The Zebra Striped Network File System

by

John Henry Hartman

Sc. B. (Brown University) 1987
M.S. (University of California at Berkeley) 1990

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

1994

The Zebra Striped Network File System

Abstract

The Zebra Striped Network File System

by

John Henry Hartman

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor John Ousterhout, Chair

This dissertation presents a new network file system, called Zebra, that provides high performance file access and is highly available. Zebra stripes file data across its servers, so that multiple servers may participate in a file access and the file access bandwidth therefore scales with the number of servers. Zebra is also highly available because it stores parity information in the style of a RAID [Patterson88] disk array; this increases storage costs slightly but allows the system to continue operation even while a single storage server is unavailable.

Zebra is different from other striped network file systems in the way in which it stripes data. Instead of striping individual files (*file-based striping*), Zebra forms the data written by each client into an append-only log, which is then striped across the servers. In addition, the parity of each log is computed and stored as the log is striped. I call this form of striping *log-based striping*, and its operation is similar to that of a log-structured file system (LFS) [Rosenblum91]. Zebra can be thought of as a log-structured network file system: whereas LFS uses a log abstraction at the interface between a file server and its disks, Zebra uses a log abstraction at the interface between a client and its servers. Striping logs, instead of files, simplifies Zebra's parity mechanism, reduces parity overhead, and allows clients to batch together small writes.

I have built a prototype implementation of Zebra in the Sprite operating system [Ousterhout88]. Measurements of the prototype show that Zebra provides 4-5 times the throughput of the standard Sprite file system or NFS for large files, and a 15-300% improvement for writing small files. The utilizations of the system resources indicate that the prototype can scale to support a maximum aggregate write bandwidth of 20 Mbytes/second, or about ten clients writing at their maximum rate.

Acknowledgements

I am deeply indebted to the many people without whose friendship, guidance, and assistance this dissertation would not have been possible. First and foremost, I would like to thank my advisor, John Ousterhout. One could not hope for a better advisor. Not only is he an outstanding computer scientist, whose work ethic, reasoning ability, and organizational skills are unparalleled, but he is also one of the most conscientious people I've known. As my advisor his door was always open, and no idea was too far-fetched nor any question too ill-formed for it to receive less than his full attention. I am very grateful for having been able to work with John, and I wish him the best in his new career in industry.

I would also like to thank the other members of my dissertation and qualifying committees: Randy Katz, Felipe Cabrera, and Ray Larson. They have been involved with Zebra from start to finish and I am thankful for the help they've given me.

David Patterson is also deserving of special thanks. He has offered me invaluable advice on topics ranging from trends in computing to better management of one's time.

Graduate school has many pitfalls, but Terry Lessard-Smith, Bob Miller, and Kathryn Crabtree were always there to make sure I avoided them. Terry and Bob helped me out on numerous occasions. Kathryn's help was critical in dealing with Berkeley's numerous rules and regulations.

My thanks also to Ken Lutz for his help in all of the projects with which I have been involved. Without his engineering abilities none would have succeeded. He especially has my thanks for keeping the SPUR running long enough for me to complete my masters project.

I am deeply indebted to the other members of the Sprite project: Brent Welch, Fred Douglis, Mike Nelson, Andrew Cherenson, Adam de Boor, Bob Bruce, Mike Kupfer, Mendel Rosenblum, Mary Baker, Ken Shirriff, and Jim Mott-Smith, without whom Sprite would not exist and Zebra would not have been possible. Brent displayed great patience in showing me the ropes when I first joined the project. Zebra grew out of the many discussions I had with Mendel concerning log-structured file systems. Mary’s help was invaluable during the course of Zebra project. She helped me solve many technical problems, and never once complained about my incessant whining.

The members of the RAID project also provided much advice during my work on my thesis. Ann Chervenak, Ethan Miller, Peter Chen, Ed Lee, and Srinivasan Seshan were all wonderful to work with, and great sources of ideas and inspiration.

I must also thank all of my fellow students who have made graduate school such a pleasure. First, I have fond memories of my years spent living at the Hillegass House for Wayward Computer Scientists. My fellow wayward computer scientists, Ramon Caceres, Steve Lucco, Ken Shirriff, Will Evans, and Mike Hohmeyer, not only kept me up-to-date on current events and the latest research during our memorable dinner-time seminars, but they have also been great friends. My thanks also to Ann Chervenak, Ethan Miller, Marti Hearst, Seth Teller, Paul Heckbert, Kim Keeton, Bob Boothe, and Mark Sullivan for their friendship and support.

My debt to my parents and family is immeasurable. My parents have always offered me their unconditional support in all of my endeavors. They instilled in me the value of hard work and a job well done, and gave me the confidence in my own abilities that made this dissertation possible.

