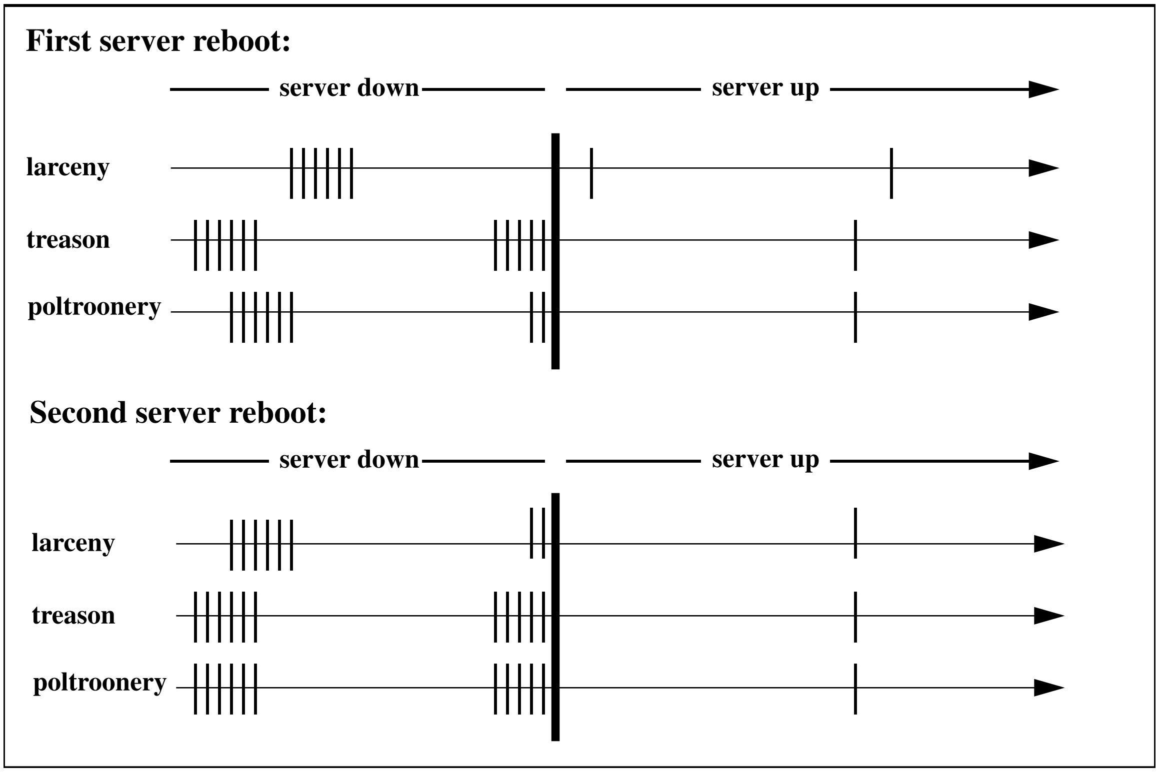

Figure 3-9. How client recovery becomes synchronized.

This figure illustrates how the clients' server reboot detection mechanism becomes synchronized over the course of two server reboots. The figure shows three clients, with time progressing from left to right. Tick marks indicate when a client is pinging the server. In the top half of the figure, the server is down, and the three clients all ping the server starting at different times. The server becomes available while both treason and poltroonery are pinging it, so the two clients detect the server reboot at the same time and synchronize their pings intervals. Single tick marks after the server is up indicate when the clients would ping the server again if they needed to. For the second server reboot, treason and poltroonery remain synchronized to ping the server at the same time. When the server reboots, though, larceny is also pinging it, so all three clients become synchronized and will initiate recovery together.

The second is that it is easy to recover the state using more RPCs than necessary. Unfortunately, these problems are likely to exist in any stateful system that uses a communication-based recovery protocol. The following two sections describe how I've modified Sprite's recovery protocol to reduce the amount of state recovered and the number of RPCs used. The third section shows the performance improvement that results from these optimizations taken together; separate measurements of the individual benefits due to these optimizations are made in chapter 4, using server-driven recovery. (The optimizations apply equally well to server-driven recovery, and it is possible to gain more exact measurements using server-driven recovery; in client-driven recovery, there is a high variability in times, because the server must wait for clients to contact it.)

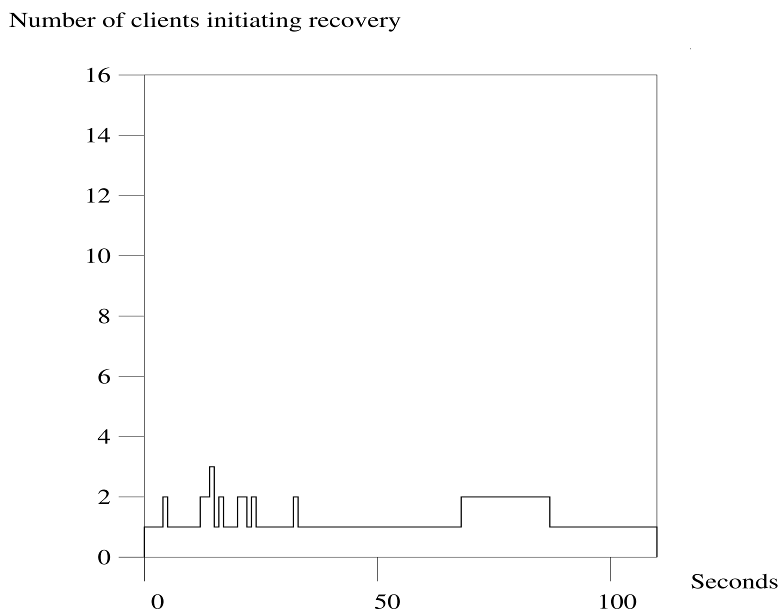

Figure 3-10. Staggered client recovery.

This figure illustrates the grouping of client recovery attempts after staggering the client-to-server ping RPC timing. The X axis shows time through recovery. The Y axis shows the number of clients initiating recovery with the server during a one-second interval. For example, at 14 seconds, three clients initiate recovery. Some clients initiate recovery more than once, due to timing out during the recovery storm. This recovery session was measured in the old Sun-3 setup.

3.5.1. Eliminating Unnecessary State Recovery

Sprite clients tend to recover more state than needed. Redundant state on the clients and server is useful for recovery, but too much state is not, because it takes extra time to recover. It is important to optimize the amount and type of state recovered in a stateful system.

For simplicity, Sprite clients reopen all their handles and streams. This includes handles for files that are neither open nor have cached dirty blocks. But to guarantee cache consistency the server only needs to know about files that are currently open or that currently have dirty data located in a client cache. If an unopen file merely has clean data in a client cache, the client will find out with its next open request whether or not its cached blocks are stale, so there is no need to tell the server about such a file. As shown in section 3.2, eliminating this extra state from recovery reduces the information recovered to about one-fourth its previous amount.

3.5.2. Batching Reopens

A further optimization of the recovery protocol decreases the number of reopen RPCs by packing many reopen requests into a single RPC. This reduces the overhead from extra communication. Clients originally executed one reopen RPC per handle or stream to recover. However, given the maximum size of a single handle's or stream's state information (52 bytes) and the maximum size of a Sprite RPC packet (16 kilobytes), we can batch about 300 reopen requests into a single RPC. Thus, it is only necessary to do 1/300 the number of RPCs for client-driven recovery.

To batch together the reopen requests, the client collects the state information for all the handles or streams that fit in an RPC packet and sends the request to the server. Because the client opens I/O handles and streams in two separate steps, this necessitates at least two RPCs. The server iterates through the list of handles or streams, performing the same processing per item as it did previously. It records the success or failure of the reopen for each item in an array of results. When it finishes processing the reopen request, the server returns this array of results to the client. The client then iterates through the array of results and updates its handle or stream information to reflect the status of each reopen request.

Two additional RPCs are necessary, per client, for a total of at least four RPCs. Each client must reopen the prefix for the file system served by the recovering server. Finally, each client also sends an *end recovery* RPC to signal to the server that it has reopened all its state.

3.5.3. Basic Client-Driven Recovery Performance

Combining all of the improvements to client-driven recovery discussed in this chapter (NACKS, staggered client recovery requests, reopening less state information, and batching reopen requests), reduces client-driven recovery times in the testbed network with the basic state setup to an average of 21 seconds. This test was run four times with a standard deviation of 1.4 seconds. This result shows an average improvement of about four seconds when compared to the test on page 52 that used only NACKS and staggered client recovery requests.

I call this measurement the *basic client-driven recovery measurement*, because I compare it with the results of other recovery techniques in later chapters. The later measurements use the same testbed and basic state setup and also include the recovery improvements due to NACKS, reduced state information and batched reopen requests.

However, this basic client-driven recovery time can vary significantly. While staggering client pings helps reduce server congestion, it also allows for a high variation in client-driven recovery times, because clients ping the server at different times over a 30-second interval. Thus client-driven recovery can easily take 30 seconds.

Because the server must wait for clients to detect its reboot and recover their state, the server spends most of its recovery time idle, waiting for clients to send it their state. Out of the 21 seconds for this basic client-driven measurement, the server's CPU is idle 87% of the time, and its disk is idle 93% of the time.

3.6. Other Client-Driven Recovery Problems

The preceding sections describe problems that are solvable within the client-driven recovery approach, but this section turns to two problems that are not: cache consistency violations and limited recovery speed.

3.6.1. Cache Consistency Violations

Client-driven recovery allows cache consistency violations to occur during recovery, because the server does not know when recovery is over. The server knows that an individual client has finished recovering when it receives the client's *end recovery* RPC, but it does not know what clients must recover with it to consider system-wide recovery complete. The server must allow clients to continue their normal activities at some point, even if other clients have yet to recover. If this happens, one client may finish recovery before others and may open and read a file that is cached on a slower client. If the slower client still has dirty data for the file in its cache, the faster client will read stale data from its cache or the server's. If the faster client closes the file before the slower client recovers, the cache consistency violation will not be detected in time to signal the error.

It is also possible for a cache consistency violation to cause data loss. A slow client will not be allowed to write back dirty data for a file that another client is already using. When the slow client contacts the server for recovery, it will receive an error message. After a reboot of the main file server in Sprite, there are usually between zero and five cache consistency violations, with the average number being one.

Interestingly, as client-driven recovery has gotten faster in Sprite, the average number of cache consistency violations due to recovery appears to have increased. I speculate that this is because adding negative acknowledgments, staggering recovery requests, and batching reopens has serialized client recovery somewhat. It is now more likely for one client to finish recovery completely and begin other operations before another client begins recovery.

There are two ways to address the cache consistency problem. The first is to give the server a list of clients with which it must recover. After all the clients on this list recover with the server, the server knows it is okay to allow clients to resume normal processing. This list must be kept in stable storage so that it is not lost across server failures. This technique is a modification of client-driven recovery and is discussed in the following chapter on server-driven recovery.

The second solution is for the server to block all file requests, except those for recovery, for at least the length of a ping interval. Since clients ping the server every 30 seconds, all available clients should have contacted the server during the 30 seconds. The server can wait until all the clients that contact it during that time finish recovery. At this point it knows that it has completed its distributed state and can allow clients to carry on with their normal processing.

The problem with the second approach is that it guarantees that state recovery will always take at least 30 seconds. Despite the extra time required by this approach, this is how the DEcorum file system (described in chapter 2) currently avoids cache consistency violations in its client-driven recovery implementation [Kazar93]. But given our desire to provide very fast recovery times in Sprite, I chose to discard this solution and opted instead to try server-driven (and transparent) recovery.

4 Server-Driven Recovery

This chapter describes server-driven recovery, a modification of client-driven recovery in which the server initiates and controls distributed cache state recovery. Rather than wait for clients to initiate recovery, the server keeps track of its active clients and initiates recovery with them. Server-driven recovery eliminates cache consistency violations, is faster, and permits more control over server congestion and the order of client recovery. Compared to the basic measurement of client-driven recovery in chapter 3, server-driven recovery in the testbed setup takes only about 2.0 seconds rather than 21. The recovery time scales linearly with the number of clients; each additional client adds 130 milliseconds. The server spends this additional time per client fetching file descriptors for the state reopened. Disk I/O is the limiting performance factor for server-driven recovery in Sprite.

The key element of server-driven recovery is a list of active clients kept on the file server. To avoid the cache consistency violations suffered by client-driven recovery, the server needs to know when its clients have finished recovering. With this knowledge, the server can wait until after recovery to process new client requests that would change the distributed cache state. When all the clients on its list have finished recovering, the server knows that recovery is done.

Server-driven recovery has two further advantages over client-driven recovery: it is faster, and the server has more control over recovery. It is faster, because the server does not need to wait for the clients to detect its reboot. It can initiate recovery with all of them when it chooses. The server has more control over its recovery load, because it can decide to recover with a limited number of clients at a time. It can even choose the order of clients for recovery, which allows it to recover first with other file servers. This is helpful, because one file server can depend on resources on another file server. For example, in Sprite the root of the entire file system is served by only one file server, so all other hosts in the system are clients of this server. If the root server fails, some activities on these other servers will hang until the root server recovers. With server-driven recovery, the root server can recover first with the other servers, freeing up their resources more quickly.

These combined advantages are likely to make server-driven recovery the distributed cache state recovery technique of choice for many systems. For example, Spritely NFS [Mogul92] [Sriniv89] has chosen to implement server-driven recovery, as described in chapter 2.

This chapter has the following organization. The first section describes the design and implementation of server-driven recovery. Section 4.2 lists some disadvantages of this recovery technique as compared to client-driven recovery. For example, it requires more special-purpose code and necessitates trickier synchronization and locking in the kernel. Section 4.3 gives performance measurements. These include timings of server-driven recovery and levels of server resource utilization across varying numbers of clients to show how recovery will scale to a larger system than the testbed setup. Finally, I measured the individual benefits of two recovery optimizations described in the previous chapter: recovering only necessary state information and batching reopen requests.

4.1. Design and Implementation

There were several issues involved in the design and implementation of server-driven recovery: where to store the active client list, how to update it, and how to make sure it is not lost or corrupted. Where there are choices for resolving these issues, I opted for the simplest design if it had no significant performance penalties. In particular, I made as few modifications to Sprite's client-driven recovery as possible.

4.1.1. Performance Impact of Client List Updates

The first design issue is where to store the client list for good reliability and performance. The list could be stored in the server's main memory and preserved across failures using a technique such as the recovery box, described in chapter 5. Storing the list in main memory would allow the fastest possible list access, but would require care to prevent corruption or loss of the data during a failure. Alternatively, the list could be stored solely on the server's disk. This is the easiest place to preserve the list across server failures, but it would be too slow if a disk I/O is required for every list access. If the only copy of the list were on disk, the server would have to read the disk on each RPC from a client to see if the client is already on the list or should be added to the list.

Instead, I chose to store the list in both places: the copy in memory provides fast read access while the copy on disk survives failures. With this solution, the server reads the list from disk only once, after a failure, and all further read accesses use the copy in memory. This avoids overhead in the normal case where the client is already on the list. Keeping this main-memory copy requires no extra complexity, because servers and clients already maintain such main-memory host list data structures for crash and reboot detection (as described in chapter 3). I just make use of the data structures already present.

However, disk writes are still necessary whenever the server must update the list. There are two cases when the server updates the client list. In the first case, the server should remove from the list any client that crashes. This prevents the server from leaving dead clients on the list. This is important, because the server would otherwise contact the dead clients for recovery. Contacting dead clients slows recovery, since the server RPC to the dead client to initiate recovery will take about five seconds to time out. Fortunately, the overhead of the disk I/O to delete a client does not hurt the client, because it only occurs when the client has crashed.

In the second case, the server must add a client to the list the first time it issues a request that would change the distributed cache or file system state. The server only needs to add a client the first time it contacts the server with such a request, because subsequent contacts will find the client on the list. The first request from a client that changes this state is likely to be an open request, or perhaps a delete or create. In practice, though, it is acceptable to add the client the first time it sends any RPC request to the server. This is because a client that contacts the server for the first time with any RPC is likely to follow that with a request that affects the distributed state.

In this case, unfortunately, the extra disk I/O will impact the client, by slowing down its first RPC to the server. This first RPC suffers the extra latency of a disk I/O, which makes it an order of magnitude longer than it would otherwise be (tens of milliseconds, depending on the type of server disk). The server must complete the I/O before it responds to the RPC to ensure that it knows to recover with the client in the event that it crashes right after responding. This is important, because the client's first RPC may cause a state change. If the server does not contact the client for recovery, it will not recover the new state. If this I/O overhead on the client is a problem for some application, it should be possible to pre-contact the server, so that the client will already be on the server's list before the application runs.

Luckily, these list updates are infrequent, because the list of active clients on the server does not change often. For instance, during May 1992, there were on average 10.7 client reboots per day in the production Sprite setup. This would cause only 10.7 additions and 10.7 deletions per day to the client list. The standard deviation of this measurement is 8.2, with a maximum of 35 reboots on one day and a minimum of zero reboots on two other days. The number of client reboots is higher in Sprite than in a non-research system, because I and several other users perform experiments that often require testing, debugging, and rebooting kernels many times a day. John Hartman, for example, often reboots the same machine 15 times in one day.

4.1.2. Maintaining and Recovering From the Client List

The second design issue in server-driven recovery is how to maintain and recover from the client list; in particular, we must decide what part of the system should handle disk I/O to the list. One possibility is that the file server's kernel could issue the disk writes to update the list and read the list for recovery from it. Since the file server kernel keeps track of the list of active clients in memory, this sounds reasonable. However, it is simpler in Sprite to issue disk I/O from user level than from within the kernel, and it is easier to debug a user-level process than kernel code. For these reasons I chose to use user-level processes on the server to maintain and recover from the list on disk.

In my design, a user-level daemon process issues updates to the list during normal execution. Figure 4-1 shows the arrangement of the server kernel and user-level daemon process. Whenever the server receives an RPC from a client that is not in its main-memory client list, it updates the main memory list and informs the user-level daemon of the new client. The kernel RPC server process that handles this RPC then sleeps until receiving a response from the daemon. The daemon writes the change to disk and responds to the kernel. The RPC server process then wakes up and responds to the client.

Although managing the client at user-level is easier to debug, it does not reduce the importance of correctness. Just like the kernel, the user-level daemon process is a point of failure. If the daemon process dies or hangs, the server is unable to maintain its client list on disk. My current solution to this problem is to start a software timer in the file server kernel every time the kernel attempts to communicate with the user-level daemon. If the server kernel does not hear back from the daemon before the timer goes off, the kernel assumes the process has encountered some failure. The kernel then panics and reboots. The timer must be long enough that small glitches in the

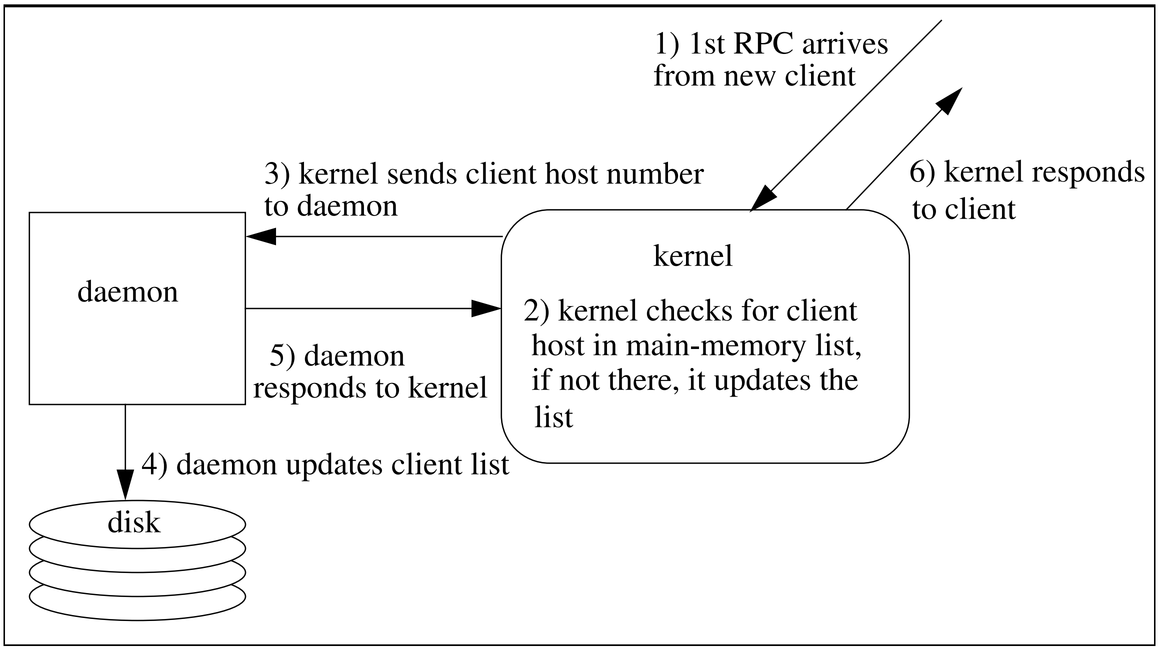

Figure 4-1. User-level daemon updates active client list.

This figure shows how the file server maintains its active client list. Everything in the illustration is running on the file server. Whenever the server's kernel receives an RPC from a client that is not already on the active client list, it updates the list in memory to include that client and informs the user-level daemon of the client's host number. The daemon updates the list on disk and responds to the kernel indicating the update completed. Only then does the server's kernel respond to the client.

daemon process's performance do not cause the kernel to panic unnecessarily. A value of five seconds for the timer has worked well. So far, daemon failures have not been a problem.

Another user-level process is responsible for reading the client list during crash recovery. When the server reboots, it starts up a user-level recovery process. The recovery process reads the client list from disk and informs the server kernel of the list of clients to contact. The server initiates recovery with the clients and updates its main-memory list of active clients in the normal fashion as these clients respond to the server. If any client has crashed while the server was down, the recovery RPC to the client will time-out using the normal crash detection mechanism, and the client will be removed from the client list on disk by the user-level daemon. To initiate recovery with a client, the server sends a *start recovery* RPC to the client. The client then executes the same code to recover its state as it would have for client-driven recovery. As described in chapter 3, clients end their recovery sessions by sending an *end recovery* RPC to the server. Figure 4-2 shows the communication between the recovery process and the kernel during crash recovery.

After the kernel receives the client list from the recovery process, it decides what to do with the list. The kernel can choose the number of clients to recover with at once and it can choose which clients to recover with first. This gives the file server an opportunity to manage its recovery load

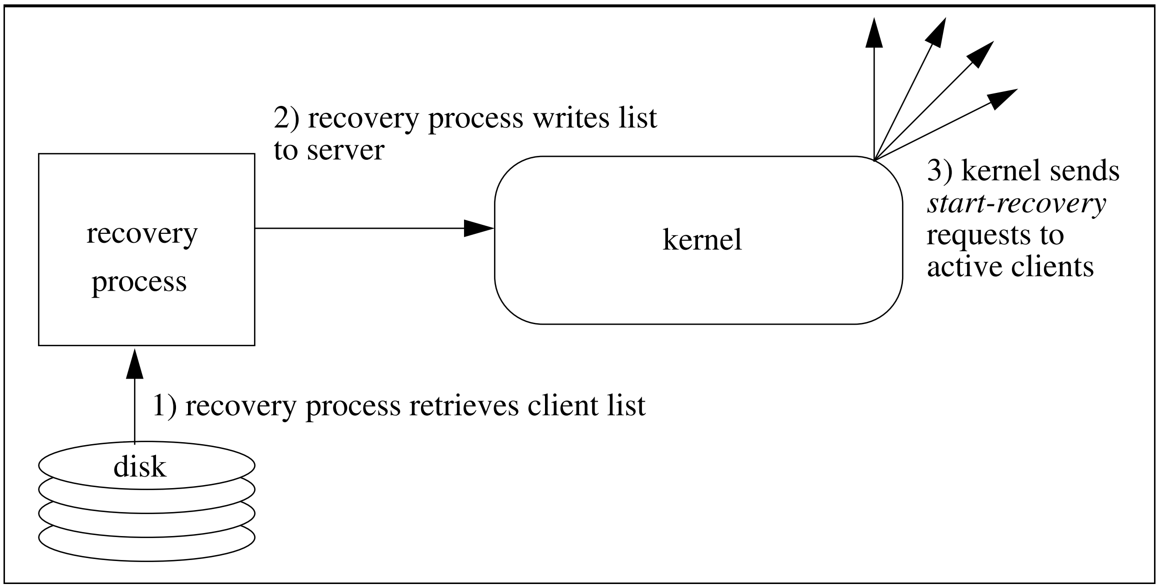

Figure 4-2. The kernel uses the active client list for recovery.

This figure shows how recovery from the client list works. Everything in the illustration is running on the file server. As part of rebooting, the server starts up the user-level recovery process. The process reads the list of active clients from disk and sends the list to the kernel. The file server kernel then sends a message to each active client requesting it to begin its recovery activities with the server.

and control congestion. With the current size of the Sprite system (20 clients), negative acknowledgments have worked well enough for congestion control that we have not had to limit the degree of concurrency during server-driven recovery.

To communicate with each other, the file server kernel and user-level processes use a device-like interface. The daemon receives client list updates by reading from the "device," while the kernel writes these updates to the "device." The daemon informs the kernel when it has safely finished the file I/O by issuing an I/O control to the device. During recovery, the recovery process writes the list of active clients to the device, and the kernel reads the list from it.

4.1.3. List Format and Atomic Update

The major concern in the design of the client list is to prevent its loss or corruption in the event of a failure. To accomplish this, I use two techniques: a simple list format that helps to detect corruption, and an atomic list update that uses two copies.

Figure 4-3 is a diagram of the list format. The file header contains a *magic number*, a timestamp, and the number of active clients. The magic number is an integer agreed upon by the user-level daemon and recovery process that is unlikely to be present accidentally at the given offset in a random file. Thus, it must be an integer other than 0, -1, an ASCII character, or an even word address.

If the magic number is not present, then either the file is corrupted or it is not really a client list file. A checksum would provide better corruption detection, but the magic number is simple and has worked well in practice. The timestamp is used for atomic file update, as described below. The number of active clients makes reading the file easier and also helps detect corruption. If the file does not contain the indicated number of clients, then something is wrong. Following the header is a list of tuples. Each tuple contains the host number of the active client and an indication of whether the host is also a server. As described previously, this allows the failed server to recover first with other servers, but I currently do not make use of this possibility and have no good mechanism for automatically determining when a client is also a server.

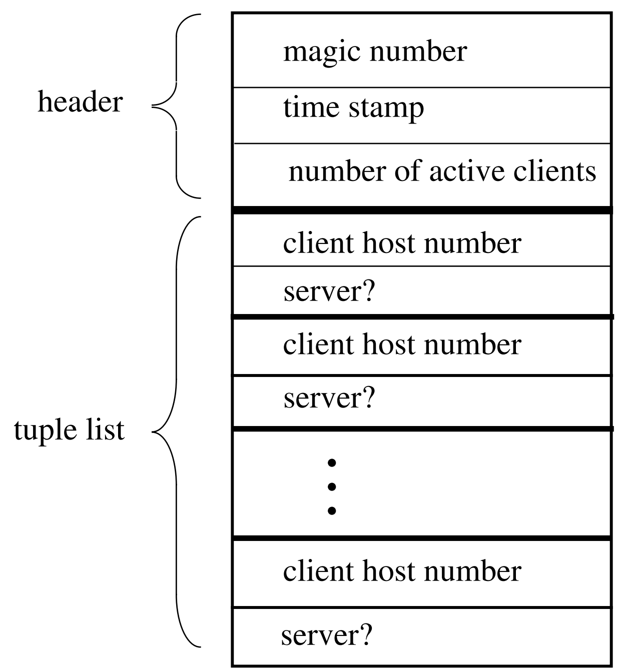

Figure 4-3. Client list format.

This figure is a diagram of the format of the active client list file. The file starts with a header containing a magic number, a time stamp, and the number of active clients listed in the file. The client list follows, in the form of a list of tuples. Each tuple contains the client host number and an indication of whether the client is a server.

To prevent corruption or loss of the client list due to a server crash, the daemon performs atomic updates to the list by using two copies. To update the list, the daemon overwrites the older copy of the list with the latest update. If the server crashes part way through this update, the newer copy of the list is still available. The only information lost will be the latest update. This is not a problem, because the latest update is not valid if the server crashes before the daemon acknowledges successfully writing the new list.

During recovery, the file header is used to select the correct file and to check that the file is complete. Before an update, the daemon clears a field in the header. After a successful update, the daemon writes the date of the operation to the cleared field. Each of these file modifications is forced synchronously to disk using the UNIX *fsync* system call. For recovery, the recovery process chooses the latest successfully updated copy of the list. The recovery process knows if the most recent copy of the list was successfully updated by examining its header. The recovery process checks the field in the header to make sure it is no longer cleared and that it contains the most recent date. This means the file was correctly and completely updated.

4.2. Problems with Server-Driven Recovery

While server-driven recovery has many advantages compared to client-driven recovery, it also has several problems: it requires more special-purpose code; it is vulnerable to slow clients; it may recover with more clients than necessary; it requires complicated synchronization and locking in the kernel; and, depending on the desired level of robustness, a backup recovery mechanism such as client-driven recovery may be desirable. This section explains these problems.

4.2.1. Special-Purpose Recovery Code

I have tried to minimize changes to Sprite for server-driven recovery, but there were still four areas where I had to add new code: I implemented the user-level daemon and recovery processes, added the “device” interface, changed the recovery system, and added a new RPC request. The user-level processes are not much of a problem, simply because they are at user level. Maintaining and reading the client list at user level is much simpler than doing this in the kernel. However, this arrangement requires an interface for the user-level processes to communicate with the kernel, implemented in this case as a special device. Adding a device to Sprite is not hard, but this device adds about 750 new lines of C code to Sprite. (Over a quarter of the lines are comments.) I also changed the recovery module to tell the device about clients to add to or to remove from the list, and also to provide synchronization described below. To the RPC system I added the new *start recovery* RPC that the server uses to notify clients that they should initiate recovery.

4.2.2. The Slow-Client Problem

Although the server has some control over server-driven recovery, it is still dependent upon its clients for fast recovery. The server can only recover as fast as its slowest client. A client can hold up recovery for the whole system if it is slow sending its state or issuing an *end recovery* RPC. The solution to this problem is to use a different recovery scheme that avoids communication with clients during recovery, such as the transparent recovery approach described in the following chapter.

The client may be slow simply because it is running a lot of jobs, or it could have some bug that causes it to hang temporarily. For example, while taking measurements for this chapter, I found that SPARCstations periodically took several hundred milliseconds too long to respond to the server’s recovery requests. They were especially likely to do so after being rebooted. Eventually I discovered the source of the problem: clients print two short messages to their consoles, one when the server contacts them for recovery and another when they finish recovering. DECstations print quickly to their consoles, but SPARCstations do not, and they leave interrupts off during the console print operation. If a client is running an X server with a default window setup, system messages are redirected to a window rather than the console device. Printing to the window is much faster. After rebooting a client, though, I often forgot to restart its X server.

4.2.3. The Too-Many-Clients Problem

Another problem with server-driven recovery is that it may recover with more clients than necessary. Once the server adds a client to the client list, it will not remove the client unless it crashes. However, a perfectly healthy client may cease to have any interest in the server if it no longer needs any of the server's files or other objects. This is unlikely to occur in an environment with one central server, but in a system with several servers it can happen easily.

This problem has two effects. First, the server will waste time sending *start recovery* RPCs to clients with no state to recover. A worse possibility, though, is that the server may wait a long time for these clients to recover. The clients will respond to the *start recovery* RPC, and the server will mark them as recovering. But if the client actually has no state to recover, it may not execute any recovery code and may not send the server an *end recovery* RPC.

There are at least two solutions to this problem. The first is that a client can respond to the server's *start recovery* RPC by indicating whether or not it has state to recover. If it does not, the server will remove it from the client list and will not wait for it to recover. In this solution, the server still sends an extra *start recovery* RPC to the client after its first failure, but it will not do so after subsequent failures. This is the solution I have used in Sprite. A second solution is for a client to unregister its interest in a server when it no longer has state associated with that server. The server would then remove the client from its list. The advantage of this solution is that the server will not send any unnecessary *start recovery* RPCs, as long as the clients can be trusted to unregister themselves. The disadvantage is that a client may register and unregister itself frequently if it only uses a few of the server's files. Sprite clients do not currently have the ability to unregister their interest in a server.

4.2.4. Synchronization and Locking

Synchronization problems occur for a combination of reasons: the server must block a subset of RPC requests from the clients until all clients have recovered, and the list of active clients can change during recovery itself. These problems are all solvable, but finding the solutions required some experimentation, and the solutions may not be easily generalizable to other systems.

One reason for tricky synchronization on the server is that it must refuse certain kinds of RPC requests during recovery, but not others. To prevent cache consistency violations the server must block new open requests and other requests that can alter the distributed cache state or file system structure. However, it must not block client reopen requests or requests from clients to write back their dirty cache pages, because these requests are part of the clients' recovery activities. Blocking these RPCs will hang the system. Unfortunately, discovering which RPCs to block seems to require some trial and error. When there was any question, I erred on the conservative side and chose to block the RPC request. Table 4-1 and Table 4-2 list the blocked and unblocked RPCs for Sprite.

There is an alternative to blocking various RPC requests to prevent cache consistency violations. Spritely NFS (described in chapter 2) instead uses an extra phase of RPCs that the server

RPC	Purpose
open	Open a file or other object.
read	Read from a file or other object.
write	Write to a file or other object.
close	Close a file or other object.
unlink	Unlink a file or other object.
rename	Rename a file or other object.
mkdir	Create a directory.
rmdir	Remove a directory.
mkdev	Create a device.
link	Create a link to a file or other object.
sym_link	Create a symbolic link to a file or other object.
set_attr	Set various attributes of a file or other object.
set_attr_path	Given a pathname, set the attributes of that file or other object.
set_io_attr	Set the attributes of a file or other object cached on the I/O server.
dev_open	Open a connection to the I/O server for a device.
io_control	Perform an I/O control.
migrate	Start migration of an I/O handle.
release	Release a stream reference during migration.
mig_command	Transfer information about a process being migrated.
remote_call	Perform a system call on a migrated process.
remote_wait	Wait on a migrated process.
fs_release	Release a stream reference during process migration.

Table 4-1. List of blocked RPCs.

This table lists RPCs that are blocked for all clients during recovery. Most of these RPCs change the distributed cache state or other file system state and could cause cache consistency violations or other inconsistencies during recovery. For more information about these RPCs, please see [Welch86] and [Welch90]

sends to clients to indicate that recovery is over. In Spritely NFS, clients know that they cannot send certain RPC requests to the server until after recovery. When all clients have finished sending their reopen requests to the server, the server sends out a special *endrecov* RPC to each client to tell it that recovery is done. Once the client receives this RPC, it knows it can start sending new requests to the server. The advantage of this technique is that the server does not need to worry about blocking RPCs. The disadvantages are that the clients must be trusted not to issue RPCs

RPC	Purpose
echo	An empty RPC that just requests a response.
send	Used to time RPCs with different amounts of data.
get_time	Get the time of day.
prefix	Look up a prefix for a pathname.
get_attr	Get the attributes for a file or other object.
get_attr_path	Given a pathname, get the attributes of that file or other object.
get_io_attr	Get the attributes of a file or other object cached on the I/O server.
select	Perform the select system call for a file or other object.
remote_call	Perform a system call for a migrated process.
consist	Cache consistency operation sent by server to client.
consist_reply	Response to consistency call, sent by client back to the server.
remote_wakeup	Notify a remote process of some event.
reopen	Reopen a file or other object.
domain_info	Return information about a file system domain.
end_recovery	Sent by client to server to indicate end of client's recovery.
get_pcb	Retrieve a process control block from another host.
bulk_reopen	Reopen many objects at once. (A batched reopen request.)
server_reopen	Sent by server to clients to tell them to start server-driven recovery.

Table 4-2. List of Unblocked RPCs.

This table lists RPCs that are not blocked for clients during recovery. Most of these RPCs are needed as part of recovery. For more information about these RPCs, please see [Welch86] and [Welch90]

---

such as new open requests until after the server has said it is okay, and that recovery takes longer because of the extra phase of RPCs sent by the server. The designers of Spritely NFS were more concerned with simplicity, though, than with recovery speed.

In Sprite, synchronization is further complicated by the fact that clients can crash or become newly available during the recovery period. The server must update its client list accordingly. Thus, the server may both read and write the client list during recovery. In my initial implementation, client list updates and recovery from the list were both handled by one user-level process, but this allowed deadlock to occur. The recovery activities from user-level would hang waiting to get exclusive access to the device module, while the device module would hang waiting for the user-level process to acknowledge writing a new client to the list. This caused deadlock, because the user-level process could not write the new client to the list while hung. The solution to this problem was to separate the user-level daemon and recovery processes.

This combination of restrictions and complications means that there are many types of locks held simultaneously during recovery and care is required to avoid deadlock. There is an exclusive access lock (monitor lock) around some of the device code to protect the device data structures. This lock can be held as a result of an open or close device call by either user-level process, or it can be held by the recovery module as it adds and deletes clients from the client list. There is also a monitor lock in the recovery module to protect updates to the recovery status information. This lock can be held as a result of calls from the RPC system or calls from the device module to mark when hosts begin recovery or when the recovery module should block various RPCs. There are also flags that must be set and unset in the per-client recovery status information, so that the server can block some RPCs for recovering clients and all RPCs for other clients. Calls from the recovery module into the device module will acquire both the recovery and device module monitor locks. To avoid deadlock, it is necessary to restrict calls in the other direction (from the device module into the recovery module) to locations in the code where the device module does not need to acquire its own monitor lock.

4.2.5. Robustness Versus Simplicity

A final potential disadvantage of server-driven recovery is that it does not necessarily eliminate the need for client-driven recovery. Whether or not a system should have both forms of recovery available involves a trade-off between simplicity and extra robustness in the face of a network partition or a lost client list file.

Both network partitions and losing the client list will cause server-driven recovery to fail. Network partitions are rare in our environment (a local-area ethernet), but they are still possible. In a network partition, the server may believe a client has died because it cannot reach the client. The server will clean up the state for that client and remove it from the active client list. When the network partition ends, the server will fail to recover with the client, because the client is no longer listed. It is also conceivable after a failure that both copies of the client list file have disappeared or become corrupted. Server-driven recovery will fail in this case, because it does not have a list of clients to contact for recovery.

There are two choices for addressing these problems. The first solution is that we could reboot the affected clients. In the case of network partition, we would reboot the clients that were separated from the server. In the case of a lost client list, we would restart the whole system by rebooting all the clients. Although this would disrupt many users, the need to do this should be very rare, and it is a very simple solution.

The second solution is to retain the ability to perform client-driven recovery. One of the advantages of client-driven recovery is that it depends only on the clients to drive recovery, so it will still work in these situations. Retaining both types of recovery adds complexity to the system, but since all of the client-driven recovery code is used by the server-driven code, the added complexity is limited. There are two extra requirements to maintain both types of recovery. The first is a flag in a field of the server's RPC headers that states it is using server-driven recovery. The second is some extra client-side synchronization, so that clients know to wait for some period of time for the server to contact them. The extra synchronization is required to prevent clients from triggering their own client-driven recovery with a server that is performing server-driven recovery.

4.3. Results and Measurements

The two main advantages for server-driven recovery are that it eliminates cache consistency violations during and after recovery, and that it is faster than client-driven recovery. The previous chapter described the cache consistency issues; this section contains recovery performance measurements. Besides the basic timings of server-driven recovery, this section includes timings of server-driven recovery with varying numbers of clients, measurements of resource utilization, timings of recovery with and without optimizations to the recovery protocol, and timings with different amounts of file sharing between clients.

Server-driven recovery is faster than client-driven recovery, because the server can initiate recovery with clients without waiting for them to detect that it has rebooted. Otherwise the work done by the clients and the server is much the same for the two recovery techniques. Using the same testbed and basic state setup used in the previous chapter (described in Table 3-5), server-driven recovery takes 1.98 seconds on average, rather than 21. This test was run five times, with a standard deviation of 1.7%. I call this the basic server-driven measurement, because it is directly comparable to the basic client-driven measurement and to the transparent recovery measurements in the next chapter.

There are two distinct phases for server-driven recovery: handling the client list file and recovering with the clients. The time for the first phase includes the cost of starting up the recovery process, executing the *stat* system call for two client list files, reading their file descriptors, opening the two files, reading their headers to determine which to use, reading the contents of the chosen file to make sure it includes the correct number of clients, opening the kernel's "device," and writing the client list to the to the device. This phase takes 280 milliseconds. During the second phase, the kernel recovers its distributed state from the clients. This phase takes about 1.70 seconds.

For the remaining measurements in this chapter, I separate the first phase from the state recovery phase. There are two reasons for doing so. First, the time to handle the client list file is relatively static. It does not vary much with the number of clients (unless the number is so huge that the file becomes very large and takes longer to read), and it does not vary at all with the amount of state recovered. Second, separating the state recovery phase makes it easier to compare the results in this chapter with those of other chapters, since the other recovery techniques do not include the first phase.

Figure 4-4 below shows server-driven recovery timings for different numbers of clients. The figure presents the times for the state recovery phase and does not include the time for handling the client list file. This data is useful for determining how server-driven recovery should scale to a larger system than my testbed setup. The results scale linearly with respect to the number of clients, at least for the number available in this test. The slope of the curve in Figure 4-4 shows that the extra cost per client is 0.131 seconds. Thus, for a server with 40 clients and an average amount of recovery state per client, the state recovery phase for server-driven recovery should take about 5.6 seconds.

Note, though, that this result does not necessarily mean that an individual client takes an average of 0.131 seconds to recover. Clients can take much longer to recover, but their recovery activities overlap. For instance, the server can process state from one client while another client unpacks its reopen results. With a single client, no such overlap is possible, and Figure 4-4 shows that a single client takes about 550 to 600 milliseconds for state recovery.

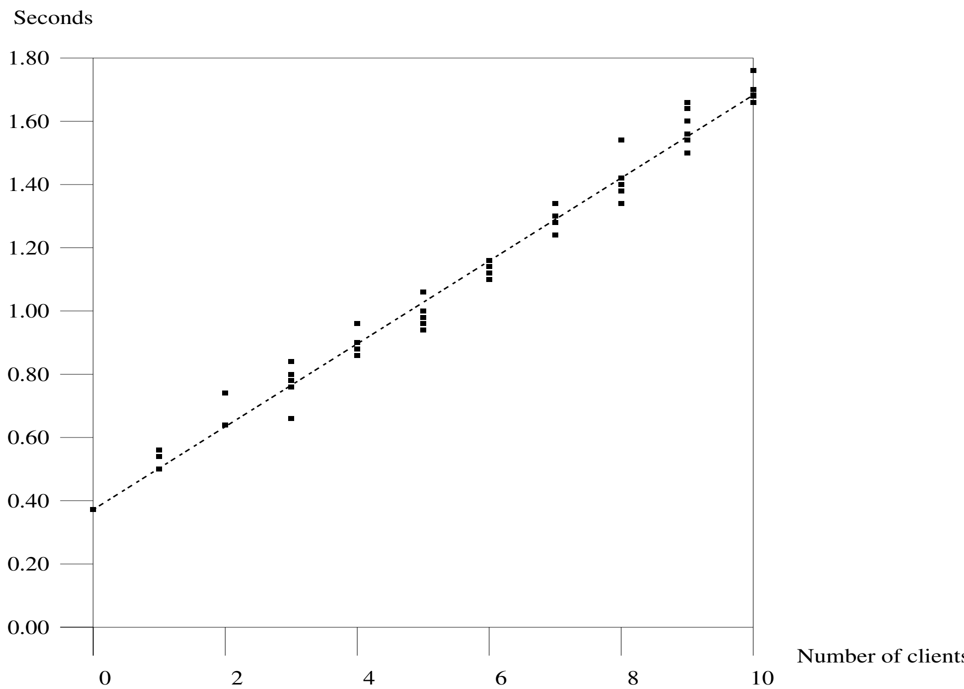

Figure 4-4. Server-driven state recovery by number of clients.

This graph gives timings for the state recovery phase of server-driven recovery versus the number of clients in the testbed setup. The X axis gives the number of clients in the timings, and the Y axis gives the number of seconds required for state recovery. (This does not include the time to handle the client list file.) There are five to six data points for each number of clients, but some of the points lie on top of each other so they are not separately visible. The dotted line gives the apparent slope of the measurements, using least-squares linear regression. The slope of the line is 0.131 seconds per client. The per-client setup for this data is shown in Table 3-5, with one modification: to make the setup more realistic, for timings of one to nine clients, only one client repeatedly truncated and wrote a file. When ten clients were available, two clients participated in this activity.

The limiting performance factor for server-driven recovery in Sprite is disk I/O. Figure 4-5 shows disk utilization during state recovery for different numbers of clients. For one or two clients, there is only limited concurrency between the disk and CPU. But as the number of clients increases, the disk rapidly becomes the limiting factor. With four or more clients, the disk is 75 to 80% utilized. The disk is actually even more highly utilized during the I/O handle part of state recovery, because it is not used at all for stream handle recovery. The last clients to recover are unable to overlap the CPU processing for stream handle recovery with other clients' I/O handle recovery.

The source of disk activity on the server is reading file descriptors. As described in section 3.1.4, a Sprite server requires file descriptors for the file I/O handles it reopens. The clients do not send

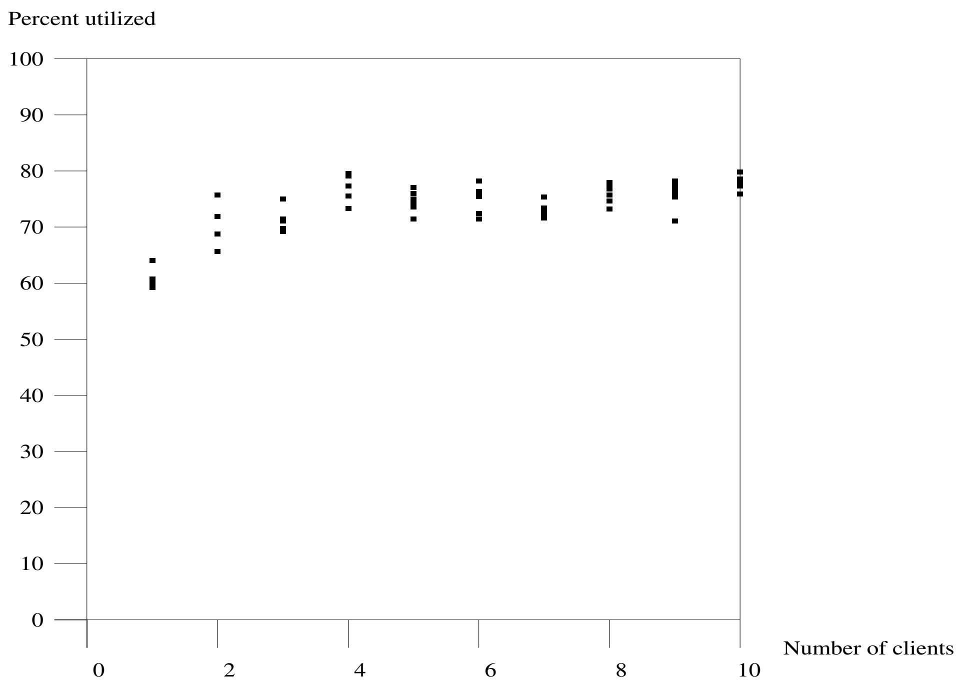

Figure 4-5. Disk utilization during server-driven state recovery.