Finally, my wife Randi is deserving of much of the credit for this dissertation. Her love and support have been unfailing. She has been especially understanding of the evenings and weekends spent in the lab, and has always been quick to offer advice and cheer me on. I hope I can be as supportive of her as she has been of me.

This research was funded by NSF grant CCR-89-00029, NASA/ARPA grant NAG2-591, NSF grant MIP-87-15235, ARPA contract N00600-93-C-2481, and the California MICRO Program.

Table of Contents

List of Figures

List of Tables

1 Introduction

The evolution of computers has been a process of decentralization. The early years of computing were ruled by time-shared behemoths that distanced themselves from their users by machine-room doors and legions of system administrators. From their vantage point in distant terminal rooms users got only a glimpse of the computer's power, diluted by slow serial lines, dumb terminals, and the necessity of sharing the computer with other users. The dilution tended to increase over time, too, as more users were added to the system and each user got a smaller and smaller share of the resources. This trend could be offset by increasing the capacity of the mainframe, but eventually it would reach its maximal configuration. There were two alternatives available at this point: buy an additional computer, or buy a computer with more of the resources in demand, such as a faster processor, more memory, etc. The former required splitting the users between multiple computers, offsetting some the benefits of sharing a machine, whereas the latter was expensive. One of the fundamental problems with a centralized resource such as a time-shared mainframe is that it does not gracefully scale to accommodate larger workloads.

Over the years computers shrank in size as advances in electronics packed more and more transistors onto a chip. Eventually the day came when they emerged from the machine rooms in the form of personal computers and workstations and took up residence on users' desktops. Networks were developed to allow computers to communicate and thus allow users to share resources such as printers and files. This new organization had a number of advantages over centralized, time-shared computers. First, moving the computing power closer to the user ushered in interactive computing in the form of bit-mapped displays and corresponding increases in productivity. Second, the computing resources scaled with the number of users. Each user had his or her own computer and was relatively unaffected by increases in users and computers. Networks of workstations and personal computers proved to be more scalable and flexible than mainframes, leading to the slow demise of mainframes.

The migration out of the machine room was not universal, however. Left behind were the file servers. These machines stored the users' files and made those files accessible to other computers (*clients*) via the network. Network file systems, such as NFS

[Sandberg85], were developed to define the interface between the clients and the servers and to allow a single server to handle many clients.

File servers were originally ordinary workstations, outfitted with more disks on which to store their files. Unfortunately these servers were not up to the demands placed upon them. The performance of the file server is critical to the overall performance of the file system, since the speed at which a client can access a file is limited by the performance of the file server. Like their time-shared ancestors, file servers do not scale well to higher performance. Once a file server saturates the addition of more or faster clients will result in lower performance for each client. If more performance is needed then a faster server must be purchased. This lack of scalability has led to larger and larger file servers, such as the Auspex [Nelson93]. These special-purpose machines are tailored to provide file service to many more clients than a mere workstation-based file server can support.

In the future, new styles of computing such as multi-media and parallel computation are likely to demand much greater throughput than today's applications, making the limitations of a single server even more severe. For example, a single video playback can consume a substantial fraction of a file server's bandwidth even when the video is compressed. A cluster of workstations can easily exceed the bandwidth of a file server if they all run video applications simultaneously, and the problems will become much worse when video resolution increases with the arrival of HDTV. Another example is parallel applications. Several research groups are exploring the possibility of using collections of workstations connected by high-speed low-latency networks to run massively parallel applications. These "distributed supercomputers" are likely to present I/O loads equivalent to traditional supercomputers, which cannot be handled by today's network file servers.

The focus of this dissertation is on developing a network file system that scales gracefully to provide higher performance, and is highly available and reliable as well. Its servers can be commodity workstations, which have a better price-performance than the file servers of today. The general idea is to distribute, or *stripe,* file data across multiple servers. This allows more than one server to participate in a file access, thereby decoupling the server performance from the file access speed. Additional performance can be had by adding more servers to the system, so that the performance of the system scales with the number of servers. Availability and reliability are a concern because file systems that stripe across multiple servers are more vulnerable to server failures than centralized file systems. There are more servers to fail and therefore a higher probability that at any given time a server will be down and file data will be unavailable. Failures can be masked either by replicating the file data or by using parity in the style of RAID [Patterson88] disk arrays. In the latter solution one of the servers in the systems stores the parity of the data on other servers; if one of the servers crashes the data it stores can be computed from the data on the other servers.

1.1 Zebra

This dissertation introduces a new striped network file system named Zebra. Zebra is designed to provide a file transfer rate that scales with the number of servers. It also uses its servers efficiently, balances server loads, and provides highly reliable and available service without excessive overheads. Zebra increases throughput by striping file data across multiple servers, and it increases availability and reliability by using parity to mask single server failures. Zebra is different from other striped network file systems in the way in which it stripes data. Instead of striping individual files, Zebra forms all the new data from each client into a stream, which is then striped across the servers. This allows the data from many small writes to be batched together and stored on a server in a single transfer, reducing the per-file overhead and improving server efficiency. The net result is a file system that provides high performance for writes of small files as well as for reads and writes of large files. Zebra's style of striping also makes it easier to use parity to mask server failures. Parity is computed for the stream of newly written data, rather than individual files, and therefore has a fixed cost and simplified management.

1.2 Applicability

Zebra makes several assumptions concerning its computing environment and the types of failures that it will withstand. Zebra is designed to support UNIX workloads as found in office/engineering environments. These workloads are characterized by short file lifetimes, sequential file accesses, infrequent write-sharing of a file by different clients, and many small files [Baker91]. This environment is also notable because of the behavior it does not exhibit, namely random accesses to existing files. Zebra is therefore designed to handle sequential file accesses well, perhaps at the expense of random file accesses. In particular, this means that Zebra may not be suitable for running database applications, which tend to randomly update and read large files. This is not to say that the Zebra design precludes good performance on such a workload, but that the current design has not been tuned to improve random access performance.

Zebra is also targeted at high-speed local-area networks. Zebra is not designed specifically to reduce network traffic. I assume that in a data transfer between a client and server the point-to-point bandwidth of the network is not a bottleneck. Zebra is also not designed to handle network partitions. New point-to-point network architectures, such as ATM, typically include redundant links that reduce the probability of a network partition, and make partitions less of a concern to the design of a network file system for use on a local-area network.

Zebra also assumes that clients and servers will have large main-memory caches to store file data. These caches serve two purposes: to allow frequently used data to be buffered and accessed in memory, without requiring an access to the server or the disk; and to buffer newly written file data prior to writing it to the server or the disk. The former filters out accesses to data that are frequently read, whereas the latter filters out short-lived data.

Zebra is designed to provide file service despite the loss of any single machine in the system. Multiple server failures are not handled; the loss of a second server causes the system to cease functioning, and data may be lost if disks fail catastrophically on two servers at the same time. Any number of clients may fail, however, without affecting the availability of file data. A client crash may lose newly written data cached on that client, but it cannot lose data older than a time limit nor can it lose data written by another client. This is analogous to losing the data stored in a UNIX file system cache when the machine crashes.

1.3 Prototype

I have implemented a Zebra prototype in the Sprite operating system [Ousterhout88], and I have run a variety of benchmarks that demonstrate the advantages of Zebra over existing network file systems. Although Sprite was used as a vehicle for developing the Zebra prototype, the Zebra architecture is not dependent on Sprite in any way. The advantages of Zebra apply to network file systems in general.

The Zebra prototype is implemented on a cluster of DECstation-5000 Model 200 workstations, connected by an FDDI network. The workstations run a modified version of the Sprite operating system. For reads and writes of large files the prototype achieves up to 2.5 Mbytes/second with 5 servers, which is 4-5 times the performance of either NFS or Sprite with LFS. For small files Zebra improves performance by more than a factor of 3 over NFS. The improvement over Sprite is only 15%, however. This is because both Zebra and Sprite require the client to notify the file server of both file opens and closes, and when writing small files these notifications dominate the running time. With the addition of file name caching to both systems Zebra should have even more of an advantage over Sprite.

1.4 Thesis Contributions

This thesis makes several contributions to the state of the art in network file systems:

- Striping can be used to provide scalable performance in a network file system. In the prototype the total read bandwidth to three clients increased from 1.6 Mbytes/second with one data server to 5.8 Mbytes/second with four data servers. Total write bandwidth correspondingly increased from 1 Mbytes/second to 3.2 Mbytes/second. The projected maximum performance of the prototype is about 20 Mbytes/second, due to bottlenecks in keeping track of file blocks and free space.
- High availability can be achieved without sacrificing the gains provided by striping. Zebra uses a novel form of striping called *log-based striping* that allows parity to be used to provide high availability without high overhead. Measurements from the prototype show that parity has no affect on single-client write bandwidth with one data server, and reduces it by only 20% with four.

- Log-based striping allows Zebra clients to batch small writes to the servers, improving the server efficiency and the overall system performance. With the addition of name-caching, Zebra is estimated to be nearly 60% faster than Sprite when writing small files.

Zebra also demonstrates that the benefits of striping and parity can be had in a network file system without requiring a hopelessly complex architecture. There are two features of the Zebra architecture that simplify the striping and parity implementations:

- One of the biggest advantages of the Zebra design is that the same logs that are used by Zebra clients to store file data are used as reliable communication channels over which changes in the distributed state of the file system are transmitted. Thus a single mechanism is used to communicate changes between the components, and since the logs are reliable, recovery from failures is made simple by re-reading the logs.
- Zebra interposes a log abstraction between the files and the disks, so that files are stored in logs, which in turn are stored on the disks. This extra level of indirection makes it easier to distribute the management of storage space and file metadata among the various system components.