This graph gives the percent of disk utilization during the state recovery phase of server-driven recovery versus the number of clients in the testbed setup. The X axis gives the number of clients in the timings, and the Y axis gives the percent utilization. There are five to six data points for each number of clients, but some of the points lie on top of each other so they are not separately visible.

---

the server this information, because they do not necessarily have the most recent copy. For the basic state setup used in these measurements, each client causes four to five disk reads. The average access delay on the WREN IV disk is 29.5 milliseconds, so disk I/O entirely explains the extra 130 milliseconds required per client for state recovery.

Only four or five reads are necessary per client, because the server does not need to read the disk for each descriptor referenced. LFS writes blocks of descriptors to disk, so for each read, the server reads a whole block of descriptors from disk and caches them. Thus, many descriptor reads do not cause I/O because the server finds the descriptors in its cache.

There are two implications for this I/O bottleneck. The first is that the only way to improve server-driven recovery time significantly is to reduce the amount of necessary disk I/O. This could be done by allowing clients to cache file descriptors and send them to the server as part of state recovery. Another possibility is that the server could delay reading some of the descriptors until it needs to use them. However, this second technique would not save as much I/O; the files recovered are either open or dirty on the clients, and in either case the server will soon need to examine an access or modification date for the file.

The second implication for this I/O bottle neck is that server-driven recovery time is very sensitive to the cost of the descriptor I/O. In the course of my experiments, I found a ten percent variation in server-driven recovery times depending upon the placement of the descriptors on disk. For more accurate comparisons, I have made sure that the disk placement of the file descriptors is the same throughout all the measurements in this thesis.

In contrast to disk I/O, the CPU is not a limiting performance factor for server-driven state recovery. Figure 4-6 shows the percent utilization of the CPU for recovery with different numbers of clients. With five or more clients, the CPU remains only 55% utilized.

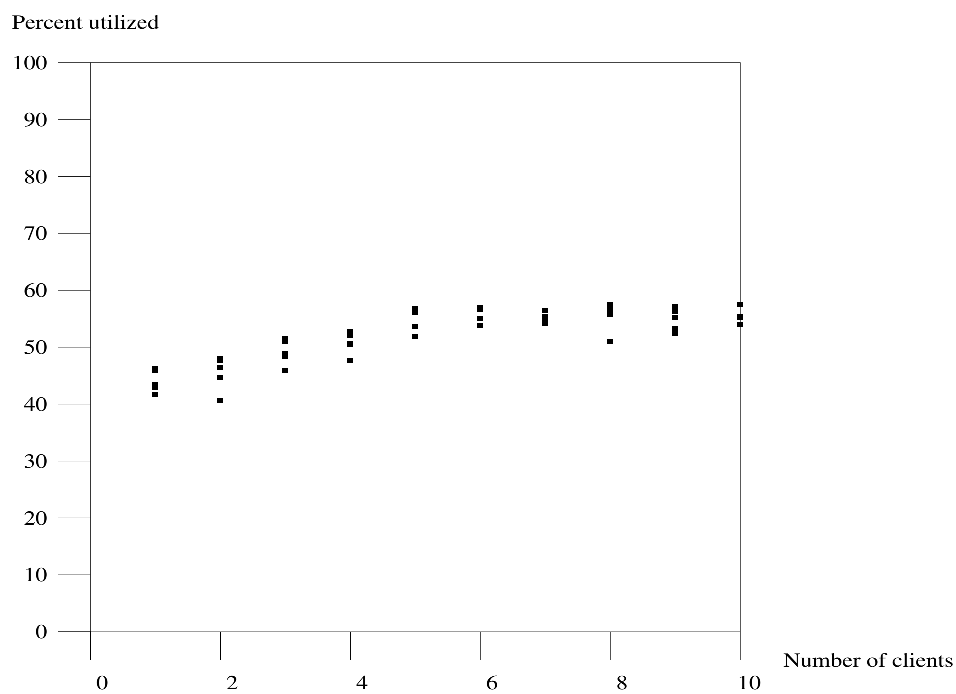

Figure 4-6. CPU utilization during server-driven state recovery.

This graph gives the percent of CPU utilization during the state recovery phase of server-driven recovery versus the number of clients in the testbed setup. The X axis gives the number of clients in the timings, and the Y axis gives the percent utilization. There are five to six data points for each number of clients, but some of the points lie on top of each other so they are not separately visible.

Table 4-3 shows the results of several other tests. The first few tests measure two optimizations already described in the chapter on client-driven recovery: batching reopen requests and eliminating unnecessary state. The last two tests show the overhead for reopening files that are not shared between clients.

A comparison of the first two measurements, *Unoptimized* and *Batching reopens*, shows the

Test	Average time (seconds)	Standard deviation	CPU utilization (%)	Disk utilization (%)
Unoptimized	16.62	0.21	85	55
Batching reopens	10.85	0.20	45	80
Necessary state	2.53	0.06	78	50
Basic server-driven	1.70	0.03	55	78
No files shared	1.90	0.05	55	82
All files shared	0.70	0.02	85	30

Table 4-3. Other server-driven state recovery tests.

This table gives the times for different measurements of server-driven state recovery from the testbed setup, excluding the time to handle the client list file. *Unoptimized* uses the same experimental setup as the basic server-driven timing, but includes neither recovery protocol optimization (batching reopen requests or reopening only necessary cache state). *Batching reopens* adds the optimization of batching reopen requests. *Necessary state* recovers only necessary cache state, but does not batch reopen requests. *Basic server-driven* uses both optimizations and is included again here for comparison. *No files shared* uses all optimizations, and the per-client state setup is the same except that no reopened files are shared. *All files shared* is the same except that all reopened files are shared. All tests were run five times.

benefits of reducing client/server communication. The two tests are the same, except that *Unoptimized* requires 790 RPCs per client for file I/O handles and 90 for streams, while *Batching reopens* packs all its file I/O handle reopens into three RPCs per client with a fourth RPC for streams. These two tests thus transfer the same amount of data and reopen the same number of files and streams on the server, but *Batching reopens* does so with about 786 fewer RPCs per client. The difference in timings between these tests shows us part of the cost on the server to process the extra RPCs. (It does not show us the entire cost, because the *Unoptimized* test is CPU bound, while *Batching reopens* is I/O bound.) The cost includes the time for the client to issue the RPC, for the server to receive it, and for the server to process it to the point of allocating an RPC server process for the request. The cost does not include data transfer time, collecting the recovery state for the request on the client, or processing the actual reopen on the server. This is because both tests reopen the same number of things. The savings from this optimization is 5.77 seconds, which is 0.58 seconds per client, or about 0.66 milliseconds per RPC. The actual cost of a round-trip empty RPC between a DECstation 5000 client and a SPARCstation-2 server, verified through direct measurement, is 0.83 milliseconds.

A comparison of the first and third tests, *Unoptimized* and *Necessary state*, shows us the benefits of reducing the amount of state recovered. As described in the previous chapter, clients keep file handles for files that are neither open nor dirty, but for which they may have clean cached data. It is not necessary to reopen these files during recovery. In the average case, this cuts the recovery state to about one-fifth what an unoptimized client would recover, eliminating 700 reopen requests per client. As seen in Table 4-3, this state reduction makes a significant difference in recovery times: 14.09 seconds, or 1.4 seconds per client.

A comparison of the tests *No files shared* and *All files shared* gives us the difference in cost for reopening shared and unshared files. Although file sharing may increase costs in normal system operation (due to cache consistency overhead), it reduces recovery time. As explained in section 3.2, the server performs less work to reopen a file already reopened (shared) by another client. *No files shared* uses the same per-client setup as the basic server-driven measurement, except that no client reopens any shared files. Each client still reopens 90 files and 90 streams apiece. But of the 90 files, none is shared with another client. Even the files repeatedly truncated and written by two clients are unshared. In contrast, *All files shared* reopens the same number of files and streams per client, but the set of files is the same across all the clients, so the server performs no extra work for nine out of the ten clients. This difference for nine out of ten clients gives a cost of 190 milliseconds per client with distinct files, and 57 milliseconds per client with shared files. Given that there are 90 file I/O handles per client, this represents an extra cost of about 1.5 milliseconds to reopen an individual unshared file versus a shared file. However, this calculation does not scale linearly with the amount of file sharing; as the test moves from sharing no files to sharing all files, it also moves from being I/O bound to being CPU bound.

4.4. Summary

Server-driven recovery is a modification of client-driven recovery with several advantages. It eliminates cache consistency violations; it is faster; and it allows the server more control over the recovery process. For these reasons, server-driven recovery is likely to be the distributed cache state recovery technique of choice for many file systems.

Server-driven recovery also has several problems. It requires more special-purpose recovery code and more synchronization and locking in the kernel. Despite the server's control, recovery speed is still vulnerable to client misbehavior. The server depends upon clients to send it their state, so it can only recover as quickly as the slowest client. Server-driven recovery in Sprite is disk I/O bound, so it can only be improved by reducing necessary I/O during recovery. Finally, server-driven recovery requires extra disk I/Os and processing to handle the client list file, so it is not as fast as transparent recovery, the subject of the next chapter.

5 Transparent Recovery

This chapter describes transparent recovery, in which the server recovers without communicating with its clients. I call it transparent, because the clients do not participate in recovery. Instead, client RPCs to the file server simply hang until the server finishes recovery and re-enables its RPC service. At this point it services the RPCs, and the clients continue processing. This recovery technique avoids client/server communication during recovery by keeping the distributed file system state in stable storage on the file server. The file server can then regenerate the state after a failure without contacting its client workstations for information. By eliminating communication, transparent recovery also eliminates the server's dependency upon its clients for recovery speed and its need for a client list file. With no client list file, transparent recovery executes fewer disk I/Os and is thus the fastest form of distributed state recovery. For ten clients in the testbed setup, this technique takes only 1.5 seconds compared to 2.0 for server-driven recovery. Transparent recovery scales linearly with the number of clients. Like server-driven recovery it is disk I/O bound in Sprite and requires an extra 130 milliseconds per client, all of which is spent reading file descriptors from disk. However, transparent recovery presents more opportunities to reduce the amount of necessary disk I/O.

The key element of transparent recovery is the *recovery box*, which is the stable storage the server uses to hold its distributed state. What is most interesting about the recovery box is that it turns main memory into a form of stable storage; the recovery box is just an area of the server's main memory that is preserved across failures. During normal execution, the server's operating system inserts backup copies of file system and cache state information in the recovery box. After a failure, the system retrieves these items from the box and uses them to regenerate the distributed state information. The recovery box enables these activities to proceed at main-memory speeds.

The recovery box interface is designed to avoid memory corruption, since this would render its contents useless. When the system inserts an item into the recovery box, it also inserts a checksum for the item. After a failure, the system examines the checksums for the items it retrieves. If the software checksum detects corruption, the system clears the recovery box memory and reverts to some more traditional form of recovery or a complete system reboot, as described later in this chapter. Fortunately, error statistics presented in this chapter show that failures are unlikely to corrupt the recovery box. Also, the recovery box memory can be write-protected in hardware.

Besides its performance benefits, an advantage of the recovery box is that it can be used for safe storage by other parts of the operating system and by user-level applications such as databases.

Later in this chapter I describe how a distributed version of the POSTGRES database [Stoneb86] uses the recovery box for fast crash recovery. The recovery box is generally useful for any system that wishes to preserve state information across failures, if it is possible to isolate the state updates, and if the state items have the following properties. The items to store should be 1) expensive to regenerate from scratch, 2) small, 3) updated too frequently to store on disk, and 4) unlikely to propagate the error that caused the system crash.

Despite these advantages, there are at least four problems with transparent recovery. First, transparent recovery requires more special-purpose code than the other recovery techniques; the additional code is needed to implement the recovery box, maintain its contents, and rebuild the server's file system state from its contents after a failure. Second, transparent recovery is prone to inconsistencies between the clients and the server; while avoiding recovery communication between the server and clients reduces recovery time, it also removes an opportunity for server and clients to re-synchronize their state information. Third, the recovery box is difficult to incorporate into a system in which operations that change the file system state, such as opens and closes, are hard to isolate or handle atomically. It is often hard to isolate these operations in Sprite, and this is likely to be true in many other distributed file systems. Finally, transparent recovery does not necessarily eliminate the need for a communication-based form of recovery, depending on the desired level of robustness. Recovery box corruption is unlikely, but if it occurs we may wish to recover the system with client-driven or server-driven recovery, rather than reboot all the clients.

This chapter first motivates the use of the recovery box for storing distributed state information locally on the file server. As part of this first section I list other possible storage techniques and explain why they are less satisfactory. I then list the failure statistics that indicate the recovery box memory should be free from corruption after most failures. Section 5.2 describes the design and implementation of the recovery box. This includes a description of the interface to the recovery box functions, a description of the internal recovery box structure, and a list of implementation shortcomings. Section 5.3 explains how Sprite incorporates the recovery box for transparent distributed state recovery. Section 5.4 gives measurements of transparent crash recovery times and measurements of file system performance degradation due to maintaining the recovery box during normal execution. Section 5.5 explains some overall disadvantages of transparent recovery and the recovery box. Section 5.6 lists two trends in file system design and their implications for using the recovery box: asynchronous operations and stateful file servers. Section 5.7 explains how applicable the recovery box is to other systems. The final section describes recovery box use for distributed crash recovery in a user-level application, the POSTGRES database, and includes application-level performance measurements.

5.1. Motivation for the Recovery Box

To avoid communication with clients for distributed state recovery, the file server must store its distributed state locally. There are two requirements for this local storage: high performance and persistence across most failures. This section motivates why I've chosen main-memory storage, in the form of the recovery box, as most likely to meet these requirements.

Keeping a persistent copy of the distributed state in the server's main memory means that the file server can update this stored cache state information at main memory speeds, avoiding much performance degradation during normal operation. The recovery box is just an area of memory set aside for access only by the recovery box code. To insert an item of state information into the recovery box, the file server kernel simply calls the recovery box procedure *InsertItem* with the item as a parameter. The recovery box code copies the item into its area of memory. After a crash the kernel recovers the state information quickly from its own memory by calling *ReturnItemArray*. The recovery box code copies the array of preserved items from its memory back out to its caller. This interface is described in more detail in section 5.2.1.

This technique contrasts with more traditional ones that store the distributed state on the server's disk or in the memory of a backup machine. Section 5.1.1 lists some of these more traditional techniques and explains why they are slower or more complex.

The second requirement for the recovery box is to preserve the distributed state across the most common types of failures. The recovery box memory can be implemented with non-volatile RAM for persistence across power failures, but further efforts are required to ensure that its contents are not corrupted by errors in the kernel code or by hardware glitches. The recovery box is designed to detect memory corruption so that the system does not try to recover incorrect distributed state. Whenever an item is inserted in the recovery box, the recovery box code also calculates and inserts a checksum for the item. When the items are retrieved, the recovery box examines their checksums and reports a failure if they are not correct.

While this technique detects the existence of memory corruption, the question remains as to how often such corruption is likely to occur. If it occurs often, then the recovery box will fail to provide fast recovery after most failures. Storing the cache state information in main memory is only reasonable if we can protect and preserve that area of memory across failures. Sections 5.1.2 and 5.1.3 give two reasons why we can preserve the recovery box memory across most failures. First, statistics about the types and distribution of system failures show that the recovery box memory is unlikely to be corrupted by failures. Second, we can also use hardware write-protection to prevent recovery box corruption.

5.1.1. Storing Distributed State

This section describes two alternate ways to store the distributed state information: on the server's disk, or in the main memory of another machine. I have not implemented these techniques, because they have disadvantages that led me to believe that the recovery box is preferable.

5.1.1.1. Disk Storage

The simplest way to maintain the distributed state information is to keep a copy of it on the server's disk, as is done with the client list for server-driven recovery. Whenever a client requests an operation that changes the distributed state information, the server can record the change in its disk copy before responding to the client. Recording the change synchronously (before responding to the client) ensures that the state remains consistent between the server and client in the event of a server crash.

Unfortunately, the state needed for file cache consistency changes too frequently to be updated synchronously on disk without performance penalty. As explained in chapter 2, the distributed state information changes on the server whenever a client workstation opens or closes a file. A synchronous disk write for each change adds too much latency to file open and close operations. The combination of a remote open and close operation on a Sprite client takes only about 3.5 milliseconds, while a disk write takes 15 to 30 milliseconds, depending on the disk. In addition, the cache state information is sometimes updated with a frequency that would require too many disk accesses. On average, the server sees two to four pairs of file open/close operations every second, which would require six to eight extra disk accesses. During periods of bursty file system activity, the server sees as many as 50 pairs of open/close operations in a second, which would require one hundred disk accesses in a second. This is unreasonable for the average server with only five or six disks, even if the updates are divided evenly among the disks and handled in parallel.

Storing the server's state information on disk thus requires some more complicated approach, such as batching together state changes resulting from many client requests into fewer disk writes. This amortizes the cost of the disk I/O on the file server. But the performance overhead seen by clients depends upon whether these batched updates are done synchronously or asynchronously, with respect to the clients' RPC requests.

If the server performs the I/O synchronously, before responding to the client RPC request, it will not lose recent state updates if it crashes. However, this delays response to all the RPCs in the batch by at least the length of a disk write, increasing the overhead of all client open requests by an order of magnitude.

Performing the batched updates asynchronously with respect to the client RPCs allows the server to respond to the client RPCs immediately, but it is more complicated. It is more complicated because the server and clients must do extra work to ensure that state information does not become inconsistent. The state can become inconsistent if the server crashes after replying to the client RPCs but before it writes the updates to disk. To combat this problem, we can use a combination of server-driven and transparent recovery. Clients must save enough information about their requests to resend the necessary information to the server if it crashes before recording the updates. If the server forces all updates to disk every 30 seconds, clients can discard information about requests older than 30 seconds. During recovery, the clients resend the state updates for the last 30 seconds to the server. The server then updates its distributed state information before allowing the clients to continue normal operation. The disadvantage of this scheme is that it is not much better than server-driven recovery, and it isn't transparent recovery, because it still requires communication during recovery. All it does is reduce the amount of state sent to the server. Since server-driven recovery is simpler and almost as fast, we might as well skip storing the state information on the server's disk altogether.

5.1.1.2. Backup Machine Storage

Another alternative to the recovery box is to keep a complete backup copy in the main memory of a machine other than the server. After a crash, the file server retrieves the state information from this backup machine. This technique requires only a few RPCs, and only with the backup machine. This is different from client-driven and server-driven recovery, because those techniques spread different parts of the state information across many machines.

If we store the backup state information on another machine, we must be able to update it whenever the server's state changes. The choices for accomplishing this are similar to those for storing the state on disk, but the performance costs are different. One way to do this is to require the file server to update the backup's state information every time a client makes an RPC request that changes the distributed state. The server issues an RPC to the backup and waits to receive its reply, before responding to the client. This technique is similar to that used by HARP [Liskov91] to make its file system updates on backup servers.

There are several advantages of this technique. Updating the information across the network to the backup machine's main memory is faster than updating it to the server's disk but still provides the same permanence in the case of a server failure. Each client request that causes an update of the cache state information suffers less than a millisecond's extra latency, compared to the tens of milliseconds required by a disk write. Also, accessing the saved state information for crash recovery requires the server to communicate with only one machine, rather than with a whole set of clients as during client-driven or server-driven recovery. Additionally, storing the information in the memory of a machine other than the server avoids the complexity of the checksums or write-protection used by the recovery box.

The disadvantage of this technique is that it still incurs a substantial performance overhead during normal operation of the system. Except for storing the state on disk, it is slower than the recovery techniques previously described. This is because the updates to the backup machine's main memory require an extra round-trip RPC for each relevant client RPC. Using DECstation 5000's or SPARCstation-2's in Sprite, this round-trip RPC adds about a millisecond of latency to every file open or close. This is 25% of the cost of a remote file open and close combined. A system running client-driven or server-driven recovery does not suffer this performance degradation during normal operation, and a server running transparent recovery only needs to update its own main-memory.

A second way to maintain the backup machine's state information is for the backup to listen on the network for client RPC requests of the file server. Whenever a client makes a request that could change the cache state information on the server, the backup machine updates its copy of the state information to reflect that change. The advantage of this technique is that the file server and the backup machine need not communicate during normal operation, so RPCs from clients are not delayed. The big disadvantage is that the backup machine's copy of the state information may become inconsistent with the server's. The server has no way to know if the backup has noted the RPC request and successfully applied the state change before the server responds to the client. For example, the backup machine could fail to notice an RPC if its network interface is congested. Since the backup machine is not the primary destination of the RPC, the client will know only that the primary server successfully received the request, and it will not resend the message. If the RPC contains a file open request, the backup will fail to record that the client is caching the file.

Another disadvantage with both these techniques for storing state information on a backup machine is the extra complexity for handling backup machine crashes. If the backup machine crashes, the server must send its state to another backup. If the server itself crashes before it finishes sending this information to the new backup, the system cannot recover without using server-driven or client-driven recovery. Storing the state information on multiple backup machines would reduce this vulnerability, but would also further degrade overall performance.

5.1.2. Error Statistics

The most important issue for storing the recovery box in main memory is whether it will survive most server crashes intact. Statistics on the frequencies of different types of system outages indicate that this should be true for three reasons. These statistics were collected by Jim Gray [Gray90] and Mark Sullivan [Sulli93a] in Tandem, MVS, and UNIX systems. First, the majority of failures are due to software failures rather than permanent hardware problems that require long downtimes to fix. Second, the majority of software errors do not corrupt memory. Third, of those errors that corrupt memory, most do not affect random memory locations. Rather, the error corrupts an area local to the data structure it intended to modify. This means that careful address and size checking for updates to items in the recovery box will catch most of the corruption errors before they do damage. Together, these statistics suggest that most failures will leave the recovery box data undamaged, as detailed below.

Published data about the frequency of different kinds of outages is scarce, but a study of Tandem systems shows that faulty software is now responsible for most failures [Gray90]. Over time, Tandem systems have experienced fewer outages caused by hardware failures. The number of software failures, on the other hand, has remained constant. Table 5-1 shows the percentages of each source of outage. In 1989, software errors accounted for 62% of Tandem system failures, while only 7% were caused by hardware. In other environments as well, software errors and other transient errors account for the majority of failures [Lin90].

Outage sources	Percent in 1985	Percent in 1989
Software failures	34	62
Operator errors	9	15
Hardware failures	29	7
Environment failures	6	6
Scheduled maintenance	19	5
Unknown	4	5

Table 5-1. Distribution of outage types.