1.5 Dissertation Outline

The rest of this dissertation is organized in the following manner. The next chapter gives more motivation and background. Topics covered are file systems, disk storage systems, network file systems, and striped file systems. Chapter 3 discusses the general issues in adding striping and parity to a network file system. Unlike a disk array, a network file system does not have a central host for striping data across the storage devices and computing parity, requiring the functions handled by the RAID host to be distributed.

Chapter 4 presents the Zebra architecture. The four components of the system are introduced (*client*, *storage server*, *file manager* and *stripe cleaner*), and their interaction during normal system processing is described. Activities covered include reading a file and writing a file. Also described is how each component recovers from its own failure, and how the rest of the system tolerates the failure of a component. The chapter concludes with a description of how the configuration of the system is changed by adding and removing clients, servers, and disks.

Chapter 5 describes the implementation of the Zebra prototype. This chapter covers some of the practical aspects of building a Zebra system. Interactions with the underlying operating system are described, as are implementation details that are crucial to the performance of the system.

The performance of the prototype and other network file systems is measured and analyzed in Chapter 6. First, the chapter presents performance measurements of the underlying communication and storage systems used in the prototype, since they limit the prototype's overall performance. These measurements are followed by a collection of benchmarks that measure the file access performance of the prototype, and how it scales with file size, the number of clients, and the number of servers. These benchmarks were also run on standard Sprite and NFS file systems, for comparison. These benchmarks show that Zebra provides 4-5 times the throughput of the standard Sprite file system or NFS for large files and a 15-300% improvement for writing small files. Following the file access benchmarks is a collection of benchmarks that measure the scalability of the prototype, to determine how many clients and servers can be supported before a resource saturates. The results of these benchmarks show that the current prototype can support up to 10 clients writing at their maximum bandwidth of about 2 Mbytes/second before the stripe cleaner saturates. Finally, the chapter concludes with a set of benchmarks that measure the overhead of Zebra’s availability mechanism, showing that while the overheads in the prototype are reasonable for small-scale systems, they need to be optimized to support larger systems.

The concluding chapter summarizes the advantages of Zebra and of logging in general. Fruitful areas for future research are identified.

2 Background

This chapter provides background and motivation for Zebra. The first section gives an overview of file systems in general and defines some terms used throughout this dissertation. The second section describes recent techniques for improving the performance and reliability of disk storage systems. The third section covers network file systems, followed by two sections on the performance and availability problems in network file systems, respectively. The last section covers file systems that distribute files across multiple servers.

2.1 File Systems

A file system provides an abstraction called a *file* that allows application programs to store data on storage devices such as magnetic disks. The abstraction of a file differs from file system to file system, but in UNIX and Zebra, a file is merely a sequence of bytes. The contents of the file are not interpreted by the file system. Applications may read and write arbitrary numbers of bytes at any offset within the file. Bytes written to the end of the file automatically extend the file. Bytes written beyond the end of the file leave holes that read as zeros. Reads beyond the end of the file return an error. Bytes may be overwritten, but it is not possible to insert new bytes between existing bytes.

Applications refer to files in a different manner from the file system itself. Applications refer to files by their names, which are textual strings. The file system refers to files by their unique identifiers. A file may have many names, but only a single identifier. The mapping from file names to file identifiers is called the file system's *name space*. Application programs provide the file's name to the file system when they wish to open a file; the file system uses the name and the name space to find the file's identifier. The identifier is used to create a *handle* for the open session which is returned to the application program. The handle is used in subsequent reads and writes of the file, and eventually to close it.

An application may see a file as an unstructured sequence of bytes, but the underlying file system implements a file as a sequence of fixed-size *file blocks*. The file system maintains the abstraction of a file by mapping each logical file block to a physical disk block. This mapping information is stored in a *block map* for each file. Thus when an application program wishes to read a range of bytes from a file the file system identifies the logical blocks that hold the bytes, uses the file's block map to determine the correct disk blocks to read, and returns the relevant bytes from each block.

The file system also keeps track of various *attributes* for each file, such as its size, date of last modification, and date of last access. The attributes and the block maps are part of what is called the *metadata* of the file system. Metadata is information stored in the file system that is not file data. In UNIX a file's attributes and block map are stored in a fixed-size data structure called an *inode*. Each inode in the system is numbered; this number serves as the unique identifier for the file. For large files whose block map won't completely fit in an inode a multi-level block map is used, in which portions of the block map are stored in disk blocks called *indirect blocks*, pointed to by the inode or other indirect blocks. Figure 2-1 illustrates an application's view of a file, the file system's view of a file, and how the block map implements the logical-to-physical block mapping.

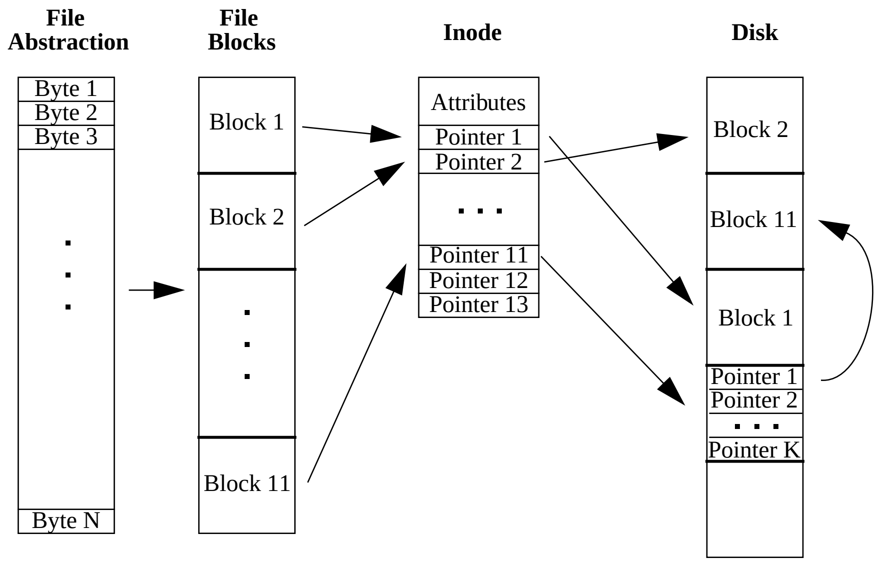

Figure 2-1. File implementation.

Applications see a file as an unstructured sequence of bytes; the file system breaks a file into blocks. The inode maps these logical file blocks into disk blocks containing the file data. The first ten pointers of the inode point directly at the data blocks. The eleventh pointer is indirect: it points at a disk block full of pointers to data blocks. The last two pointers in the inode are doubly and triply indirect, respectively.

The information about the file system name space is also considered part of its metadata. In UNIX the name space is a hierarchy of directories with files at the leaves. Each level of the hierarchy is delineated by a '/' character. Thus the file name `/foo/bar` refers to the file `foo` within the directory `bar`, which in turn is in the root directory /. Directories are nothing more than files whose data consist of the names and inode numbers of other files. When an application refers to a file by name the file system uses the name to traverse the directory tree; at each level it searches the directory for an entry that matches the next component of the path name and uses the corresponding inode number to open the directory on the next level. This mapping of file name to file identifier is called *name lookup* or *name resolution*.

2.1.1 File Caches

The primary limitation to file system performance is the performance of the underlying disk subsystem. Magnetic disks are mechanical devices and are inherently slower to access than memory. It takes tens of milliseconds to transfer a file block from a disk, as compared to hundreds of microseconds from memory. This access gap makes it worthwhile to *cache*, or store, file blocks in main memory to avoid disk accesses.

There are three primary purposes of a file cache: to eliminate disk accesses, to hide the latency of those accesses that cannot be avoided, and to improve the efficiency of the disk through scheduling. Disk accesses are eliminated by retaining recently used file blocks in the cache. Subsequent reads or writes to those file blocks can be satisfied by the cache, and don't have to access the disk. The intent is to exploit the locality of file accesses so that most of the accesses hit in the cache, allowing the file system to operate at the speed of the memory system rather than the speed of the disk.

A file cache can also eliminate disk accesses by filtering out short-lived file data. It is possible that newly-written data will be deleted or overwritten while they are sitting in the cache, and before they are written to the disk. If this is the case, the data do not need to be written to the disk at all. A recent study of file cache behavior found that up to 63% of the bytes written die within 30 seconds, leading to a significant reduction in disk traffic [Baker91][Hartman93]. One catch, however, is that data may need to be written to the disk for reliability reasons, so that a cache cannot eliminate all disk writes, even if it is of infinite size. These cache reliability issues are described in greater detail in the next section.

The second way in which a file cache can improve file system performance is as a buffer to hide the latency of those disk accesses that are not satisfied in the cache. The idea is to overlap disk accesses with application processing, so that the application does not have to stop and wait for the disk. For reads the mechanism that hides disk latency is called *read-ahead* (or *prefetching*), and for writes it is called *write-behind*. Read-ahead is used to bring file blocks into the cache prior to their use by an application program. If the application does indeed read from those blocks then it can do so without having to wait for the disk. Read-ahead requires predicting what blocks will be accessed in the future; fortunately, most UNIX applications read files sequentially and in their entirety [Baker91], so read-ahead can be easily done. Write-behind allows application programs to write a block into the cache without waiting for it to be written to disk. The application can continue processing while the disk access occurs.