This table shows the distribution of types of outages by fatal fault occurring in Tandem systems between 1985-1990 [Gray90]. Software failures are caused by errors in the software. Operator errors are mistakes made by humans responsible for maintaining the machines. Hardware failures are caused by problems with the hardware. Environment failures are caused by floods, fires, and long power outages. The table also includes outages due to scheduled maintenance and unknown causes. The overall failure rate has decreased since 1985 but has held steady since about 1987. The total percentages do not all add to 100, due to rounding.

The direction of change in these failure statistics is part of the motivation for the fast recovery approach. Without redundant hardware, the recovery box and other fast recovery approaches are not able to recover from hardware failures, but hardware is becoming more reliable and software is now the greatest source of unreliability. This means that we should be able to handle the majority of failures by concentrating on software failures and other failures that do not cause permanent hardware damage.

Even if most system outages are due to software rather than permanent hardware failures, the question remains as to whether the software failures will damage the contents of the recovery box. Two studies that categorize the types of operating system software errors indicate that most software errors will not corrupt the recovery box. These two studies are summarized in Table 5-2 and are described below.

The BSD UNIX study [Sulliv90] divides errors into *synchronization* (47%), *exception-handling* (12%), *addressing* (12%), and *miscellaneous* (29%) errors. Synchronization errors include problems such as deadlock or waiting endlessly for a signal from some event. Exception-handling errors are those that occur in code for handling other errors, including transient hardware errors. Addressing errors are those for which the software uses an incorrect memory address. They are the errors most likely to corrupt memory, since the software may modify bytes at the wrong location. For UNIX, addressing errors account for only 12% of errors.

The MVS study [Sulliv91] classifies errors in terms of low-level programming errors, of which 41% were *control* problems, 30% were addressing errors, 8% were *data miscalculations*, and 21% were miscellaneous errors. Control errors include such problems as deadlock, in which the program stops without corrupting anything but transient state. Data miscalculations include errors in which the wrong variable is used or a function returns the wrong value. Most of the errors classified as miscellaneous are related to performance or denial of service. In MVS as well as in UNIX, addressing problems are not responsible for the majority of failures.

Error classes	BSD UNIX	MVS
Addressing-related errors	12	30
Control-related errors	NA	41
Data miscalculation errors	NA	8
Synchronization-related errors	47	NA
Exception-related errors	12	NA
Miscellaneous errors	29	21
Total	100	100

Table 5-2. Software error type distributions.

This table shows the distribution of software errors analyzed by Mark Sullivan in studies of error reports from the IBM MVS and 4.1/4.2 BSD UNIX operating systems [Sulli93a] [Baker92a]. The results columns give the percentage of errors that fall into each category. The two studies used different classification schemes, but both list addressing errors – the errors most likely to corrupt recovery box memory. The other error classes are described in the text. *NA* means not applicable due to differences in the classification schemes.

In addition, the MVS study shows that most memory corruption due to addressing errors is local to the data structure being manipulated. As shown in Table 5-3, at least 57% of addressing errors either corrupt the data structure the operating system intends to modify or the memory immediately following the data structure. Only 19% of the MVS addressing errors covered in the study damaged parts of the system unrelated to the one where the error occurred. For example, a common type of addressing error in MVS is *copy overrun* in which a copy transfers too many bytes from one buffer into another, overwriting the data structure that follows the intended destination. A second common addressing error, called an *allocation management error*, occurs when the operating system continues to use a structure after deallocating it. Careful size and bounds checking and

Location of damaged area	Percent of MVS errors
Near intended area	57
Anywhere in storage	19
Not evident from report	24
Total	100

Table 5-3. Data corrupted by addressing errors.

This table shows the relationship between the location of data corrupted by an addressing error in MVS and the location of the intended modification. Usually data corruption occurs near the data owned by the faulty code. In 24% of MVS error reports studied, this location information was not evident.

careful memory allocation will reduce the likelihood of corruption from these local address errors.

These statistics indicate that random addressing errors are rare, which is fortunate, since they are the type of addressing error most likely to corrupt the recovery box. Random addressing errors are the most dangerous, because they are the hardest to guard against. Any errant code in the operating system kernel could mistakenly write to the recovery box area of memory. Because these statistics were gathered across different systems, we cannot combine them with accuracy, but we can get a rough idea of the likely distribution of errors. If about two thirds of errors are software errors (Table 5-1), and about 12 to 30% of these are addressing errors (Table 5-2), and 19 to 43% of these are random addressing errors (Table 5-3), then only one to seven percent of overall failures are due to an addressing error that corrupts random memory. Figure 5-1 is a pie chart showing this calculation of random addressing errors. Using measurements from Internet sites [Long91] that indicate that UNIX machines fail on average once every two weeks, we can expect one to two failures per year due to random addressing errors. Even then, the error does not necessarily mean the recovery box will be damaged.

With these statistics in mind, I designed an interface to the recovery box that reduces the possibility of corruption from software failures. For example, corruption is less likely if we use the recovery box to store backup copies of critical data structures, rather than the primary copies that callers access directly. The recovery box interface does not allow callers to access its memory directly. The interface does strict length checking when items are copied into the recovery box to prevent copy overruns. The recovery box must also do very simple memory allocation and must not allow the software calling it to allocate or free recovery box space directly.

Unfortunately, the failure statistics do not give us enough information to protect the recovery box against complex faults involving *error propagation*. If the system fails due to a logical error in one of its data structures, and if this data is stored in the recovery box, then the system will suffer

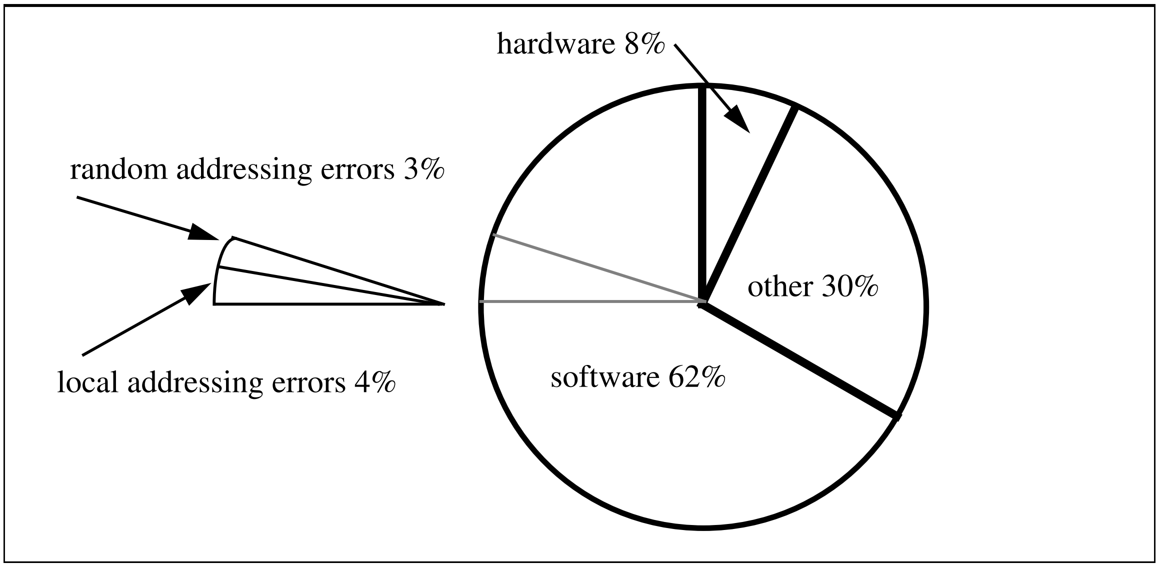

Figure 5-1. Percentage of random addressing errors.

This pie chart gives a very rough estimate of the percentage of errors that are likely to corrupt random memory. About two-thirds of errors are software errors, about 12 to 30% of these are addressing errors, and about 19 to 43% of these are random addressing errors. Random addressing errors are the ones likely to corrupt the recovery box. This estimate is very rough, because it combines data from several different studies of system failures.

the same failure after recovery. The only way we can protect the system from this problem is to choose data structures for insertion in the recovery box that have proven themselves over time to be relatively error-free. In commercial systems using a recovery box, data gathered from error reports would help indicate which data structures are likely to propagate errors.

5.1.3. Write Protection

Another way to protect the recovery box from memory corruption is to write-protect its area of memory. We can set the protection bits in the hardware pages containing the recovery box to disallow any writes to those pages. Only the recovery box code, in one place, should execute the instructions necessary to make the pages writable, make the update, and re-protect the pages. There is no guarantee that the recovery box code itself, between making the pages writable and re-protecting them again, will not suffer an addressing error and corrupt the memory. But the amount of code executed between making the pages writable and re-protecting them is very small and can be checked with high confidence. Write protection means that other parts of the kernel code will not be able to write to the recovery box accidentally without suffering a write-protection fault.

Write-protecting the recovery box adds some performance overhead to recovery box updates, because changing the hardware page protection on some architectures requires flushing translation look-aside buffers or virtually addressed caches. However, the overhead seen on the SPARCstation-2 is low and is described in this chapter in section 5.4.2 on file system overhead.

5.2. Design and Implementation

The goal of the recovery box is to provide fast insert, delete and retrieval operations while protecting the stored data against common addressing and memory management errors. Both the operating system kernel and user-level applications can use the recovery box, but user-level applications must execute their operations through a system call interface. This section first describes the interface to the recovery box and then its internal structure.

5.2.1. Interface

The recovery box interface should help Sprite and its application programs manage backup data without exposure to some of the common software errors that corrupt main memory. For this reason I chose a structured and relatively inflexible interface, rather than a more flexible one in which clients directly allocate and manage data structures in the reserved area of memory. This structured interface provides more opportunity to avoid and detect recovery box corruption.

In this interface, each item belongs to a type, and users of the recovery box must pre-define the types of items they use. When a new type is defined, the recovery box assigns it a unique *typeID* and returns this ID to the caller so that the caller can refer to the type in the future. All items of the same type have common characteristics, such as size and checksum calculation routine. Maintaining item size in the recovery box makes it easy for the interface to detect common errors that cause memory corruption, such as writing past the end of a data structure.

Callers of the recovery box must explicitly insert, delete, and update recovery box items. This makes it easy for the recovery box to detect memory management errors. For example, the recovery box does not need any sort of complex garbage collection, because callers must explicitly delete items when done with them. When a client creates a new item, the recovery box manager generates a unique *itemID* and returns this to the caller so that the caller can refer to the item in the future. An itemID consists of the item's typeID and an *itemNumber.*

The Sprite kernel and its user-level applications use the same interface to the recovery box, except that applications call the recovery box routines via a system call. The major difference for user-level applications calling recovery box functions is that they have no control over the order in which other types and items are inserted in the recovery box. For this reason, they have no control over the typeIDs and itemNumbers assigned to their items. When an application program begins fast recovery, it must be able to find its recovery box items to regenerate its state from them. This means the application must be able to remember, across reboots, the typeIDs and itemNumbers assigned to its items by the recovery box. To solve this problem without requiring the application to store a list of these IDs on disk, the recovery box allows the application to choose its own well-known typeIDs and itemNumbers and to map from them to the system-assigned IDs. The application can specify its *applicationTypeID* or *applicationItemNumber* when it initializes a type or inserts a new item. On recovery, the application maps from its application IDs to the system IDs once, at initialization, and not every time it accesses recovery box items. Allowing applications to specify their own IDs also facilitates sharing of recovery box items between cooperating UNIX processes. The recovery box functions return an error to the application if it chooses an applicationTypeID or applicationItemNumber already chosen by another caller.

Table 5-4 lists the available recovery box functions. The first two functions operate on types. To initialize a new type of item, the system or application must call *InitType*. As parameters to this

Category	Name	Parameters	Results
Operations on types	InitType	itemSize, maxNumber, applicationTypeID (optional), ChecksumFunctionOrFlag	typeID
	DeleteType	typeID	none
Operations on single items	InsertItem	typeID, itemPtr, applicationItemNumber (optional)	itemID
	DeleteItem	itemID	none
	UpdateItem	itemPtr, itemID	none
	ReturnItem	itemID, itemBufferSize, doChecksum	*itemPtr*
Operations on multiple items	InsertItemArray	typeID, numberOfItems, itemBuffer, applicationItemtNumberBuffer	*itemIDBuffer*
	DeleteItemArray	numberOfItems, itemIDBuffer	none
	UpdateItemArray	numberOfItems, itemBuffer, itemIDBuffer	none
	ReturnItemArray	typeID, itemBufferSize, itemIDBufferSize,applicationItem NumberBufferSize (optional)	*itemBuffer, itemIDBuffer, applicationItemNumberBuffer (optional)*
ID mapping operations	GetTypeIDMapping	applicationTypeID	typeID
	GetItemIDMapping	typeID, applicationItemNumber	itemID

Table 5-4. Recovery box operations.

This table lists the operations available through the recovery box interface. The first set of functions applies to item types; the second set applies to individual items, and the third to multiple items. The last set of functions map from an application's chosen typeID or item numbers to those used by the recovery box. Results in italics are actually parameters that the caller supplies to the function into which the recovery box copies the results. In this way, the recovery box avoids returning any internally-used addresses to its caller.

function, the caller specifies the maximum number of items that can be valid simultaneously, the item size, an optional applicationTypeID, and a flag that signals whether checksums should be calculated and stored for items of this type. When called by kernel code, the checksum flag may instead be a pointer to a specialized checksum routine for the type. Otherwise the recovery box uses a generic checksum. Stating the maximum number of items ahead of time is a potential problem for applications that do not know how many items they will need. However, it makes memory management much easier. *DeleteType* frees up an item type, but I do not expect this function to be called often, except perhaps when a new application is being tested.

Interface functions operating on individual recovery box items include *InsertItem*, *DeleteItem*, *UpdateItem*, and *ReturnItem*. To insert an item in the recovery box, the caller provides the item's typeID, a pointer to the data for the item, and an optional applicationItemNumber. DeleteItem frees the space in the recovery box allocated for an item. To update the contents of an item in the recovery box, the caller must provide the itemID and a pointer to the new data for the item. ReturnItem returns a copy of the specified item in a buffer provided by the caller.

Additional interface functions, such as *InsertItemArray* and *ReturnItemArray*, allow applications to operate on multiple items, avoiding the system call overhead that would be incurred by multiple operations on individual items. To insert multiple items the caller must provide the typeID, the number of items it wishes to insert, an optional array giving the application's itemIDs for each item, and an array of items to insert. The function returns an array giving the new system-assigned itemIDs for the inserted items. ReturnItemArray returns copies of all the items for an item type and an array of itemIDs. If there is insufficient space in the buffers given to it as parameters, the function returns an error and an indication of the amount of room required for each of the buffers.

There are two safety issues to note in the organization of parameters and results of these functions. First, the recovery box never returns a pointer to a result. In this way, it avoids passing pointers to its internal memory back to its caller. This makes it less likely that the caller will accidentally access recovery box memory directly. Instead, the caller must always supply a parameter that is a pointer to its own buffer for results. The recovery box merely copies the results into the buffer. This incurs an extra copy, but it provides extra safety.

The second safety issue is that the caller must also indicate as a parameter the size of any buffer it supplies to the recovery box. The caller supplies both the size of the buffer and the number of objects it should be able to contain. The recovery box does its own calculation given the item type's size and checks this against the buffer size supplied by the caller. It reports an error if the caller's buffer is not large enough. Although the caller may not supply a buffer as large as it claims, this extra check helps eliminate mistakes due to incorrect computations in the caller's code. The recovery box code is less likely to make these mistakes; it is likely to be exercised more frequently, so mistakes will be exposed early on. It is important that it be well-tested.

Finally, there is another technique to reduce mistaken accesses to recovery box memory. Before returning to its caller, the recovery box code should zero out the part of the system stack used for any recovery box functions. If this is done, the caller will not find recovery box addresses in uninitialized stack variables during its next function call. I have not implemented this technique.

5.2.2. Recovery Box Structure

The recovery box implementation is designed to provide fast atomic insert and delete operations and fast access to items and their type information. It is important that the operations be atomic to avoid leaving the recovery box in an inconsistent state due to a failure. For example, atomic updates ensure that a system failure that interrupts an update will not cause checksum failure, because the old value will be left intact.

Figure 5-2 shows the recovery box layout in memory. There are three sections of the recovery box: a header, followed by an array of per-type information, followed by the item area. In the item area there are two arrays for each item type: an array of per-item information and an array storing

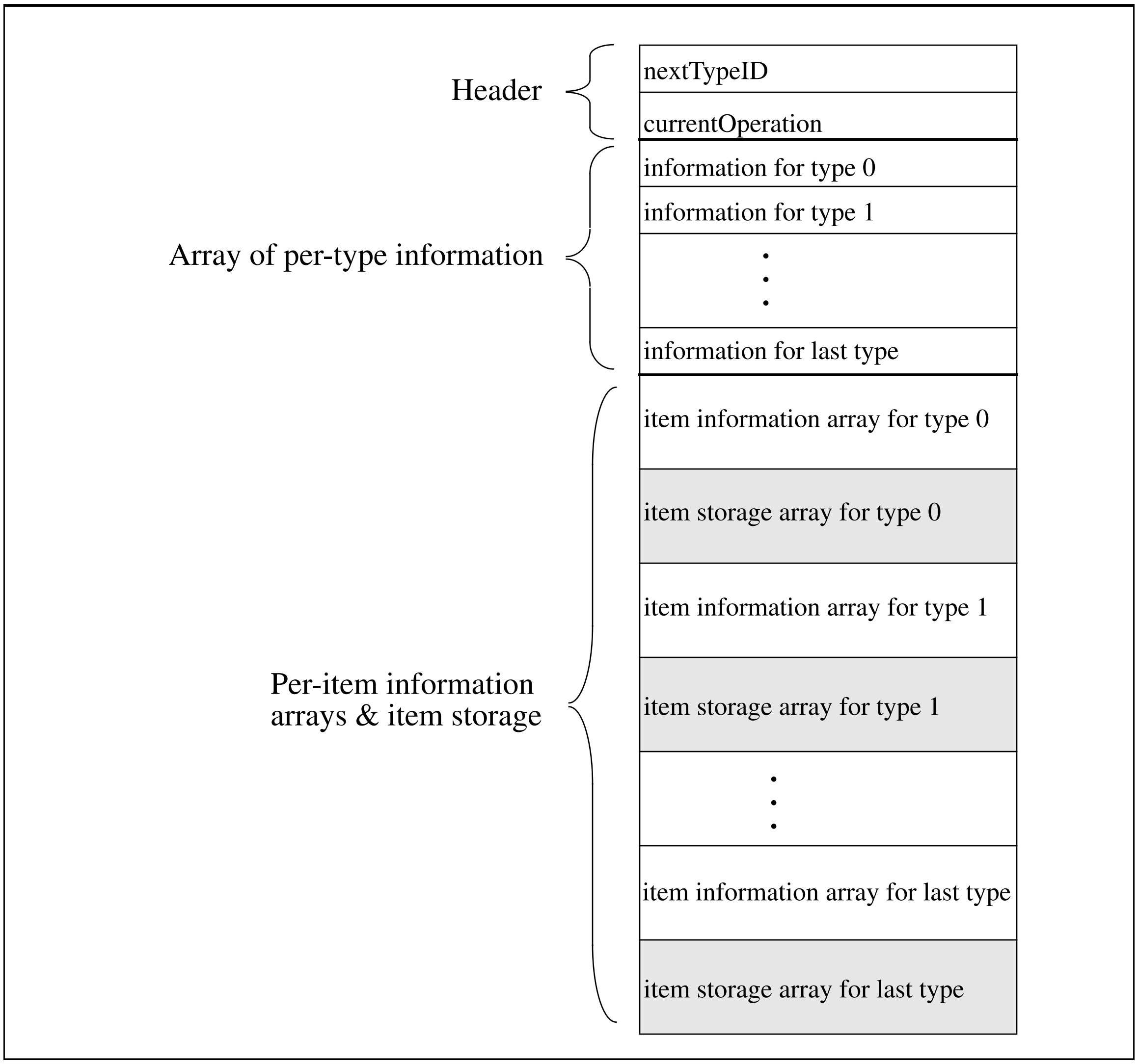

Figure 5-2. Layout of recovery box in memory.

The recovery box layout in memory starts with a header that gives the next typeID to allocate, and provides a code for the current operation. The current operation code is used to ensure atomicity of recovery box insertions, deletions, and updates. The header is followed by an array that gives information about each type of item stored in the table. The per-type information is followed by the per-item information array for the first item type. Following the per-item information array for each item type is the array of the items themselves, shown as a shaded entry in the figure.

the items themselves. As described below, a portion of the per-item information array implements a stack of the free items for each type, providing constant-time item insertion and deletion. The type information, item information and item storage are all implemented as arrays for fast access given an array index; the recovery box typeIDs and itemNumbers are actually indices into the per-type and per-item information arrays, respectively. (Since small integers are more likely than large ones to be used accidentally as incorrect IDs, a better implementation would be to add a positive offset to all indices to create their corresponding IDs. I have not implemented this.)

The header at the very beginning of the of the recovery box contains information that must be saved across fast reboots. The first field in the header, *nextTypeID*, gives the index of the next typeID to be allocated. The second field in the header specifies the current operation on the recovery box in order to ensure that insert, delete, and update item operations are atomic. (Other functions, except those operating on multiple items, are already assured of being atomic in this implementation, because they do not modify data.) At the beginning of an insert, delete or update operation, this field is set with a code for the current operation. In the case of a delete or update operation, it also includes the target itemID. The field is not cleared until the operation completes, making it possible to detect and back out of incomplete operations. At the beginning of an update, the original value of the item is copied to an extra item space at the end of the item storage array. If the machine crashes during the update, the original value of the item can be retrieved.

itemSize
maxNumItems
currentNumItems
applicationTypeID
firstFreeItem
address of item info array
address of item storage array
checksumFunc (optional)

Figure 5-3. Contents of type array.

This figure shows the contents of an entry in the per-type array. This array lists, for each type, the size of the items, the maximum number of items that can be allocated, the current number of allocated items, the itemNumber of the first free item, the memory address for the per-item information array, the memory address for the item storage array, and a possible checksum routine.

The recovery box header is followed by an array, accessed by typeID, that gives information about each item type. Figure 5-3 shows the information stored in an entry of this per-type array. Each entry lists the item size, the maximum possible number of items, the current number of items, the application typeID, if any, the index in the item storage array of the first free item, the memory address of the per-item information array, the memory address of the item storage array, and a pointer to a checksum routine.

The caller of InitType provides a checksum routine, if the caller is part of the system code. For implementation reasons, it is currently not possible for a user-level application to provide its own type-specific checksum routine. The checksum routine must take as a parameter the size in bytes of the item to be checksummed. However, the checksum routine used for the type containing the Sprite kernel's file handle items is unrolled to optimize for the 52-byte file handle items. If a type's checksum routine field is zero, then no checksums are calculated for that type's items.

The per-item information arrays are used for item allocation and for storing checksums and application itemNumbers. An alternative to grouping the per-item information into an array is to preface each item with a header that includes its per-item information. I chose to separate the items and per-item information to make the ReturnItemArray function faster and easier. All that function needs to do is copy a contiguous area of memory to return to its caller. Separating the two arrays also makes memory management and debugging easier.

Figure 5-4 shows the contents of an entry in an item information array. Each entry in the array consists of two fields. The first is only valid if the item has been allocated. It contains the application's choice of itemNumber, if the application provided one while inserting the item, or a -1 if there is no application itemNumber. The second field has a different meaning depending on whether the item has been allocated or not. If the item has been allocated with a checksum performed on it, the field contains the checksum result. If the item has not been allocated, the field contains the index of another unallocated item in the array, thus implementing a stack of free items. For the last item in the stack this field is -1. The index of the first free item on the stack is stored in the per-type information. Upon deleting an item, its itemNumber is added to the top of the stack. The stack of free items makes it fast to find free spaces in which to insert new items.

if allocated	if free
application item # (-1 if none)	unused
checksum	index of next free item

Figure 5-4. Contents of item information array.

This figure shows the contents of an entry in the per-item information array. This array lists, for each item, the application itemNumber, if one exists, and a possible checksum value for the item. If the item is not allocated, the checksum field instead gives the itemNumber of the next unallocated item.

Storing the application itemNumbers in the per-item array provides quick mapping from the recovery box system's itemIDs to the application's itemIDs; however, mapping from application IDs to system IDs is not particularly fast. This is why none of the functions in the recovery box interface take the application IDs as a parameter, except for the functions that return the system ID given the application ID. Applications can maintain their own hash tables to do this mapping quickly, but I leave this functionality outside of the recovery box for the sake of simplicity.

Besides considering the internal memory structure of the recovery box, we must also consider where it is placed in memory. First, the system must be able to find the recovery box's memory location after a reboot. For this reason, the recovery box is always allocated at the same virtual address. Keeping the recovery box at the same virtual address also ensures that absolute addresses inside the recovery box (specifying the per-item information arrays and storage areas) remain valid across fast reboots. Finally, the virtual address of the recovery box must also map to the same physical address upon every fast reboot so that the memory location actually contains the recovery box data. The operating system must avoid clearing and initializing this area of physical memory on a fast reboot.

The recovery box is located in a special area of the kernel text segment to minimize damage due to memory corruption. The memory pages for the recovery box are marked writable as well as readable, in contrast to the rest of the text segment. I chose the text segment because it reduces the sources of possible system address errors that could overwrite the recovery box. Except for the recovery box, there are no writable data structures in the text segment and no sources of pointer manipulation using text segment addresses. Since error statistics (presented in the next section) show that most addressing errors are localized around the intended data structures, this should eliminate most addressing errors except for those caused by manipulation of the recovery box itself.

5.2.3. Implementation Shortcomings

This section discusses four shortcomings in the current recovery box implementation: its lack of access protection, its static memory allocation, its lack of atomicity for operations on multiple items and the cost of creating and deleting types.

At present, there is no security provided by the recovery box, except that only applications with root privilege can access items allocated by the kernel. Thus any privileged application can examine or modify items stored by another application or by the operating system itself. Also, any unprivileged application can access items stored by another unprivileged application. This has not been a problem in Sprite, because we do not yet have many applications that use the recovery box.

The second problem, static memory allocation, imposes a limit on the number of item types that can be initialized and a limit on the overall size of the recovery box. This is one reason why the maximum number of items that will be valid simultaneously for a type must be specified when that type is initialized. The type initialization function returns an error if there is insufficient space for the desired number of items. Currently, the size of the recovery box is compiled into the operating system. If the recovery box resides in non-volatile memory such as battery-backed RAM (to protect its contents in the event of a power failure), it is likely that the amount of non-volatile memory will already impose a size restriction. Even without such a restriction, placing the recovery box in a static area of system memory, such as the text segment, makes it difficult to expand the recovery box in physical and virtual memory while guaranteeing the same virtual/physical address mapping across reboots. More complex solutions are possible, such as putting a page map of the recovery box in a well-known location and allocating the recovery box pages non-contiguously.

The third problem is that operations on multiple items are currently not atomic. It would be easy to change the implementation to provide atomicity of multiple item inserts, but providing atomicity of multiple updates and deletes would significantly increase recovery box complexity. While neither the operating system nor the applications I considered would benefit from atomicity of multiple updates, this could be a worthwhile problem to tackle in other environments.

The final problem is the cost of creating and deleting types. If these operations occur frequently they will have poor performance. Freeing a type from the middle of other allocated types can result in a fragmentation problem. The space consumed by the freed type may not be enough for any type allocated subsequently. If this occurs, all the item information and storage arrays could be shifted down (copied) in memory to leave enough space for the new type. The addresses in the per-type information would be updated to point to the new location of the item arrays. This shift operation would not affect clients of the recovery box, because the clients have no direct pointers to internal recovery box data. Although shifting the item data would only occur on type initialization, it would be costly and the shift is not an atomic operation. It is necessary to put a code in the current operation field of the header to signal whether a crash occurred in the middle of a shift. If so, the recovery box would be unusable and the system would not be able to use it for fast recovery. If clients of the recovery box need to delete types frequently, a different recovery box design will be necessary.

5.3. How Sprite Uses the Recovery Box

The previous sections have described in general the motivation for the recovery box and its interface and design. This section turns to a more specific use of the recovery box: how Sprite uses it to preserve a copy of the server's distributed state information across failures. This section explains when Sprite must update the recovery box, what it stores there, and how much memory this requires.

Sprite must update recovery box items every time the distributed cache or file system state changes. This occurs whenever a client opens or closes a file or other object. In Sprite, all opens and closes are processed on the file server anyway, which provides the opportunity to insert the appropriate calls to recovery box functions. Cache state must also be updated when a process with associated open files migrates from one machine to another, but this is treated as if the source client has closed its files and the destination client has opened them.

Table 5-5 lists the components of each of the cache and file system state items Sprite stores in the recovery box. Although there is a single item type for this file system data, the meaning of the item contents differs depending upon what is stored. Each item stores information about an I/O handle or stream. Chapter 3 explains how the recovery system uses each field of these items. The first field of each item is an *ID*. For files, pseudo devices and other objects in the file systems, this is the file ID that identifies the object via its I/O handle. For streams, this is the stream ID that identifies the open stream that references an I/O handle. I have modified these IDs to include the host number of the client responsible for the item reference. I/O handle and stream IDs usually include the server's number, but the server is well-known in this case. The second field, *otherID*, is only used if the ID is for a stream. If so, then otherID is the file ID of the I/O handle referenced by the stream. The *use* field gives the number of references a client has to the handle or stream, separated into read and write references. For an I/O handle, this is how many open streams the client has for that I/O handle. For streams, this is the number of times the client has *dup*'ed the stream or the stream has been inherited by another process on that client. The *info* field gives information specific to the type of handle. For file I/O handles this gives the version number of the file. For pseudo devices this gives the ID of the host running the pseudo-device server. For streams it gives further stream-specific user flags. The *clientData* field is also specific to the handle type. For files, this field indicates whether the file is cached or uncacheable. For pseudo devices, it gives a number called the seed which indicates the instantiation of the pseudo device. For streams, the field gives the current offset into the stream.

Field	Purpose
ID	File ID for I/O handle (file, pseudo device, or other) or stream ID for stream. This includes the client host ID.
otherID	If ID was for a stream, this is the ID of the associated I/O handle. Otherwise it's empty.
use	Number of client's references to handle or stream.
info	For files: version number. For pseudo devices, serverID. For streams: more use flags. Otherwise unused.
clientData	For files: whether cacheable. For pseudo devices: seed number. For streams: offset. Otherwise unused.

Table 5-5. Components of distributed cache state items.

This table lists the components of each item Sprite stores in the recovery box. These components are further described in the text. There is one item for each I/O handle and stream accessed by each client.

Using the recovery box requires the addition of a some new bookkeeping code in the Sprite file system for a couple of reasons. First, this code is responsible for inserting, deleting and updating items in the recovery box. It copies the relevant portions of I/O handles and streams into items to store in the recovery box. It currently uses a hash table to map from I/O handles and streams to their recovery box items. Second, for convenience this extra code also maintains the per-client reference counts in these hash table entries, to avoid extra accesses to the recovery box. (These reference counts are different from the ones already in I/O handles, because the I/O handle use counts on the server include the total references from all clients to the object, rather than per-client references.) Much of this extra bookkeeping could be eliminated if I were to restructure Sprite's server file handles to include the per-client reference counts and the recovery box item ID.

The recovery box does not need much of the file server's memory. On average, the server maintains 10,000 to 15,000 handles and streams for about 40 clients, as measured on the main file server in the production Sprite system. However, only about half of these need to be stored in the recovery box, because only half are streams or I/O handles for objects that are open or have dirty cache blocks on a client. (The ratio of unopened, clean files to open or dirty files is higher on the server than the ratio described in chapter 3 for the clients, because the server recycles the unnecessary I/O handles faster than its clients do.) With 40 clients this gives us 5000 to 7500 handles on average that need to be stored in the recovery box. From each of these structures the server preserves 52 bytes of essential information in the recovery box, making the server's recovery box space requirements less than 400 kilobytes for 40 clients, including space for the per-type and per-object information arrays.

5.4. Results and Measurements

This section evaluates the performance of transparent recovery and the recovery box itself. It first gives the transparent distributed state recovery time, including measurements across varying numbers of clients to show how it scales to larger systems. This is followed by timings of the individual components of transparent recovery. Besides recovery timings, I also include measurements of the file system overhead caused by maintaining the recovery box during normal operation. The effect of the recovery box on regular execution is only a few percent. The overall effect of transparent recovery on reboot times depends on how often the recovery box is corrupted. If the recovery box has been corrupted by a software error or power failure, recovery time will take as long as the backup recovery mechanism plus the time it took to start transparent recovery and discover that the recovery box was corrupted. Unless otherwise specified, all measurements were done in the testbed setup described in chapter 3 using the basic state setup presented in Table 3-5.

5.4.1. Recovery Times

Transparent recovery is the fastest method for recovering the distributed cache state information. The time required to recover state for ten clients in the testbed setup is 1.47 seconds. Run six times, this test showed a standard deviation of 3%. However, transparent recovery in Sprite is disk I/O bound, so it scales no better to larger numbers of clients than does server-driven recovery. Figure 5-5 shows transparent recovery times for varying numbers of clients. From the apparent slope of the measurements, we see that each additional client in the testbed setup adds 130 milliseconds to the recovery time. Figure 5-6 and Figure 5-7 show us that with three or more clients, the disk is 75 to 80% utilized, while the CPU is only 45 to 50% utilized.

Table 5-6 shows a breakdown into component parts of the transparent recovery time for ten clients. I/O handle recovery is the most expensive part of transparent recovery, because the server spends the bulk of this time retrieving file descriptor information from disk for the I/O handles, as described in section 3.1.4. This transparent recovery test causes 632 descriptor read requests. Of these requests, 600 correspond to the 60 unshared files per client. There are 30 additional requests for the shared files, since these need only be read once. There are two additional descriptors fetched: one for the file system root directory and one for the file created and unlinked by one of the clients. However, many of these descriptor read requests do not require an actual disk I/O. File descriptors for files in the same directory are usually placed together in a disk block, so one disk I/O may read in many file descriptors (up to 32 descriptors), which the server then caches in its main memory. Thus some requests find the descriptor already cached. I arbitrarily spread the files out through about 30 directories, with the result that 51 disk accesses are necessary for the test. Direct measurements confirm that the disk is 95% busy during this part of transparent recovery.

I/O handle recovery would take even longer if I did not enable overlap of I/O and CPU processing. One of the problems for transparent recovery in Sprite is that the distributed state information is not retrieved from the recovery box through RPC requests. In client-driven and server-driven recovery, RPC server processes automatically provide us with multiple processes to handle recov-

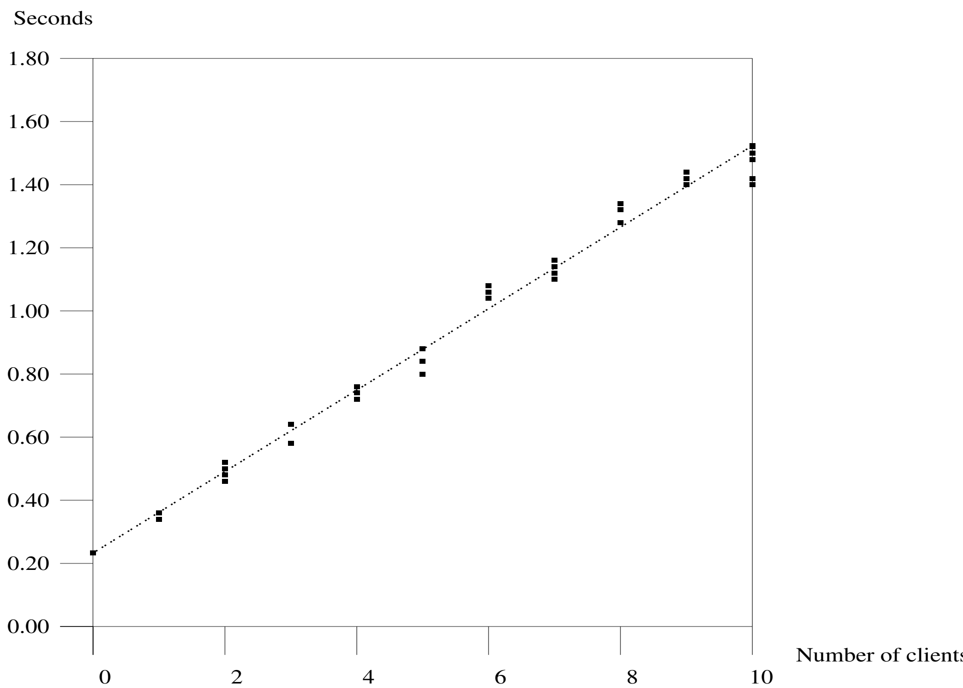

Figure 5-5. Transparent recovery by number of clients.

This graph gives the time for transparent recovery as a function of the number of clients in the testbed setup. The X axis gives the number of clients in the timings, and the Y axis the number of seconds required for recovery. There are five to six data points for each number of clients, but some of the points lie on top of each other so they are not separately visible. The dotted line gives the apparent slope of the measurements, using least-squares linear regression. The slope of the line is 0.129 seconds per client. The per-client setup for this data is shown in Table 3-5, with one modification: to make the setup more realistic, for timings of one to nine clients, only one client repeatedly truncated and wrote a file. When ten clients were available, two clients participated in this activity.

---

ery. This allows for automatic overlap of I/O and CPU processing as the different processes execute. For transparent recovery, I had to create extra processes specifically to enable this overlap.

To improve transparent recovery time further it is necessary to eliminate disk I/O for file descriptor fetches. There are two options for avoiding this I/O. As described for server-driven recovery, one possibility is to delay the I/O until after recovery, and only fetch the descriptors as they are referenced. This may not save much time, since the descriptors are likely to be referenced quickly. Transparent recovery presents a new option: the system can store some of the needed descriptor information in the recovery box, so that the server can recover it too from main memory. I have not implemented either of these options.

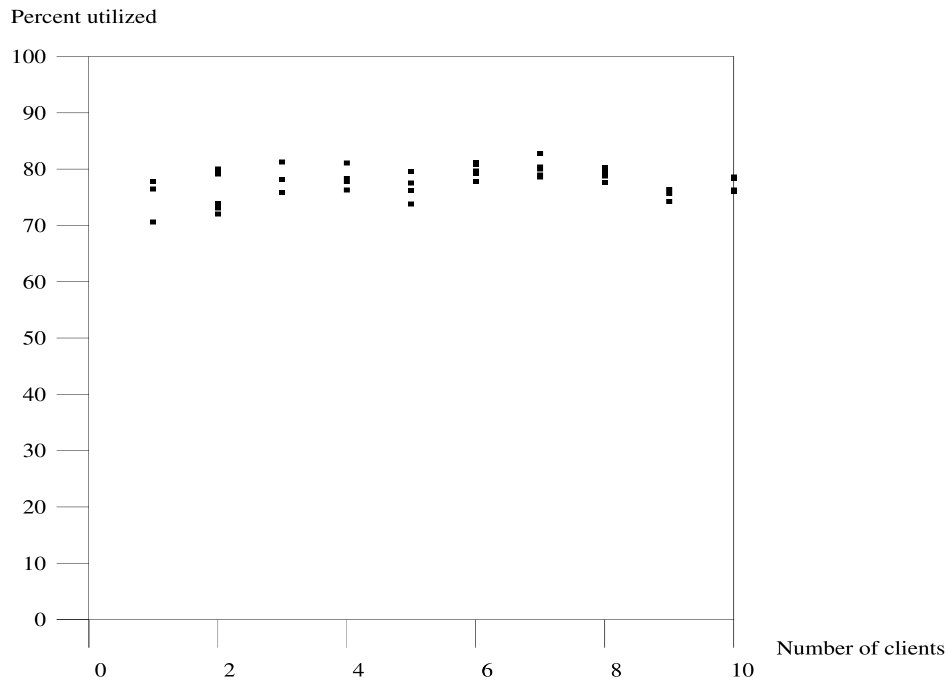

Figure 5-6. Disk utilization during transparent recovery.

This graph gives the percent of disk utilization during transparent recovery versus the number of clients in the testbed setup. The X axis gives the number of clients in the timings, and the Y axis gives the percent utilization. There are five to six data points for each number of clients, but some of the points lie on top of each other so they are not separately visible.

5.4.2. File System Overhead

Maintaining the contents of the recovery box during normal system processing does not significantly reduce the performance of Sprite. Table 5-7 shows the breakdown of time required for the Sprite file system recovery box operations. The table gives accumulative measurements for the recovery box code itself, for the file system layer that calls the recovery box code, and for the file open and close times seen by a SPARCstation-2 client of the file server, with and without the recovery box. The file open/close measurements include the time for the kernel-to-kernel remote procedure calls between the client and file server. For convenience I report the measurements in terms of pairs of operations – recovery box item insert/delete operations and file open/close operations.

For a SPARCstation-2 Sprite file server, the time to insert, checksum and delete a file handle item in the recovery box is 28 microseconds on average. This includes four microseconds per han-

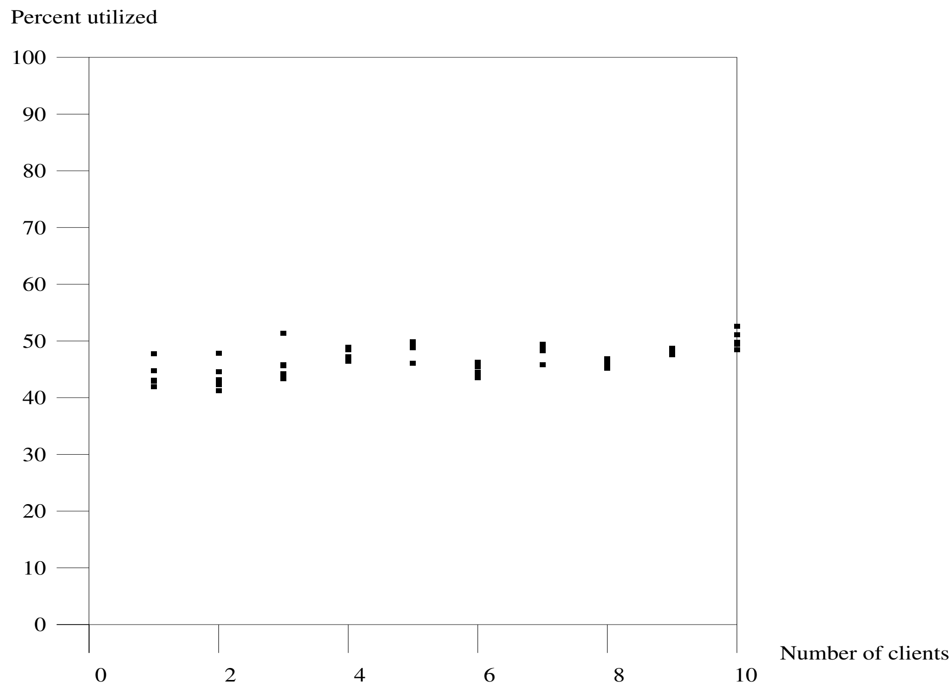

Figure 5-7. CPU utilization during transparent recovery.

This graph gives the percent of CPU utilization during transparent recovery versus the number of clients in the testbed setup. The X axis gives the number of clients in the timings, and the Y axis gives the percent utilization. There are five to six data points for each number of clients, but some of the points lie on top of each other so they are not separately visible

---

dle for the checksum calculation. The open/close test performs two insert operations during file open and two delete operations during file close for reasons explained below.

The file system bookkeeping code that calls the recovery box adds more time to the insert/delete operation, for a total of 72.3 microseconds on average. This includes the 28 microseconds for the recovery box functions. In part, the extra time results from setting up the items to insert in the recovery box, but it is also due to a problem in the current implementation. Inserts and deletes of file handle items currently require some extra bookkeeping and a hash table lookup. The hash table maps from file handles to itemNumbers and recovery box item reference counts. Much of this extra code would be unnecessary if we could avoid this mapping by changing the format of the file handle structure in Sprite to include the itemNumber and recovery box reference count. Unfortunately modifying the I/O handle structure would require recompiling and rebooting the entire Sprite cluster, because the hosts in my test network still depend on the real Sprite servers for accessing the root file system and other services. The resulting outage would affect many irritable graduate students and aggressive faculty members.

Operation	Seconds	Standard deviation
Item retrieval from recovery box	0.08	0.007
Hash table creation	0.08	0.007
I/O handle recovery	1.20	0.040
Stream handle recovery	0.10	0.007
Total	1.47	0.044

Table 5-6. Transparent recovery time and component times.

This table gives the time in seconds for different steps of distributed cache state recovery for the server using a recovery box in the testbed setup. Only 0.08 seconds are actually spent retrieving information from the recovery box. The rest of the time is spent building the server's main memory data structures, including a hash table of handles. Most of the time is spent recovering I/O handle information, since this requires disk I/O operations on the file server. The test was run six times. The results do not quite add to 1.47 due to rounding errors.

The latency experienced by a client opening and closing a file on a server with a recovery box includes twice the cost of a recovery box insert/delete operation, along with the corresponding file system bookkeeping code, because each file open/close pair requires inserting and deleting two items in the recovery box: an I/O handle and a stream. The extra latency for remote open/close pairs from the client's view is 166 microseconds on top of 3450 microseconds, or a little less than five percent extra latency. Write protection adds eight microseconds (two microseconds for each of the four recovery box updates) bringing the cost to 174 microseconds – almost exactly an extra five percent latency.

While the recovery box adds a five percent latency to the open/close times seen by a client workstation, the effect on file server throughput is not as large. On average, the main Sprite file server for 40 clients receives two file open/close pairs per second. The current recovery box implementation thus adds, on average, an extra 332 microseconds of server processing per second, or less than a 0.1% increase in processing demand at the server. However, file system activity is bursty, and the server sometimes sees as many as 50 file open/close pairs in a second. During peak activity, the recovery box would thus add 8.3 milliseconds of server processing per second, for a potential 0.8% reduction in server throughput. With write-protection, this comes to about a 0.9% reduction in server throughput. I believe this means the recovery box overhead is acceptable, even in a high performance system. And if we were able to eliminate much of the overhead in the file system bookkeeping code on the file server, then the increase in latency seen by the clients could be cut in half.

5.5. Disadvantages of Transparent Recovery

This section lists some of the disadvantages of transparent recovery and difficulties with the recovery box. There are three main problems. First, transparent recovery requires more special-

Operation	Average time (microseconds)
Recovery box insert/delete with checksum	28
Recovery box insert/delete with no checksum	24
FS insert/delete with checksum	73
FS insert/delete with no checksum	72
Open/close with recovery box	3616
Open/close without recovery box	3450
Added cost of write-protection	8

Table 5-7. Sprite recovery box performance.