The third way in which file caches are beneficial to file system performance is in allowing disk accesses to be scheduled intelligently to improve disk efficiency. Disk accesses do not take a fixed amount of time: accesses that are close together can be completed faster than those that are far apart. Read-ahead and write-behind allow the blocks to be transferred between the cache and the disk in a different order than they are transferred between the cache and the application. By collecting together blocks to be transferred to and from the disk the cache can schedule the transfers to minimize the total transfer time. Without a cache the blocks would have to be transferred in the order in which the application accesses them, which would not necessarily minimize the total transfer time.

2.1.2 File System Reliability and Crash Recovery

Ideally a file system should provide reliable file service, meaning that it should have a vanishingly small probability of losing any of the data written to it. Most file systems do not provide this level of reliability because it is too expensive, both in terms of resources required and performance degradation. Instead, some compromises are made in the reliability of the system. First, most file systems will lose data if a disk fails. Disk failures are typically handled by backing up the contents of the file system to an archive device such as a tape, but data will still be lost if the disk fails before the backup is made. The second threat to reliability comes from caching dirty file blocks in memory to improve performance. Unfortunately, these blocks will be lost in a machine crash. Several techniques are employed to minimize exposure to data lost in this manner (as described below), but they cannot eliminate the problem entirely. Thus the guarantee that most file systems make is that, barring a disk failure, once file data has made it to disk it will remain available until it is overwritten or deleted.

File caches use several techniques for minimizing the amount of dirty cache blocks lost in a machine crash. Some caches write dirty blocks through to the disk immediately. This type of cache is called a *write-through* cache. A write-through cache ensures that a crash won’t cause the file system to lose dirty blocks, but it does not improve the performance of file writes. Each time an application writes a file block it must wait for the block to be written to the disk before proceeding. A write-through cache is still beneficial to the system, however, because it improves the performance of file reads. Other types of caches delay the writing of a newly modified block to the disk. The application’s write completes as soon as the block is in the cache, but the block is not written to disk until later. One variation on this theme writes all the dirty blocks of a file through to the disk when the file is closed. This is called *write-through-on-close*. This improves the performance of applications that write the same file blocks many times in a single open session, but unfortunately this is an uncommon activity [Baker91]. Another scheme writes the data to disk only when the application program makes a special “fsync” request, or when the new data has reached an age limit (typically 30 seconds). This type of cache is called a *write-back* cache. A write-back cache holds dirty blocks in the cache as long as possible in the hope that they will be deleted.

One of the effects of needing to eventually write dirty blocks to the disk is that file caches are more effective at filtering out read requests than write requests [Baker91]. A larger cache can be expected to satisfy a larger number of read requests since there are more blocks in the cache. The same isn’t true for writes because a dirty block that lives longer than the cache's write-back time interval will have to be written to disk, independent of the size of the cache. The study of cache behavior by Baker et al. [Baker91] found that almost all disk writes were due to the 30-second writeback, and were therefore independent of the cache size. The result is that the read/write ratio seen by the disk has been skewed by the cache towards writes. This effect should grow more pronounced as memory sizes grow and caches get larger.

In addition to reliability problems caused by losing the contents of the cache during a crash, file systems may also lose data due to inconsistencies between the file system metadata and the data it stores. For example, consider a write that appends a block to the end of a file. For the write to complete successfully both the inode and the file block must be written to the disk. If a crash occurs after the file block has been written but before the inode is written, the file block will not be referenced by the file's block map and will be inaccessible. Conversely, if the inode is written but not the file block then the file will contain the old contents of the disk block. In both cases the data written by the application is lost due to an inconsistency in the metadata.

Most file systems deal with possible metadata errors by checking the consistency of the file system after a reboot. UNIX file systems use a program called `fsck` to do this checking. `Fsck` examines all of the files in the file system to find and repair such problems as inaccessible blocks. The repairs do not guarantee that the operation in progress at the time of the crash will be completed properly, but they do guarantee that the file system metadata is consistent.

The biggest problem with performing a file system consistency check on reboot is that it is slow. If the file system has no idea what operation was in progress when the machine crashed then it has no choice but to examine the entire file system. This may take many minutes to complete. Such is the case with UNIX file systems, leading to lengthy reboots.

The lengthy reboot time has led to the development of file systems that use a *log* to keep track of file system modifications. The log is an area of the disk that contains records that describe modifications to the file system. Records are added to the log in an append-only fashion. Prior to updating the metadata the file system writes a record to the end of the log that describes the change to be made. During reboot a recovery program is run that uses the log to verify that the operation in progress at the time of the crash completed successfully; any modifications described by the log that are not reflected in the metadata are applied during recovery. The advantage of using a log is that the recovery program need only check the metadata referred to by the records in the log, greatly reducing the recovery time. Examples of file systems that use logging in this way are Alpine [Brown85] and Cedar [Hagmann87].