The first two rows give the time to insert and delete a file handle item in the recovery box, with and without a checksum. The second two rows give the time to insert and delete a file handle, including the extra overhead in the file system (*FS*) bookkeeping code. The last two rows give the time to execute a file open/close operation from a client workstation, with and without a recovery box running on the file server. This time includes the kernel-to-kernel remote procedure calls between the client and file server. The last row gives the number of microseconds to add to the other results when using hardware write-protection of the recovery box pages. Each result is the average of 50,000 operations.

---

purpose code on the server than the other recovery techniques. Second, it requires more careful bookkeeping to avoid inconsistencies between the clients and the server. Third, the recovery box is difficult to incorporate into a system in which operations that change the file system state, such as opens and closes, are hard to isolate or handle atomically. Another issue is what to do if the recovery box is corrupted. For extra robustness, a system may need to use client-driven or server-driven recovery as a backup recovery technique. Finally, this section also discusses two possible problems with implementing the recovery box: uncooperative hardware and power failures.

Unfortunately, transparent recovery requires a significant amount of special-purpose code on the file server. The recovery box implementation takes 1475 lines of C code. The file system layer above the recovery box (for maintaining the hash table, and for logging directory operations as described in the next section) takes about 2700 lines. Finally, there is code executed only during recovery for each type of I/O handle: 130 lines for files, 40 for pipes, 65 for devices, and 40 for pseudo-devices. This gives us 4450 total extra lines. These numbers include all testing and debugging code, and about one quarter of the lines are comments.

The less code a system executes specifically during recovery, the less likely recovery is to fail due to bugs. From this point of view transparent recovery does not fair too badly, since only 615 lines of the 4450 are executed only during recovery. The rest of the code is used during normal processing to maintain the distributed state in the recovery box. One good feature is that transparent recovery requires less special-purpose code on the clients, since they do not need to participate in recovery.

A second problem with transparent recovery is that it requires more careful bookkeeping than the communication-based techniques, particularly with reference counts. An example of this problem concerns close RPC requests that hang waiting for a file server to recover. In Sprite, users may terminate the hung operation at a high level on their workstations. This is convenient if a user does not want the request to hang until the server reboots, but it should be disallowed when using transparent recovery. A successful close request decrements the reference count on a file. A file server may crash after decrementing this reference count in the recovery box but before it responds to the client. In this case the client's state information becomes inconsistent with the server's. This does not matter with server-driven or client-driven recovery, since the server does not preserve its state information across failures. After server-driven or client-driven recovery, the server's state will match the clients' state, because the server simply uses the clients' state information. But in transparent recovery, the clients and the server have no natural opportunity to resynchronize their state information. The solution to this problem is to disallow termination of RPCs that modify the distributed cache or file system state information.

Additionally, these RPCs must be idempotent: when retried by the client, the server must be able to detect whether it has already modified the reference count. One way to implement this is to keep a log in the recovery box of the status of outstanding RPCs, but I have not implemented this.

A third problem with transparent recovery is the difficulty of incorporating it into file systems that were not designed for it from the beginning. In Sprite, the most difficult aspect of incorporating the recovery box was isolating the locations in the code where the Sprite server updates its distributed cache state information. The file server processes all file open and close operations, which makes it theoretically possible to find all the distributed cache state changes. However, isolating the important changes proved difficult. This is because the changes are not well-confined, much less atomic. For example, a Sprite file server processes an open request in several steps, first translating the name of the file to its descriptor, then updating or creating an I/O handle for the file, and only at the end creating the open stream handle for the file. A file is not officially open until all steps are complete, and the server executes all these steps before responding to the client. However, there can be an error during any step, making it necessary to back out of all the steps and undo all associated changes to the recovery box. Two of these steps require updates to.the recovery box (modifying the file I/O handle and creating the stream handle), so we must undo changes in the recovery box as well. Unfortunately, this problem is likely to be present in many file systems that were not designed from the beginning to have atomic, or at least well-confined, state changes.

A fourth potential problem involves a trade-off between robustness and complexity. Switching to transparent recovery does not necessarily mean we should leave out a communication-based recovery technique, such as client-driven or server-driven recovery. Although the failure statistics included in this chapter indicate that recovery box corruption is unlikely, and even though we can write-protect the recovery box, it is still possible to lose the recovery box contents. If this happens, we could recovery the system by retrieving the file system state information from the clients with client-driven or server-driven recovery. However, this makes the system more complex, because it requires implementing two types of recovery. For simplicity, we could instead just reboot all the clients to restart the system. This would disrupt many users, but it should only be necessary on rare occasions.

Although implementing two forms of recovery is complex, it is worth mentioning that this two-tier recovery mechanism is much less complicated than systems using error-repair and other high-availability techniques described in chapter 2. If any part of transparent recovery fails, we can just back out to client-driven or server-driven recovery, and we need not try any intermediate recovery techniques.

Memory corruption in the recovery box is not the only reason we may need to back out to some other form of recovery or system reboot. Another reason is that the contents of the recovery box could be logically incorrect. This can occur if the same incorrect object that caused the system failure was also copied into the recovery box. This is an example of error propagation. If an item in the recovery box is wrong, the system may fail soon after it retrieves the item from the recovery box. Thus, the system should back out to another form of recovery (or at least indicate that a full system restart is necessary) if any failure occurs within some period of time after recovery from the recovery box. So far, though, I have no data to suggest what this period of time should be or how much it will vary across systems.

Finally, there are a couple of reasons why the recovery box itself may be impractical to implement on some architectures and in some environments. The first problem is power failures. While our department rarely suffers from power failures, this is not true in many geographical areas. If the recovery box is stored in the server's volatile main memory, the contents will be lost after a power failure. One solution is to use a half-megabyte or so of non-volatile RAM for the recovery box. This is less costly than an uninterruptable power supply [Baker92b]. However, our experience has been that the client workstations also lose power during a power failure. In this case, there's no point in preserving their state on the server across the failure, because they must all be rebooted from scratch anyway. So we have found little need to use non-volatile memory for the recovery box.

The second problem is that the boot PROMs of some machines modify or clear memory. For example, the memory check initiated by the boot PROM of the MicroVax 3500 modifies a word for every 1024 words in memory [Johnso93]. If it is not possible to turn off such activities as the memory check, it may be impossible to find a large enough area of physical memory that remains unmodified by the boot PROM. In this case, the recovery box cannot be implemented in contiguous physical memory and non-contiguous memory layout complicates the implementation.

5.6. File System Trends and Their Implications

This section turns to a couple of trends in distributed file systems, and their effect on the recovery box. There is a trend toward more asynchronous file system operations and towards stateful file servers. Both of these trends make the recovery box more attractive, but also lead to some problems with the recovery box.

5.6.1. Logging Asynchronous Directory Operations

For higher performance, new file systems tend to use more non-blocking asynchronous operations than their predecessors. Applications need not wait for these operations to complete, since they can be handled in background. But if such an operation modifies the distributed file system state, its asynchronous properties make bookkeeping more difficult. However, it is possible to use the recovery box to handle this bookkeeping problem.

Examples of such asynchronous operations in Sprite are directory modifications. In Sprite, operations such as file creation and deletion are executed asynchronously on the file server. For instance, when the server gets a request for a file creation from a client workstation, it updates its main memory data structures and responds to the client indicating that the file has been created, but only queues the disk write to be handled later. The client assumes the file was created, but it may be 30 seconds before server actually creates the file on disk. Non-blocking asynchronous directory operations make many applications faster, because these applications create temporary files to hold intermediate results, and they delete the files soon afterwards. Handling these operations synchronously would require applications to wait for the operations to complete, tying their performance to the speed of the disk.

The problem with asynchronous non-blocking directory operations is that Sprite clients and the server can end up with different directory state after a server crash. If the server crashes after responding to a client but before updating its disk to reflect the new state, a client may believe that a file has been created while the server knows nothing about it. In client-driven or server-driven recovery, the client and server communicate during recovery and have the opportunity to resolve the inconsistency. If the client attempts to reopen a file that the server does not find on disk, the client will receive an error message (although currently Sprite clients take no corrective action except to issue an error to the application program if it is still running). With transparent recovery, we lose even this opportunity to resolve the inconsistency during recovery. If a client created a file and still has it open, the server will find an item in its recovery box for an open cached file, but it won't find this file on disk.

To solve this problem in transparent recovery, we must record the directory operations (create, unlink, link, and rename) in some way. I record them in the recovery box. The file server can log file creation and deletion events at main memory speed in a section of the recovery box before responding to the client. When the associated disk operations complete, the server removes their log entries. The log in the recovery box is of fixed length. If it becomes full, then the server empties it by forcing to disk all the directory changes. The I/O is amortized across all of the changes in the recovery box. Logging the directory change events does not significantly increase the latency of the clients' operations, and it allows the server to make its directory state on disk consistent with the clients' views by processing the directory change log during crash recovery.

During recovery, the server processes the log by examining the directory events and checking its disk to see if the file has actually been created or deleted as requested. This adds I/O operations to recovery, but not many. On average, there are only three of these events pending on the main Sprite file server. In the worst case, each of these events would require checking the disk and modifying the disk to create or delete a file. This would be six I/O operations and would add, on average, 0.18 seconds to recover for 40 clients, assuming 30 milliseconds per I/O operation. Although I have implemented recovery from this log in Sprite, it is not yet reliable enough for better performance measurements. Unfortunately, handling this log is complicated.

5.6.2. Stateful File Servers

Besides the trend towards asynchronous operations, recent file systems are increasingly stateful, and this too has implications for the recovery box. While NFS avoided maintaining state on the file server in order to make crash recovery easier, recent file systems such as DEcorum, Echo and Sprite keep state information in order to increase performance, as described in chapter 2. The closer information is kept to where it is processed, the faster the system runs, so it is likely that future file systems will also keep state on their servers. With more state kept on the file servers, the recovery box becomes more attractive for preserving the state across failures. However, the more state that is maintained in the recovery box, the more likely some of it is to be wrong and cause error propagation problems. This is a trade-off that will vary from system to system, so it is hard to make generalizations about how much state should be kept in the recovery box.

5.7. Applicability to Other Systems

So far this chapter has primarily described how Sprite makes use of the recovery box, but this section addresses the requirements for other systems and user-level applications that could use it. Many client/server systems with state information that changes in response to client requests will find the recovery box useful.

There are several requirements that must be met before a system will find the recovery box practical. For instance, it must be possible to identify and isolate the changes to the state information. As mentioned earlier, this can be difficult in a pre-existing system where state changes may be made incrementally and in several places. Also, there are four desired characteristics about the type of state items the system stores in the recovery box. Without these characteristics, magnetic disk is likely to be a better form of stable storage. First, the items should be expensive to regenerate from scratch, or there's no point in preserving them across failures. Second, the individual items should be small, so that the cost of copying them into the recovery box is not great. The total size of the recovery box is also likely to be limited, so for example, copying all kernel data structures to it is probably a bad idea. Third, the items should be updated too frequently to be reasonably stored on disk. If they are rarely updated and the latency of a synchronous disk write is not a problem, disk storage is preferable to the recovery box because it is simpler. Fourth, the items should be correct. This last requirement is the most difficult.

The items stored in the recovery box should be correct in order to avoid error propagation problem, as described on page 81. Ensuring that the items are correct is difficult and hard to generalize across systems. This requires some experience with the system to know what types of state information do not have a history of causing system failures. In Sprite, for example, I do not store the internal file system locking information in the recovery box, because incorrect locking has been the cause of many of our failures.

5.8. Application Use of Recovery Box

Up to this point, this chapter has mostly described the operating system's use of the recovery box, but user-level applications can also use the recovery box to preserve state across failures. This section describes how one application, an experimental version of the POSTGRES database management system (DBMS) [Stoneb86], uses the recovery box.

In this experimental version, written by Mark Sullivan [Baker92a], POSTGRES runs as an application program on a Sprite file server and responds to requests from client programs running on other Sprite machines. POSTGRES accesses the recovery box using the system call interface described in section 5.2.1.

5.8.1. How POSTGRES Uses the Recovery Box

Recovery performance in the POSTGRES database management system is dominated by the cost of reinitializing the DBMS server's connections with clients. In a conventional database management system, recovery includes the cost of write-ahead log processing (recovering disk state) in addition to client connection reestablishment. However, POSTGRES has an unconventional storage manager that maintains consistency of the data on disk without requiring write-ahead log processing [Stoneb87], so it does not need the recovery box to avoid this costly recovery step.

Without the recovery box, connection reestablishment is driven by the clients. When the database manager fails, all transactions executing on behalf of clients are aborted and all connection state held at the server is lost. Connection state includes such things as authentication information (secret keys or authentication tokens), client addresses, packet sequence numbers, and any packets queued at the DBMS.

When the server recovers, it must wait for the clients to detect that it has failed. After a connection times out, the client must reopen a connection with the DBMS and reauthenticate itself. POSTGRES can establish and authenticate (application-level) connections with only three messages, because it uses a sequenced packet protocol built on top of a datagram protocol (UDP), rather than a protocol based on streams (TCP/IP). After establishing the connection, each client must query the database to find out if its last transaction committed. Then, it must resubmit the transaction or take some other recovery action.

In the system with a recovery box, the DBMS stores authentication information and the client address associated with each connection in a recovery box item. The DBMS also stores the transaction ID of the last transaction it was executing on behalf of each client. Every time the DBMS begins a new transaction, it updates the recovery box item with the new transaction ID. Storing this transaction ID on disk would be a bad idea, since POSTGRES is already disk bound for workloads with high transaction rates.

After a failure, the DBMS reinitializes its connection data structures from the backup data stored in the recovery box. Once the connections are reinitialized, the DBMS sends a message to each client indicating that recovery has occurred (a new DBMS server ID is sent) and that a new sequence number has been chosen by the DBMS. The restart message also indicates the status of the last transaction that the DBMS executed on behalf of the client. If the message initiating the transaction was lost in the failure, or if the transaction was aborted, the client must either resubmit the transaction or take some other recovery action. Authentication of the client is reverified when the DBMS receives the next message from the client.

POSTGRES does not use the recovery box to store any of the state associated with its storage system. Storage system performance optimizations requiring non-volatile RAM are discussed in [Stoneb87]; for example, to reduce commit latency, committed data can be stored in non-volatile RAM instead of on disk. But this technique requires the operating system to guarantee that data stored in non-volatile memory be permanent. The recovery box does not make this guarantee, because if the system detects any errors during the fast recovery path, it will revert to the traditional recovery path and will discard the contents of the recovery box.

5.8.2. POSTGRES Performance with the Recovery Box

The most lengthy step for POSTGRES recovery is reinitializing the server's connections with client applications. Without the recovery box, POSTGRES clients discover failures using time-outs. After the time-out, each client must query the database to find out whether its last transaction committed. The time-out alone requires several seconds since it must allow for worst-case queueing delays before assuming that the DBMS is down. Eliminating this communication by using the recovery box is an easy way to fix this problem.

To measure POSTGRES recovery times, Mark Sullivan ran a debit/credit benchmark based on TP1 [Anonym88], but to expedite the measurements he used a much smaller database than the actual TP1 benchmark requires. A single POSTGRES DBMS managed the database from a Sprite file server. Ten POSTGRES client processes running on a single Sprite client machine generated the transactions. For the experiments, he used a version of the DBMS that was single-threaded. While the DBMS executes a transaction for one client, transactions sent by the others are queued in the DBMS address space. Single-threaded execution means that adding POSTGRES clients increases recovery time (due to client connections) without increasing the overall transaction rate of the DBMS.

With a fast reconnect protocol that relies on the recovery box, POSTGRES clients are notified immediately of a DBMS failure and can resubmit lost transactions with a single message exchange. The time to recover ten client connections using this protocol is less than a second. Using the recovery box, total recovery time for POSTGRES and ten clients (ignoring operating system recovery time) is about six seconds. Most of this cost comes from reloading the database code and system catalogs into main memory.

Other systems have implemented client recovery with ad-hoc mechanisms involving several message exchanges, queries of the database, and sometimes human intervention (including the original POSTGRES implementation). The times required by these mechanisms vary widely but all are longer than a single message exchange per client.

Individual POSTGRES recovery box operations are slower than those for Sprite, because POSTGRES must pay the overhead of system calls. Table 5-8 shows the breakdown of times for POSTGRES recovery box operations. POSTGRES updates a 92-byte item in the recovery box on each transaction versus Sprite's 52-byte items; the entire connection structure is updated even though only the transaction ID changes. Because it is inserting larger items into the recovery box, the cost of copying data and computing checksums also increases. The operating system is able to use an optimized checksum routine for file handles (with an unrolled loop) while application programs must use a generalized one. POSTGRES could compute and check its own checksums if the difference in performance were critical. However, the measured cost of the recovery box operations is much smaller than the variance of POSTGRES transaction execution times, so there is no reason to optimize checksum calculations further.

Operation	Average time (microseconds)
System call overhead	19
Checksum 92 bytes	10
Copy 92 bytes	8
Other	18
Total for recovery box update	55

Table 5-8. POSTGRES recovery box performance.

This table shows a breakdown of the time required to update a POSTGRES client connection in the recovery box, as measured by Mark Sullivan in the testbed setup. The first three entries give the major individual components of this cost – system call overhead, checksum calculation, and copying costs. The results give the average across 100,000 measurements.

5.9. Summary

While disk is the most common form of stable storage, it is not always a better storage medium than main memory for frequently updated information. Systems designers commonly assume that the contents of a disk are more reliable after a failure than the contents of memory, but there is no guarantee. With suitable precautions, such as the checksum methods described in this chapter, the recovery box is an attractive form of stable storage for systems that are allowed to suffer an occasional longer reboot.

Transparent recovery, using the recovery box, is the fastest technique for recovering distributed state information in Sprite. It requires 1.47 seconds to recover the state for ten clients in the testbed setup. The recovery box does not significantly degrade overall system performance, as it adds at most a five percent latency to remote open and close operations on client workstations.

The recovery box is useful for user-level applications as well as the operating system kernel. State information suitable for storing in the recovery box should be expensive to regenerate, small, updated too frequently to store on disk, and free from errors that would cause repeated failures.

The biggest problem for incorporating the recovery box into a pre-existing file system is that may be hard to isolate the changes to the state information stored in the recovery box and make these changes atomic. This was true in Sprite. However, we can design future file systems with this in mind: state recovery will be faster and easier if state changes are well-confined.

6 Other Fast Reboot Techniques

The previous chapters describe how I reduced distributed state recovery in Sprite from many minutes to a few seconds using server-driven or transparent recovery. However, a file server exectues several other steps during a reboot. This chapter turns to these other steps and gives mechanisms for improving the time of each. Combining all these improvements, including those for file system state recovery, we have reduced the total time required for Sprite file server recovery from 45 minutes to under 29 seconds. This has improved overall file server availability.

To reboot, a Sprite file server executes five general steps. First, the server must reacquire an image of the kernel code and data. Second, it runs start-up code to initialize its hardware and various data structures. Third, it ensures that the file system is in a consistent state. Fourth, it recovers its distributed state information. Finally, it must start up various kernel and user-level daemon processes to handle different parts of its workload. The first five sections of this chapter (sections 6.1 through 6.5) provide more details for each of these steps in turn. Section 6.6 lists some further techniques for improving overall reboot times.

Table 6-1 lists the five reboot steps and gives the time for each, before and after various improvements. The timings in the table come from the testbed setup described in chapter 3 and have been scaled for 40 clients, except for the following three cases. First, the seven-second time to download the kernel is taken from the production Sprite system. For reasons explained below, I could not take this measurement in the testbed setup. Second, the disk check and LFS times are also from the production Sprite system, since the testbed server has only one small disk. Third, the 4 1/2 minute timing for the original distributed state recovery is the length of the recovery storm from chapter 3, Figure 3-6, using the testbed setup with only ten clients rather than 40. However, this recovery storm was artificially induced, and since recovery storms are due to instability, they are of arbitrary length.

6.1. Reusing Text and Initialized Data

The first step a server executes to reboot is to reacquire an image of its kernel to run. In particular, it needs the text segment, because that contains the kernel code, and it needs the initialized data segment, because that contains all the data structures that must start out with initial values. The server does not need to acquire the zero-filled data segment (BSS segment), since it can create this area merely by zeroing a section of memory. The uninitialized data requires no start-up treatment.

Reboot step	Time	Improvement	New time
Download kernel image	7 seconds	Reuse text and initialized data	0.03 seconds
Initialize kernel modules	23 seconds	Store pre-computed values	18 seconds
Check disks	40 minutes	Switch to LFS	3 seconds
Recover distributed state	4 1/2 minutes	Transparent recovery	5.3 seconds
Start kernel and daemon processes	10 seconds	Delay some processes	2 seconds
Total	~ 45 minutes	New total	28.3 seconds

Table 6-1. Server reboot steps and timings.

This table lists the steps a server goes through during rebooting. It gives rough times required for these steps originally, and the time required after various improvements described in this chapter. The times are scaled for 40 clients in the testbed setup, except for three cases. The time to download the kernel from disk and the disk check and LFS times were measured in the production Sprite system. The original 4 1/2-minute recovery storm was measured in the testbed setup with only ten clients and was artificially induced.

Traditionally, the file server *downloads* the kernel (text and initialized data) from disk or across the network from another machine. On Sun workstations, the server's boot PROM first locates (on disk or across the network) a disk boot or network boot program and downloads this program. The server then uses the boot program to download the kernel image. On the main Sun-4/280 file server, downloading the kernel from disk takes about seven seconds. On a SPARCstation-2, downloading the kernel from across the network takes about 60 seconds, because the network boot program delivers the data in small packet sizes. Unfortunately, I could not measure the time for a disk boot on the SPARCstation-2, because the only file system on that server is an LFS file system, and we do not yet have a disk boot program that works for LFS.

To reduce the time needed to reacquire the kernel image, we have two options: optimize the download programs, or reuse the kernel image already in the server's memory. I chose the second option, because it will always be faster than downloading the kernel. To do this, we must treat the text segment and initialized data segments separately. The kernel text can be reused in place, because it does not change during the operation of the system. The text segment is also read-protected, so it will not be corrupted by kernel addressing errors. Reusing the initialized data is more complicated, because it is modified during system execution. To solve this problem, the first time a host downloads the kernel image, the system copies the initialized data to an area of memory that is then read-protected. For convenience, I use a few pages of the text segment. To reboot after a crash, the system merely copies the original initialized data out of the text segment to its correct address and then begins execution with the fresh copy of the data.

The time required to reuse the kernel image this way depends on the time needed to copy the initialized data to the correct place in memory plus the time needed to clear the zero-filled data area. In turn, this depends on the size of these segments and the speed and memory bandwidth of the server. Table 6-2 shows the size of the kernel segments and the time required to copy or clear them,

Segment	Size (kilobytes)	Time to initialize (seconds)
Text	1059.0	0
Initialized data	76.1	0.015
Zero-filled data	147.6	0.014
Total	1282.7	0.029

Table 6-2. Size and initialization times for Sprite kernel segments.

This table gives the size in kilobytes of the Sprite kernel from a binary image for the SPARCstation-2 and the times to initialize the segments. The text segment requires no time, because it is reused in place. The initialized data segment must be copied to the correct location, and memory pages for the zero-filled segment must be cleared.

as necessary. These measurements assume that the SPARCstation-2 can copy uncached data at five megabytes/second and can zero uncached data twice that fast. (My measurements use an unoptimized byte-copy routine at user-level, so the time should be even less inside the kernel. This step occurs too early in the boot procedure to time it during reboot.) The text segment is reused in place, so there is no need to copy it. The initialized data segment is 76 kilobytes and takes 0.015 seconds to copy. The zero-filled data segment is 148 kilobytes and takes 0.014 seconds to clear, for a total of 0.029 seconds to initialize the kernel image. Reusing the text and initialized data segments allows us to avoid downloading over a megabyte of text and data in Sprite, saving seven to 60 seconds.

Reusing the kernel image may also enable switching to a new version of the kernel with minimal downtime. It should be possible to download a new version of the kernel while executing the previous version and then jump to the location of the new kernel to begin executing it. This would eliminate the downtime suffered by systems when they are upgraded to new versions of the software, but it would necessitate some cooperation and compatibility between the old and new versions of the software. The old version must have the ability to jump to the correct location in the new version. And it must be possible to start execution of the new version at a different address than the address used in the previous version. This has not been implemented in Sprite.

6.2. Using Pre-Computed Values

The second step a Sprite file server executes to reboot is to initialize various system modules. Some of these modules initialize parts of the hardware, and almost all of them set up various software data structures. For example, the virtual memory module sets up the virtual address space by initializing the hardware page tables and filling in software data structures that represent those hardware page tables.

Module initialization is the step I have optimized least, because it is generally the most difficult to optimize. The virtual memory module, for instance, is the slowest module to initialize, but I do not see an easy way to improve it.

However, some other modules are easy to improve, because we can preserve information they need across reboots, so they need not recalculate it. For example, the Sprite timer module calibrates the speed of the machine by idling for five seconds and counting the number of timer ticks that occur during that time. Although this number is different for each machine, it will not change across reboots on the same machine. The first time the kernel boots, it computes the number the old way. It then stores the value in an area of memory that is made read-only. For a fast reboot, the kernel retrieves this value from memory without a need to idle for five seconds.

This technique is very similar to preserving a copy of the initialized data in memory. In fact, I implement the technique by writing the computed value into a variable in the preserved copy of the initialized data. One low-level routine in the kernel makes the appropriate page of the saved initialized data writable, updates the value, and makes the page read-only again. On a fast reboot, we then get the correct value of the variable automatically in the initialized data. However, the first time the kernel boots, the variable in the initialized data segment must have a value that signals the system to execute the necessary computation.

Avoiding this computation in the timer module reduces overall module initialization from 23 to 18 seconds, but there are many other values we could pre-compute in this fashion. A Sprite machine's host ID and Internet address are possibilities. When a Sprite host boots, it currently broadcasts its ethernet address on the network and waits for another machine to respond to it telling it what its host ID or Internet address is. Storing these values across reboots would save the time consumed by this broadcast. With optimizations such as these I believe we could reduce the time needed for module initialization by several more seconds.

Finally, the ability to preserve the values of certain variables across reboots is useful for purposes other than fast recovery. I have found it particularly useful for timing events during start-up. To time reboots, I need to know exactly when they start. When the file server goes through a fast reboot test, its last task before booting is to record the current time in a variable that is preserved across the reboot. Because this is the server's last task, this time is identical to the start of the reboot. I've thus been able to take much more accurate start-up measurements than I could have otherwise.

6.3. Using LFS

The lengthiest step in rebooting most UNIX-like operating systems is checking the consistency of the file system on disk. For file systems such as the BSD Fast File System (FFS) [McKusi84] and the original Sprite file system (OFS), this operation is carried out by the *fsck* file system check program. In Sprite, fsck requires about 40 minutes to check five gigabytes of disk. We have reduced the time to check and recover the file system to about three seconds by switching to Mendel Rosenblum's Log-Structured File System (LFS) [Rosenb91]. It was this switch that made it reasonable to consider speeding up the other aspects of recovery. Without this switch, other improvements would largely be unnoticed.

There are two steps to recovering most file systems [Seltz93a]; checking the physical consistency of the file system on disk, and checking its logical structure. Both fsck and Sprite LFS recovery perform the first step. In FFS and OFS the first step requires a lot of work, because a single write to a file may result in several possible file system updates, all of which are performed at different locations on disk. If the server crashes, some updates may succeed while others may not, leaving the file system physically inconsistent. Fsck must search the contents of the entire file system to find such inconsistencies. For example, a write to a file may add a new block to the file. This may require updating an indirect block of the file, and also the metadata (inode) for the file. Because these operations are not performed atomically, the system may create the new file data block before the crash, but may not assign it to any file. Fsck must find and fix these inconsistencies.

In LFS, by contrast, this first step does not require much work. All file system modifications are made to the end of the log on disk, so it is only necessary to examine the end of the log to find all possible physical inconsistencies. This is done by “rolling forward” from the last checkpoint on the disk. At a checkpoint, the file system is physically consistent. All updates performed after the checkpoint are written to the log on disk and can thus be repeated or undone. The amount of time to roll forward from the log depends on the number of updates performed since the last checkpoint. Three seconds is a generously large estimate for a single disk, and multiple disks will be checked in parallel [Rosenb92].

The second part of recovering the file system is checking its logical structure, which can be corrupted by a media failure. Fsck performs this task, while Sprite LFS recovery does not. For example, a media failure or software bug can corrupt the contents of a directory, so that files become inaccessible. This leaves the file system logically incorrect. The BSD implementation of LFS [Seltz93b] (a more recent LFS implementation than Sprite’s) considers this an important flaw and thus performs the logical consistency check in background after roll-forward recovery from the log. This allows applications to begin running as soon as the roll-forward is done, while the system eventually checks the logical consistency of the entire file system.

A background check of the Sprite LFS directory structure at recovery time is possible, but has not been implemented for several reasons. First, media errors usually occur at the location of the most recent update. In this case they can quickly be found during LFS’s roll-forward recovery. Second, other media errors can occur at any time anyway; if it is important to check for these, the system should do this as a periodic background check during normal operation. The checks could be performed during periods of low load, to avoid degrading performance. Third, checking the logical structure of the file system will not uncover media errors that affect only the contents of file data blocks, rather than the logical consistency of the file system. Fsck does not detect this kind of file data corruption. Finally, in highly reliable systems, disks are monitored for soft (correctable) errors. If these become frequent, the disk should be replaced. By monitoring the error trends it is usually possible to replace the faulty disk before it causes a permanent error [Lin90].

There are two remaining problems with the Sprite implementation of LFS that have an effect on the measurements in this dissertation. The first is that we lack an LFS disk boot program. Without this boot program, we are obliged to boot Sprite from an OFS file system or across the network. For this reason, I measured the disk boot times with the OFS disk boot program on the main Sun-4/280. The second remaining problem is that Sprite does not actually use LFS roll-forward recovery, because it still has bugs. I have compensated for this problem by adding the three seconds for roll-forward recovery to the reboot times listed in this thesis. Sprite currently just checkpoints the file system every 30 seconds and recovers using the last checkpoint.

6.4. Recovering Distributed State

The previous chapters of this thesis describe how I’ve improved distributed state recovery times. Before implementing congestion control in the RPC system and otherwise optimizing the recovery protocol, client-driven recovery was unstable and could take many minutes to complete. Sometimes it did not complete at all. With the improvements described in chapters 3-5, client-driven recovery now takes no more than 30 seconds in the testbed setup. Other recovery techniques are even faster, with the fastest being transparent recovery. Given its per-client costs, transparent recovery should less than six seconds in the testbed setup with 40 clients. Further improving distributed file system state recovery in Sprite requires reducing the amount of disk I/O the server performs.

6.5. Delaying Daemon Start-Up

A final boot step that permits some performance improvement is the start-up of various kernel processes and user-level daemons. For this step I have not actually reduced the amount of work that must be done but have chosen to delay some of it until after the server finishes rebooting. Delaying the start-up of some user-level daemon processes allows the most necessary processes to start up more quickly. For example, we can delay the line printer daemon, the tftp daemon, sendmail, cron, the migration daemon, and the boot daemon for a few seconds with no harm. This allows us to start up the necessary daemons (such as Sprite's user-level IP server) eight seconds faster. Unfortunately, most kernel processes are immediately necessary, so they cannot be delayed. For example, without the RPC server processes in the file server kernel, no requests from clients can be handled.

While the choice of what to delay will be different on different systems, the important point is that not all start-up work must be done during reboot. If any of the start-up work can be done "lazily," reboot will be faster. By delaying less crucial services, we improve overall availability. This is one of the most generally useful techniques for increasing recovery speed.

6.6. Further Improving Reboot Times

After the improvements described in this chapter, the largest amount of time during reboot is now spent initializing kernel modules. To gain further significant improvements, we need to reduce this amount of time from its current 18 seconds. Unfortunately, this is the hardest area to improve with general techniques, because different modules do different sorts of things for initialization. For example, some modules must initialize hardware and are hampered by the response time of these devices. As another example, the virtual memory module must initialize software and hardware page tables for a large amount of memory. Neither of these examples is easily improved by the techniques I have described, such as storing pre-computed values. Finding techniques to improve these initialization times will greatly benefit fast recovery.

6.7. Summary

To provide fast crash recovery, we must reduce the time required by the complete server reboot sequence, and not just distributed state recovery. Combining the improvements described in this chapter, a Sprite file server can now recover in under 29 seconds. Using LFS to avoid a lengthy file system check contributes the largest improvement. The next largest improvement comes from the distributed state recovery techniques described in previous chapters. In addition, Sprite file servers reuse the text and initialized data segments to avoid downloading new kernel images, and they preserve pre-computed values across reboots to avoid time-consuming initialization.

7 Designing for Fast Crash Recovery

This chapter contains some of the lessons I have learned about designing systems to recover quickly after crashes. The overall lesson is that it is easier to design fast recovery into a system from the beginning than it is to fix an existing system that suffers from poor recovery performance. The comments in this chapter come from my experience fixing Sprite file servers to recover quickly, but most of the suggestions are applicable to other systems as well. The sections below deal with issues such as compensating for the stress that recovery places on a system, avoiding start-up overhead, using main memory to store data across failures, choosing and maintaining distributed state information, and debugging recovery code.

7.1. Recovery Stress

One of the first lessons learned from fixing Sprite's file server crash recovery is that recovery places a greater stress on the system than its normal activities. As the Sprite system grew to include more machines and users, the first thing to break under the increased load was crash recovery. As described in chapter 3, the problem was a lack of negative acknowledgments in the RPC system. This was a general flaw, but it surfaced first during the communication load caused by client-driven recovery storms.

There are two reasons that recovery is stressful to a system. The first is that the server is likely to suffer a high load during recovery. Designers must account for this when planning the capacity of a system. Harp and other systems with backup servers provide an example. To avoid wasting resources during normal execution, all backup servers in Harp are primary servers for their own file systems. After the failure of a primary server, its backup must handle its own load plus the load of the failed primary. If the backup is already loaded during normal execution, the load during recovery may cause severe performance problems. It is necessary to make a trade-off between wasted capacity during normal system execution, and slow and perhaps unstable performance during recovery.

A second reason that recovery is stressful for systems is that it is less well tested than other system activities. Recovery almost always exercises some special-purpose code. Because this code is only exercised during recovery, it is more likely to be faulty, and recovery itself is more likely to fail or cause damage.

Thus it seems like a good idea to design system recovery to use only commonly executed code, but there is a trade-off involved. The problem is that commonly executed code is less efficient for recovery, since it is not optimized solely for that task. We find an example of this trade-off in Sprite. Client-driven recovery executes less special-purpose code than server-driven and transparent recovery, but it is also the slowest form of recovery. Adding a reasonable amount of special-purpose code gives us server-driven recovery, which is faster and eliminates the cache consistency violations seen in client-driven recovery. Transparent recovery is the fastest technique. However, it uses the most special-purpose code, and therefore requires much more testing and debugging than client-driven recovery.

7.2. Start-Up Overhead

Avoiding start-up overhead is another pre-requisite for fast recovery. A common design trade-off is to perform extra processing during system start-up in order to avoid extra processing later, during normal execution. This is often the correct trade-off, but the amount of start-up processing adds up quickly, and can be prohibitive for fast recovery. To avoid this overhead, designers can consider each part of system start-up, to see if it can be delayed, optimized, or eliminated completely.

There are examples of each of these three choices in Sprite. First, I delay the start-up of some user-level daemons until file server recovery finishes, allowing crucial services to start up faster. Delaying start-up work is perhaps the most generally applicable and easiest technique. Second, I optimize the initialization of some system modules by pre-computing and storing information they need. Finally, Sprite eliminates some overhead completely by using LFS rather than a traditional UNIX-like file system. Using LFS allows us to skip the traditional consistency check of the file systems in favor of quicker recovery from a short log.

7.3. Using Main Memory

Using main memory to store data across crashes is another example of a technique that reduces start-up overhead while providing good overall performance. More systems should consider this option, especially as decreasing RAM costs make large memories more attractive. Sprite recovery uses main memory for several recovery tasks. The recovery box, used in transparent recovery, helps avoid communication overhead between clients and servers, can be used to avoid some disk I/O, and adds negligible processing during normal execution of the system. Sprite also reuses its kernel text and initialized data segments from main memory after a failure. This eliminates the overhead of downloading the kernel image. Using main memory this way is unconventional, and is not guaranteed to work after all failures. But it provides the fastest possible recovery with the least overall performance loss during normal execution.

7.4. Maintaining Correct Distributed State Information

Maintaining correct state information is a problem more systems designers will encounter as they opt for the higher performance of stateful systems. Keeping extra state close to where it is needed can provide huge performance benefits. However, maintaining state information also requires extra processing overhead, extra space to store the state, and extra complexity to keep it consistent and recover it correctly after failures. Thus it is a good idea to minimize the amount of state a system keeps. Designers must choose what and how much state to store, where to store it, when to update it, and how to retrieve it after a failure. In particular, handling state is the trickiest recovery-related task. This section covers some of the lessons I've learned about recovery of stateful systems.

7.4.1. Managing State Recovery With More State

While minimizing the amount of necessary state information is usually a good idea, sometimes the solution to the state recovery problem is to maintain even more state. There are many examples of this phenomenon in Sprite. The first obvious example is distributed cache state. For high performance, Sprite allows clients to cache file data in their main memories. To provide cache consistency, the server keeps extra state information that records which clients are caching which files. After a failure, the server must recover its distributed cache state to continue providing cache consistency. The solution to this recovery problem is to use distributed state. With client-driven or server-driven recovery, we can recover the server's state by retrieving the clients' state. Or we can improve recovery times by using transparent recovery, which maintains even more state on the file server, in a recovery box.

Another example of this phenomenon concerns the reliability of data after a server failure.The file data cached by clients and the server is itself a form of distributed state. On many UNIX systems, newly written data is vulnerable to loss for the length of a cache write interval (about 30 seconds). But on Sprite, new data from clients is vulnerable for the length of two cache write intervals. When clients modify files, they cache the dirty data for about 30 seconds before writing it back to the server. The server then caches this dirty data for another 30 seconds before writing it to disk. This new data will be lost if the clients reuse these cache blocks and the server crashes before writing the data to disk. A solution to this problem is to keep more state on the clients. The clients can refuse to reuse dirty cache blocks until the server writes the data to disk. To do this, the clients can either set aside the cache blocks for the cache write interval of the server, or they can wait until the server issues a call-back indicating the data is safe. Either way, the clients must mark which cache blocks may be lost during a failure. Using this extra state information, they can then resend the appropriate data to the server as part of recovery.

7.4.2. Careful Bookkeeping

Another lesson I've learned is that the fastest recovery technique for stateful systems, transparent recovery, requires more careful bookkeeping of state information than pre-existing systems may provide. For better performance, transparent recovery eliminates communication between the clients and server during recovery, but it is harder to keep distributed state consistent without this extra communication. This problem is further explained in chapter 5, section 5.5. In Sprite, the cache state maintained on the clients and the copy on the server can diverge. In client-driven and server-driven recovery, this is usually not a problem, because the server refreshes its copy of the state from that of the clients', ensuring consistency of state information after every failure. But with transparent recovery we lose this opportunity to refresh the server's state information. Thus transparent recovery requires careful bookkeeping of such state as reference counts on files. Ensuring the consistency of file reference counts requires hanging client close RPCs across server failures, so the server can process the close requests after recovery.

Transparent recovery brings up another bookkeeping problem. As described in section 5.6.1, Sprite servers handle directory modifications asynchronously, so the server's and clients' view of the file system may diverge if a server crashes before issuing the necessary disk I/O. Because the server does not communicate with the clients during transparent recovery, it loses the opportunity to refresh its directory state from that of the clients. The solution to this problem is to keep more state on the server, by logging directory operations to the recovery box. This allows the server to finish the operations after it reboots.

7.4.3. Tools for Debugging Distributed State

In the course of working with Sprite, I've found two techniques that make management and debugging of distributed state information easier. First, designers should develop good tools for examining state information. Second, it must be possible to clean up or destroy state information. With the proper tools, stateful systems need not be more mysterious than stateless ones.

To debug how the system handles state information, it must be possible to determine what state exists at any time. Therefore, a user-level program that gathers a snapshot of the distributed state is essential. However, developing such a tool for an existing system can be difficult. In Sprite, for example, it is not always possible to view all the file system state on a machine, because some I/O handles may be locked by parts of the kernel, and their contents cannot be fetched until they are unlocked. Unfortunately, it is often exactly these I/O handles that the person debugging the system most wishes to examine. When designing a system from scratch, it should be possible to develop a mechanism to step around normal locks and minimize the times when it is impossible to get a complete snapshot of important distributed state information.

A second technique that makes debugging a stateful system easier is the ability to destroy state. For example, it should be possible to force a client workstation to give up all references to resources on a server. This ability makes it much easier to test and retest operations that accumulate or change distributed state. This in turn makes it possible to provoke and isolate state-related problems, such as circular resource dependencies amongst machines. In Sprite, such a tool must be able to make a client do at least three things. First, the client must close all files and other objects served by a particular server. Second, it must release its references to any file system prefix serviced by that server. Third, it must delete that server from its recovery state information. This is the state information clients use to determine which servers to monitor for failures and reboots.

7.4.4. Isolating State Changes

While working with Sprite, I have also learned that managing distributed state is much easier if the code that updates it is well-defined and easy to isolate. This sounds simple, but may not be in a pre-existing system. In Sprite for example, it is not easy to isolate where files and other objects are opened. This is because there are several tasks associated with opening a file, and they are performed in different places and at different times: incrementing the file I/O handle reference count and creating the stream handle for it are not executed atomically. An error at any point in the code that opens a file may invalidate previous state changes. Dealing with this was one of the trickiest parts of implementing transparent recovery. Modifying a pre-existing system to use transparent recovery may necessitate rearranging some of the code that executes such state changes, so that state updates to the recovery box can be made atomic, or at least confined to only a few places.

When implementing a system from scratch, designers should try to isolate code that changes state they may wish to recover.

7.5. Debugging Recovery Code

Finally, one of the difficulties in debugging recovery code is that it is hard to force the system into all of the possible states that can occur during and after crashes. One of the reasons for a crash is that something unexpected happens in the system, so it is hard to predict the existence of the state that causes the crash. Nonetheless, it is worth testing recovery under many different conditions. A helpful tool is a test suite that drives the system through different states and causes it to recover from different conditions. It is hard to guarantee the completeness of such a test suite, but this does not negate its usefulness.

7.6. Summary

System designers generally acknowledge that failure recovery should be included as part of the original design of a system, but in practice the attention paid to recovery in first designs may be cursory. This is a pity, since choices made early on may render fast recovery harder to implement later. Examples of such choices are planning the capacity of the system to accommodate recovery loads, avoiding start-up overhead, using main memory for recovery information, and determining what state information to preserve across failures, how to maintain it, and when to update it. There are several techniques to keep in mind that will save a lot of time during reboot without hurting overall system performance. Examples in this thesis are choosing a file system that requires little consistency checking during start-up, and saving state information in main memory rather than retrieving it from disk or from other hosts. Table 7-1 lists some of these techniques.

Fast crash recovery design techniques
Consider recovery issues early in the design process
Design recovery to handle extra stress
Avoid start-up overhead
Use main memory to avoid communication and disk I/O costs
Manage state recovery with more state
Use careful bookkeeping of state information
Build tools for debugging distributed state
Isolate state changes in the code
Use a crash recovery test suite
Use a file system that requires no disk consistency check

Table 7-1. Fast crash recovery design techniques.

This table lists some of the techniques described in this thesis for designing systems that recover quickly from crashes.

8 Conclusion

High-performance distributed file systems keep more state information than their predecessors. Unfortunately, this trend makes failure recovery slower and more complex, because these systems must recover the distributed state. To combat the problem, we need techniques to make state recovery fast and manageable. With that goal, this thesis describes and evaluates three distributed state recovery techniques for file servers and lists some lessons I've learned about designing systems to provide fast recovery.

The fast recovery approach is useful for more than state recovery. It is an alternative for providing increased availability in environments where traditional fault-tolerant techniques are too costly, slow, or complex. If non-stop availability is not essential, then we can increase overall system availability with fast crash recovery. The goal of fast crash recovery is not to prevent or mask failures, but instead to recover from them so quickly that they are not noticed. Therefore this thesis lists all of the fast crash recovery techniques I've used in the Sprite distributed system to make it possible to recover our file servers in under 30 seconds.

This concluding chapter summarizes the results of this thesis, lists future work in this area, and makes some final comments about this project.

8.1. Results

With the techniques described in this thesis, Sprite now recovers three orders of magnitude more quickly than most distributed file systems. Most of this improvement comes from switching to the LFS file system and using server-driven or transparent distributed state recovery.

Table 8-1 summarizes the trade-offs between the three distributed state recovery techniques described in this thesis: client-driven, server-driven, and transparent recovery. Client-driven recovery requires the least amount of special-purpose code, because clients initiate recovery using the usual request/response RPC path. However, client-driven recovery is also the most prone to congestion on the server and may require careful optimization to scale to large numbers of clients. It also necessitates a trade-off between slower recovery and cache consistency violations. In client-driven recovery, avoiding cache consistency violations requires a waiting period after recovery before the server begins accepting new client requests. Even without this extra waiting period, client-driven recovery is the slowest technique of the three: in the test-bed setup it takes 21 seconds to recover ten clients on average and can easily take up to 30 seconds.

Recovery technique	Average recovery speed (seconds)	Comments
Client-driven	21	+ Least special-purpose code for client/server RPC systems. – Tends to cause server load and congestion. – Requires trade-off between slower recovery and cache consistency violations. – Slowest recovery (can be at least as long as *ping interval*).
Server-driven	2.0	+ Few modifications beyond client-driven. + Fixes caches consistency problems. + Faster recovery. + More server control over congestion. – Tricky synchronization and locking in kernel – Only as fast as the slowest client. – May require client-driven recovery as backup.
Transparent	1.5	+ Recovery box useful for applications too. + Fixes cache consistency violations. + Fastest recovery. + Allows recovery of directory operations too. ± Unconventional use of main memory. – Most special-purpose code. – Tricky bookkeeping. – May require client-driven recovery as backup.

Table 8-1. Evaluation of distributed state recovery techniques.

This table summarizes the trade-offs among the three distributed state recovery techniques evaluated in this thesis. The second column gives the time in seconds to recover 10 clients of a SPARCstation-2 server, with the average client state setup of Table 3-5. The third column lists advantages (marked with a '+') and disadvantages (marked with a '–') of the techniques.