2.2 Disk Storage Systems

Despite the benefits of caching file data in main memory, doing so cannot eliminate disk accesses completely. Disk accesses will occur to bring new data into the cache, and to write newly created data out of the cache to ensure that they are not lost in a crash. This causes the performance of the underlying disk storage system to have a significant effect on the overall performance of the file system.

There are two ways to improve the performance of the disk subsystem. The first is to improve the performance of the disks themselves, and the second is to improve the way in which disks are used, so that the existing disks are used more efficiently. Prior to describing these techniques, however, it is necessary to describe the operation of a magnetic disk in more detail.

Figure 2-2 illustrates the components of a disk. A disk contains one or more *platters*, which are surfaced with a magnetic media and fixed to a rotating *spindle*. The *arm* contains magnetic heads, one for each surface of the platters, that read and write the media. The surface of each platter is divided up into concentric circles called *tracks*. The arm moves radially across the platters to move the heads from track to track. These arm movements are referred to as *seeks*. All of the tracks that are accessible at a given arm position belong to the same *cylinder*. Each track is divided into smaller regions called *sectors*. A sector is the smallest unit of data that can accessed on the disk. Sectors are typically 512 bytes in size, although this can be configured on some disks.

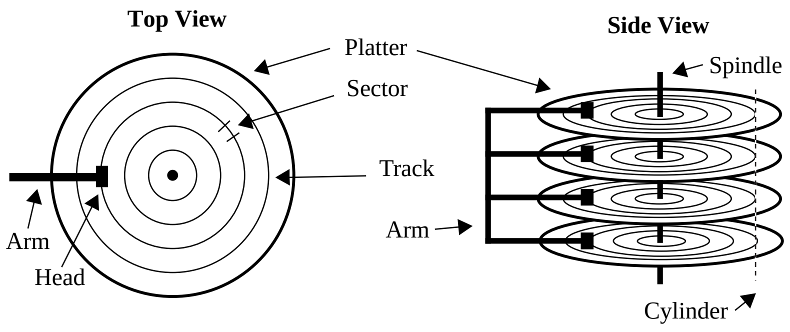

Figure 2-2. Disk components.

The platters are coated with magnetic media and are attached to the spindle, which spins. The read/write heads are attached to the arm, which moves in and out radially. Each radial arm position defines a track. The unit of reading and writing is called a sector. All of the tracks accessible at the same arm position define a cylinder.

The time to access a sector is comprised of three components: seek time, rotational latency, and transfer time. Seek time is the time it takes to move the arm to the cylinder that contains the desired sector. It is directly related to the distance between the current cylinder and the target cylinder: the larger the distance the greater the time. A one-cylinder seek typically takes less than 5 ms, whereas a seek across the entire disk may take upwards of 30 ms. Average seek times are in the range of 10 to 20 ms.

Rotational latency refers to the time it takes for the desired sector to rotate to the head, once the head is over the correct track. To access a random sector the rotational latency will average half of the time it takes the platters to make a complete rotation. Typical disk rotational speeds are 3600-7200 RPM, resulting in rotational latencies of 4.2 to 8.4 ms.

The time spent actually reading or writing data is called the transfer time. The transfer time is based on the rate at which the bits pass under the head, which in turn is a function of the bit density of the media and the rotational speed of the disk. Typical transfer rates are 2 to 4 Mbytes/second, or 120 to 250 microseconds per sector.

2.2.1 Disk Performance Improvements

The mechanical nature of disks makes it difficult to improve their performance. Faster seeks require more power or lighter arms (or both), as well as improved positioning electronics. Faster seeks can also be achieved by shrinking the size of the platter, but this either reduces the capacity of the disk or requires higher density. Reduced rotational latency requires higher rotational speed, resulting in higher power consumption. Higher transfer rates require either higher rotational speed or higher bit density. Higher bit density is achieved through reducing the flying height of the head or improving the head sensitivity. The net result is that raw disk performance isn’t improving very rapidly, only about 7% per year [Ruemmler93].

Advances have been made, however, in improving the effective disk performance by using caching to take advantage of locality in the workload. Most disks now contain a track buffer, which is used to store the contents of the track currently being accessed. This allows the disk to read the contents of the track prior to their use, improving the disk access latency because those requests that are satisfied by the track buffer do not need to access the disk at all. As long as there is sufficient locality in the disk workload, the track buffer will improve the disk’s performance. One caveat is that most file systems already cache file data in the main memory of the computer, reducing the locality of disk accesses and reducing the effectiveness of a cache on the disk itself.

2.2.1.1 RAID

The difficulties in improving disk performance led to the development of RAID (Redundant Array of Inexpensive Disks) [Patterson88], in which many small disks work together to provide increased performance and data availability. A RAID appears to higher-level software as a single very large and fast disk. Transfers to or from the disk array are divided into blocks called *striping units*. Consecutive striping units are assigned to different disks in the array, as shown in Figure 2-3, and can be transferred in parallel. A group of consecutive striping units that spans the array is called a *stripe*. Large transfers can proceed at the aggregate bandwidth of all the disks in the array, or multiple small transfers can be serviced concurrently by different disks.