Server-driven recovery is a modification of client-driven recovery that allows the server to control recovery. It eliminates cache consistency violations, gives the server more congestion control, and is faster. Server-driven recovery takes only two seconds on average for ten clients.

Transparent recovery is the fastest and most novel technique of the three, because the server recovers its distributed state information from its own main memory without the need to communicate with clients. Transparent recovery takes only 1.5 seconds on average for ten clients. It eliminates cache consistency violations and also allows the server to log directory modifications. Using this log, the server can ensure that no directory modifications are lost in a failure, rather than just report the errors to clients during recovery. Transparent recovery uses main memory in an unconventional way – as stable storage to preserve state information across failures. The recovery box can be used by applications as well as the kernel. The main disadvantage of transparent recovery compared to the other techniques is that it requires the most special-purpose code and careful bookkeeping of state information.

8.2. Future Work

There are far more topics related to fast recovery than are covered by this thesis. This section lists a few of these topics.

8.2.1. Other Metrics for Comparing State Recovery Techniques

This dissertation has compared distributed state recovery techniques based upon speed, complexity and reliability, but there are other useful metrics. System security is one example. Client-driven and server-driven recovery may offer less system security than transparent recovery, because the server must trust the state information sent to it by clients. A system with significant security requirements would need to authenticate the clients before they recover their state information. This could be a very costly operation. Perhaps transparent recovery can provide this extra security at less cost.

8.2.2. Long-Term Evaluation of the Recovery Box

While this thesis includes an initial evaluation of the recovery box, its true value must be determined over time. Sprite has not used transparent recovery enough for us to know whether it provides fast restart after the majority of failures, or whether too many failures corrupt the recovery box. If the recovery box works for most failures, then its extra implementation effort is worthwhile. But if the recovery box becomes corrupted frequently or the data it stores is susceptible to error propagation, then the technique is not interesting in this environment. Measurements over a period of several months would help answer this question. It would be useful if these measurements were also performed in systems other than Sprite – systems intended more for production use and less for research.

8.2.3. Long-Term Evaluation of Fast Recovery

Along similar lines, a long-term evaluation of fast crash recovery is necessary. We do not yet know whether fast crash recovery significantly improves availability over a long period of time. It is conceivable that we find permanent hardware failures occur frequently enough that only redundant hardware can significantly improve availability. For this reason we cannot yet be sure of the range of environments that will find fast crash recovery preferable to traditional fault tolerant techniques.

8.2.4. Evaluation of Complexity

This thesis also has not addressed the issue of how to evaluate the relative complexity of systems using the different recovery techniques. Intuitively, I believe the recovery box is simple compared to distributed systems that use a high level of redundancy to mask failures. But I have found no good metrics to prove this. Measuring lines of code is not satisfactory, and other metrics I've found do not account for the complexity of shared state and control paths between software modules.

8.2.5. Side-Effects of Fast Recovery

Another question about fast recovery is whether it has any negative side-effects. It is possible that very fast recovery could result in gradually buggier code. If recovery is fast enough, then users may be more likely to tolerate software bugs that cause infrequent failures. These are often the most difficult bugs to find and fix, so fast recovery from these failures may be the most economical solution. However, it is unnerving to consider the acceptance of buggier code. Certainly, if the number of bugs increases to the point where failures are frequent, even fast recovery will not suffice. The bugs will have to be found and fixed. It would be interesting to investigate the range and frequency of failures that users will tolerate if the system recovers very quickly.

8.2.6. Improved Crash Diagnostics

A similar issue concerns crash diagnostics for systems with fast recovery. If a system recovers quickly and automatically from crashes, it may be hard to debug the problem and determine the cause of the failures. It is possible to set a parameter in the running system that prevents it from doing a fast reboot after the next failure. This provides an opportunity after the crash to examine the system. However, this approach will not significantly reduce downtime. If the system instead records detailed crash diagnostics automatically, it might be possible to debug it without making it unavailable for long periods of time. Future work in this area should help provide diagnostics beyond just the type of fault and the location in the code where it occurred.

8.2.7. Using the Recovery Box for Applications

While this thesis concentrates on fast restart of operating systems, the fast restart of applications is also important. This is particularly true for those applications that act as servers and must be started up soon after system reboot. It would be interesting to investigate the recovery box's use for a variety of applications. This thesis describes Mark Sullivan's use of the recovery box for fast restart of the POSTGRES database, but I have not explored its use for any other applications. The most obvious applications that might benefit are those providing network services. These applications could use the recovery box in the same way that file servers do to avoid disk I/Os and communication with clients during recovery.

An example of such an application is a network name server, such as the Berkeley Internet Name Domain Server [Dunlap86]. A name server is a distributed database that allows clients to name objects in a distributed system. The server is a daemon called *named* that responds to queries on a network port. Given the name of an object, such as a host on the Internet, the server responds with an address at which one can access the object. Some of the servers store this name-to-address translation information on disk and read it in during start-up. Others obtain the information from another server and recheck with that server periodically to see if their data is still correct. Either of these types of servers could store the name-to-address translation data in the recovery box. During start-up, the daemon could then retrieve the information from the kernel's memory rather than from disk or another server. To add, delete or change an address translation, the daemon would update the kernel's main memory through the recovery box system call interface.

8.2.8. Fast Application Page-In

Another fast restart issue for applications is optimizing their initial page-in. Some applications must page in a large amount of text and initialized data before continuing with their normal operations. The problem is particularly severe for some applications written in object-oriented languages such as C++. In these applications many data structures have start-up code associated with them. The code and the data are not necessarily located contiguously in the disk image and may require many disk seeks to be loaded into the application's in-memory image. It would be interesting to investigate compiler enhancements that lay out the application's code and data with the goal of reducing start-up times.

8.3. Final Comments

I am very fortunate to have had the Sprite operating system available for this work. Sprite has proven to be more flexible, robust, and well-documented than I could have expected. And while there have been many frustrations associated with modifying Sprite to behave in ways not contemplated by its original designers, I would never have finished this work without an existing system as a foundation.

As part of modifying Sprite, I've learned the value of real implementations for judging one's research ideas. Nothing is as simple as it seems. This is why paper designs and simulations frequently overlook some feature that is required in a real implementation and that qualitatively changes the results. An example from this thesis is the implementation of the recovery box. My original design entirely overlooked the problem of asynchronous directory modifications.

Finally, I hope that future systems will consider recovery as an important part of system design. File server technology has moved beyond stateless servers and now requires more attention to recovery. Unfortunately, relatively few systems have paid much attention to recovering quickly, and fast recovery is harder to implement if it is not designed in from the beginning. It is human nature to believe that the system you design will not fail often, but it is also human nature to make mistakes. Fast failure recovery may make these mistakes more tolerable in the long term.

Bibliography

[Accett86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. "Mach: A New Kernel Foundation for UNIX Development." *Proceedings of the Summer 1986 USENIX Conference*, pages 93-112, June, 1986.

[Anonym88] Anonymous. "A Measure of Transaction Processing Power." Tandem Corporation Technical Report 85.1. In *Readings in Database Systems*, edited by Michael Stonebraker, pages 300-312. Morgan Kaufmann Publishers, San Mateo, California, 1988.

[Babaog90] Ozalp Babaoglu. "Fault-Tolerant Computing Based on Mach." *ACM Operating Systems Review*, pages 27-39, January, 1990.

[Baker91a] Mary Baker and John Ousterhout. "Availability in the Sprite Distributed File System." *ACM Operating Systems Review*, pages 95-98, April, 1991.

[Baker91b] Mary Baker, John Hartman, Michael Kupfer, Ken Shirriff and John Ousterhout. "Measurements of a Distributed File System." *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 198-212, October, 1991.

[Baker92a] Mary Baker and Mark Sullivan. "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment." *Proceedings of the Summer 1992 USENIX Conference*, pages 31-43, June, 1992.

[Baker92b] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. "Non-Volatile Memory for Fast, Reliable File Systems." *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10-22, October, 1992.

[Banatr89] J. P. Banatre, M. Banatre, G. Muller. "Architecture of Fault-Tolerant Multiprocessor Workstations." *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 20-24, September, 1989.

[Bartle81] J. Bartlett. "A NonStop Kernel." *Proceedings of the Eighth Symposium on Operating Systems Principles*, December, 1981.

[Bartle90] Wendy Bartlett, Richard Carr, Dave Garcia, Jim Gray, Robert Horst, Robert

Jardine, Dan Lenoski, Dix McGuire, and Joel Bartlett. "Fault Tolerance in Tandem Computer Systems." Tandem Technical Report 90.5, Tandem Part Number 40666, March, 1990.

[Birman84] Kenneth P. Birman, Amr El Abbadi, Wally Dietrich, Thomas Joseph, and Thomas Raeuchle. "An Overview of the Isis Project." Technical Report number TR 84-642, Department of Computer Science, Cornell University, October 1984.

[Birman89] Kenneth Birman and Keith Marzullo. "The ISIS Distributed Programming Toolkit and the Meta Distributed Operating System." *SUN Technology*, volume 2, number 1, pages 90-104, Summer 1989.

[Birman91] Kenneth Birman, Andre' Schiper, and Pat Stephenson. "Lightweight Causal and Atomic Group Multicast." *ACM Transactions on Computer Systems*, volume 9, number 3, pages 272-314, August, 1991.

[Birrel82] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. "Grapevine: An Exercise in Distributed Computing." *Communications of the ACM*, volume 25, number 4, pages 260-274, April, 1982.

[Birrel87] [Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber. "A Simple and Efficient Implementation for Small Databases." *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 149-154, 1987.

[Blasge79] M. Blasgen, J. Gray, M. Mitoma, and T. Price. "The Convoy Phenomenon." Operating Systems Review, volume 13, number 2, April, 1979.

[Borg83] Anita Borg, Jim Baumbach, and Sam Glazer. "A Message System Supporting Fault Tolerance." *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 90-99, November 1983.

[Borg89] A. Borg and W. Blau and W. Graetsch and F. Herrman and W. Oberle. "Fault Tolerance Under UNIX." *ACM Transactions on Computer Systems*, volume 7, number 1, February, 1989.

[Bruell88] G. Bruell, A. Z. Spector, R. Pausch. "Camelot, a Flexible, Distributed Transaction Processing System." *Thirty-third IEEE Computer Society International Conference* (COMPCON), pages 432-437, March 1988.

[Cherit84] D. R. Cheriton. "The V Kernel: a Software Base for Distributed Systems." *IEEE Software*, volume 1, number 2, pages 19-42, April 1984.

[Cherit93] David R. Cheriton and Dale Skeen. "Understanding the Limitations of Causally and Totally Ordered Communication." *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, December, 1993.

[Dasgup88] P. Dasgupta, R. LeBlanc, W. Appelbe. "The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work." *Eighth International Conference on Distributed Computing Systems*, San Jose, CA, pages 13-17, June 1988.

[DeWitt84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R.

Stonebraker, David Wood. "Implementation Techniques for Main Memory Database Systems." *Proceedings of the 1984 ACM SIGMOD Annual Meeting*, Boston, MA, June 1984.

[Dougli91] F. Douglis and J. Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation." *Software − Practice and Experience*, volume 21, number 8, August, 1991.

[Dunlap86] Kevin J. Dunlap. "Name Server Operations Guide for BIND." Release 4.3, Unix System Manager's Manual. Printed by the USENIX Association. April, 1986.

[Gait90] Jason Gait. "A Safe In-Memory File System." *Communications of the ACM*, volume 33, number 1, pages 81-86, January, 1990.

[Garcia92] Hector Garcia-Molina and Kenneth Salem. "Main Memory Database Systems: An Overview." *IEEE Transactions on Knowledge and Data Engineering*, volume 4, number 6, December, 1992.

[Giffor88] David K. Gifford, Roger M. Needham, and Michael D. Schroeder. "The Cedar File System." *Communications of the ACM*, volume 31, number 3, pages 288-298, March 1988.

[Gray89] Cary G. Gray and David R. Cheriton. "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency." *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 202-210, December, 1989.

[Gray88] Jim Gray. "The Transaction Concept: Virtues and Limitations." *Lecture Notes in Computer Science*, Conference Proceedings, Netherlands, 1981. In *Readings in Database Systems*, edited by Michael Stonebraker, pages 140-150. Morgan Kaufmann Publishers, San Mateo, California, 1988.

[Gray90] Jim Gray. "A Census of Tandem System Availability Between 1985 and 1990." Tandem Technical Report 90.1. Part number 33579, January, 1990.

[Gray93] Jim Gray, and Andreas Reuter. "Transaction Processing: Concepts and Techniques." Morgan Kaufmann Publishers, 1993.

[Hagman86] Robert B. Hagmann. "A Crash Recovery Scheme for a Memory-Resident Database System." *IEEE Transactions on Computers*, volume 35, number 9, September 1986.

[Hartma93] John H. Hartman and John K. Ousterhout. "The Zebra Striped Network File System." *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, December, 1993.

[Haskin88] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. "Recovery Management in QuickSilver." *ACM Transactions on Computer Systems*, volume 6, number 1, pages 82-108, February, 1988.

[Hisgen89] Andy Hisgen, Andrew Birrell, Timothy Mann, Michael Schroeder and Garret Swart. "Availability and Consistency Tradeoffs in the Echo Distributed File

System." *Proceedings of the Second Workshop on Workstation Operating Systems*, September, 1989.

[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." *ACM Transactions on Computer Systems*, volume 6, number 1, pages 51-81, February, 1988.

[Jewett91] D. Jewett. "Integrity S2: A Fault-Tolerant UNIX Platform." *Digest 21st International Symposium on Fault-Tolerant Computing*, June,1991.

[Johnso93] Paul Johnson. Personal Communication. June 18, 1993.

[Juszcz89] Chet Juszczak. "Improving the Performance and Correctness of an NFS Server." *Proceedings of the Winter 1989 USENIX Conference*, pages 56-63, February, 1989.

[Kazar90] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu and Edward R. Zayas. "DEcorum File System Architectural Overview." *Proceedings of the Summer 1990 USENIX Conference*, pages 151-163, June, 1990.

[Kazar93] Michael L. Kazar. Personal Communication, June, 1993.

[Kistle91] James J. Kistler, and M. Satyanarayanan. "Disconnected Operation in the Coda File System." *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 213-225, October, 1991.

[Kleima86] S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX." *Proceedings of the Summer 1986 USENIX Conference*, pages 238-247, June, 1986.

[Lai89] Nick Lai. Personal Communication, Fall, 1989.

[Lin90] Ting-Ting Y. Lin and Daniel P. Siewiorek. "Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis." *IEEE Transactions on Reliability*, volume 39, number 4, October, 1990.

[Liskov88] Barbara Liskov. "Distributed Programming in Argus." *Communications of the ACM*, pages 300-312, March 1988.

[Liskov91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. "Replication in the Harp File System." P*roceedings of the Thirteenth Symposium on Operating Systems Principles*, October, 1991.

[Long91] D. Long, J. Carroll and C. Park. "A Study of the Reliability of Internet Sites." *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, September, 1991.

[Mackle91] Rick Macklem. "Lessons Learned Tuning the 4.3 BSD Reno Implementation of the NFS Protocol." *Proceedings of the Winter 1991 USENIX Conference*, pages 53-64, January, 1991.

[Mann93] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. "A Coherent Distributed File Cache with Directory Write-behind." Digital SRC Research Report Number 103, June, 1993.

[McKusi84] M. McKusick, W. Joy, S. Leffler, and R. Fabry. "A Fast File System for UNIX." *ACM Transactions on Computer Systems*, volume 2, number 3. pages 181-197, August, 1984.

[Mogul92] Jeffrey C. Mogul. "A Recovery Protocol for Spritely NFS." *USENIX File Systems Workshop Proceedings*, pages 93-109, May, 1992.

[Mogul93] Jeffrey C. Mogul. "Recovery in Spritely NFS." Digital Western Research Laboratory Research Report 93/2, June, 1993.

[Muelle83] E. Mueller, et al. "A Nested Transaction Mechanism for LOCUS." *Proceedings of the Ninth Symposium on Operating System Principles*, October, 1983.

[Nelson88] M. Nelson, B. Welch, and J. Ousterhout, "Caching in the Sprite Network File System." Transactions on Computer Systems, volume 6, number 1, pages 134-154, February, 1988.

[Ouster88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. "The Sprite Network Operating System." *IEEE Computer*, volume 6, number 1, pages 23-36, February, 1988.

[Ouste90a] John K. Ousterhout. "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" *Proceedings of the Summer 1990 USENIX Conference*, pages 247-256, June, 1990.

[Ouste90b] John K. Ousterhout. "The Role of Distributed State." *Proceedings of the 25th Anniversary Symposium, School of Computer Science*, Carnegie Mellon University, September, 1990.

[Patter88] David A. Patterson, Garth Gibson and Randy H. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)." *Proceedings of the 1988 ACM SIGMOD*, pages 109-116, June, 1988.

[Popek85] Gerald J. Popek and Bruce J. Walker, editors. "The LOCUS Distributed System Architecture." The MIT Press, Cambridge, Massachusetts, 1985.

[Pu86] C. Pu, J. D. Noe, and A. Proudfoot. "Regeneration of Replicated Objects: A Technique and its Eden Implementation." *Proceedings of the Second International Conference on Data Engineering*, pages 175-187, February 1986.

[Reed81] D. Reed and L. Svobodova. "SWALLOW: A Distributed Data Storage System for a Local Network." In *Networks for Computer Communications*, pages 355-373, North-Holland, Amsterdam, 1981.

[Reuter84] Andreas Reuter. "Performance Analysis of Recovery Techniques." *ACM Transactions on Database Systems*, volume 9, number 4, pages 526-559, December, 1984.

[Rodehe91] Thomas L. Rodeheffer and Michael D. Schroeder. Automatic Reconfiguration in Autonet." *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 183-197, October, 1991.

[Rosenb91] M. Rosenblum and J. K. Ousterhout. "The Design and Implementation of a Log-Structured File System." *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 1-15, October, 1991. Also published as *Transactions on Computer Systems*, volume 10, number 1, pages 26-52, February, 1992.

[Rosenb92] Mendel Rosenblum. Personal Communication. 1992.

[Salem86] Kenneth Salem and Hector Garcia-Molina. "Crash Recovery Mechanisms for Main Storage Database Systems." CS-TR-034-86, Princeton University, Princeton, NJ, 1986.

[Sandbe85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. "Design and Implementation of the Sun Network Filesystem." *Proceedings of the Summer 1985 USENIX Conference*, pages 119-130, June, 1985.

[Satyan90] M. Satyanarayanan. "Scalable, Secure, and Highly Available Distributed File Access." *IEEE Computer*, volume 23, number 5, May, 1990.

[Satyan93] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. "Lightweight Recoverable Virtual Memory." *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, pages 146-160, December, 1993.

[Seltz93a] Margo Seltzer. "File System Performance and Transaction Support." Ph.D. Thesis, Computer Science Division, U. C. Berkeley. Available as Electronics Research Laboratory, College of Engineering, University of California at Berkeley Memorandum No. UCB/ERL M93/1, January, 1993.

[Seltz93b] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. "An Implementation of a Log-Structured File System for UNIX." *Proceedings of the Winter 1993 USENIX Conference*, pages 307-326, January, 1993.

[Siegel89] Alex Siegel, Kenneth Birman, and Keith Marzullo. "Deceit: A Flexible Distributed File System." Technical report 89-1042. Department of Computer Science, Cornell University, 1989.

[Siewio92] Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems*. DEC Press, 2nd edition, 1992.

[Smith81] W. B. Smith and F. T. Andrews. "No. 5ESS Overview." *Proceedings of the Tenth International Switching Symposium*, Montreal, Canada, September, 1981.

[Specto85] A. Z. Spector, J. Butcher, D. S. Daniels, D. J. Duchamp, J. L. Eppinger, C. E. Fineman, A. Heddaya, and P. M. Schwarz. "Support for Distributed Transactions in the TABS Prototype." *IEEE Transactions on Software Engineering*, volume 11, number 6, pages 520-530, June 1985.

[Sriniv89] V. Srinivasan and Jeffrey C. Mogul. "Spritely NFS: Experiments with Cache-

Consistency Protocols.” *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 45-57, December, 1989.

[Stoneb86] M. Stonebraker and L. Rowe. “The Design of POSTGRES.” *Proceedings of the Fifth ACM SIGMOD Conference*, June, 1986.

[Stoneb87] M. Stonebraker. “The Design of the POSTGRES Storage System.” *Readings 13th International Conference on Very Large Databases*, Brighton, England, September, 1987.

[Stratu89] Stratus Computer, Inc. “VOS Transaction Processing Facility Guide.” Part Number R215-00, November, 1989.

[Sulliv90] Mark Sullivan. Unpublished survey of software errors reported in 4.1 and 4.2 BSD UNIX. 1990.

[Sulliv91] Mark Sullivan and Ram Chillarege. “Software Defects and Their Impact on System Availability - A Study of Field Failures in Operating Systems.” *Digest 21st International Symposium on Fault Tolerant Computing*, June 1991.

[Sulli93a] Mark Sullivan. “System Support for Software Fault Tolerance in Highly Available Database Management Systems.” Ph.D. Thesis, Computer Science Division, University of California at Berkeley. Available as Electronics Research Laboratory, College of Engineering, U. C. Berkeley Memorandum No. UCB/ERL M93/5, January, 1993

[Sulli93b] Mark Sullivan. Personal Communication, July 21, 1993.

[Tandem90] Tandem. Sales brochures and price lists, October, 1990.

[Thomps87] J. G. Thompson. “Efficient Analysis of Caching Systems.” Ph.D. Thesis, Computer Science Division, University of California at Berkeley. Available as EECS technical report number UCB/CSD 87/374, October, 1987.

[Thomps89] James G. Thompson and Alan Jay Smith. “Efficient (Stack) Algorithms for Analysis of Write-Back and Sector Memories.” *ACM Transactions on Computer Systems*, volume 7, number 1, pages 78-116, February 1989.

[Toy92a] L. C. Toy. “Part II: Large-Scale Real-Time Program Retrofit Methodology in AT&T 5ESS Switch.” In *Reliable Computer Systems*, Daniel P. Siewiorek and Robert S. Swarz, pages 574-586, DEC Press, 2nd edition, 1992.

[Toy92b] W. N. Toy. “Part I: Fault-Tolerant Design of AT&T Telephone Switching System Processors.” In *Reliable Computer Systems*, Daniel P. Siewiorek and Robert S. Swarz, pages 533-574, DEC Press, 2nd edition, 1992.

[Walker83] Walker, Popek, English, Kline and Thie. “The LOCUS Distributed Operating System.” *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 49-70, October, 1983.

[Webber92] Steven Webber. “The Stratus Architecture.” In *Reliable Computer Systems*, Daniel P. Siewiorek and Robert S. Swarz, pages 648-670, DEC Press, 2nd edition, 1992.

[Welch86] Brent Welch. "The Sprite Remote Procedure Call System." Technical Report Number UCB/CSD 86/302, Computer Science Division, U. C. Berkeley, June, 1986.

[Welch88] Brent B. Welch and John K. Ousterhout. "Pseudo Devices: User-Level Extensions to the Sprite File System." *Proceedings of the Summer 1988 USENIX Conference*, pages 37-49, June, 1988.

[Welch90] Brent Ballinger Welch. "Naming, State Management, and User-Level Extensions in the Sprite Distributed File System." Ph.D. Thesis, Computer Science Division, University of California at Berkeley. Available as EECS technical report number UCB/CSD 90/567, April, 1990.