Since a RAID has more disks than a traditional disk storage system, disk failures will occur more often. Furthermore, a disk failure anywhere in a RAID can potentially make the entire disk array unusable. To improve data integrity, a RAID reserves one of the striping units within each stripe for parity instead of data (see Figure 2-3): each bit of the parity striping unit contains the exclusive OR of the corresponding bits of the other striping units in the stripe. If a disk fails, each of its striping units can be reconstructed

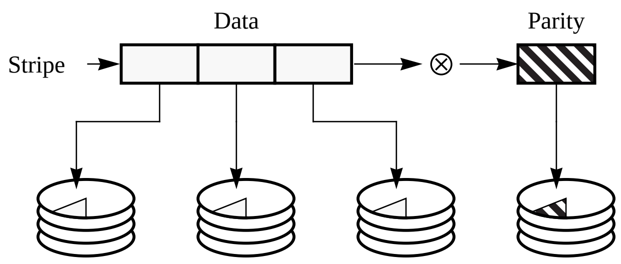

Figure 2-3. Striping with parity.

The storage space of a RAID disk array is divided into stripes, where each stripe contains a striping unit on each disk of the array. All but one of the striping units hold data; the other striping unit holds parity information that can be used to recover after a disk failure.

using the data and parity from the remaining striping units of the stripe, as shown in Figure 2-4. This allows the file system to service accesses to the failed disk by reconstructing the desired data.

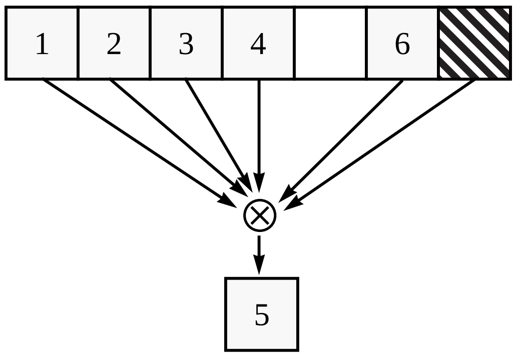

Figure 2-4. Striping unit reconstruction.

A missing striping unit is reconstructed by computing the XOR of all of the other striping units in the same stripe.

A RAID offers large improvements in throughput, data integrity, and availability, but it presents three potential problems. The first is that the parity mechanism makes small writes expensive. If a write operation involves all of the striping units in a stripe (called a *full stripe write*), then it is easy to compute the stripe's new parity and write it along with the data. The additional bandwidth consumed by writing the parity is only 1/N of the array's overall bandwidth, where N is the number of striping units in a stripe. However, writes that don't span an entire stripe (*partial stripe writes*) are much more expensive. In order to keep the stripe's parity consistent with its data, it is necessary to read the current contents of the data block and corresponding parity block, use this information to compute a new parity block, then write the new data and parity blocks. A partial stripe write is illustrated in Figure 2-5. As a result of the need to update the parity, a partial stripe write can require up to four times as many disk accesses on a RAID as it would in a disk array without parity.

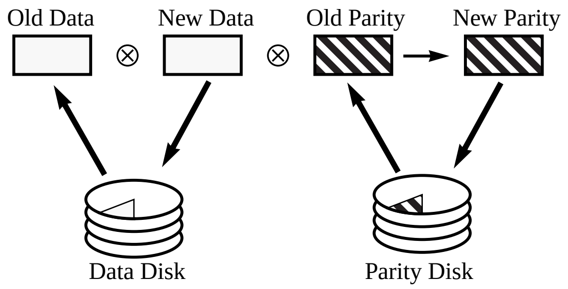

Figure 2-5. Partial stripe write.

A write that does not fill a whole stripe requires a parity update. The old data and parity must be read, the new parity computed, and the new data and parity written. This results in four disk accesses.

Partial stripe writes may be expensive, but they will only have an effect on system performance if they occur frequently. Unfortunately there are a number of factors that conspire to ensure that this will be the case. First, the best size for a striping unit appears to be tens of kilobytes or more [Chen90], which is larger than the average file size in many environments [Baker91], so that even writes of entire files are not likely to fill an entire stripe. Second, when a file is written the file system must update its metadata. If new blocks have been added to the file then new versions of the file’s inode, and perhaps its indirect blocks, will need to be written. These objects are relatively small and are almost guaranteed to be smaller than a stripe. Third, application programs can force a file’s data to disk using the `fsync` system call. If an application chooses to force out the data in small amounts then the RAID will have to deal with partial stripe writes.

The second problem with RAID is that a machine crash during a write may leave the affected stripe’s parity inconsistent with its data. Every write to the RAID involves writing more than one disk since the parity disk must always be updated. If the machine fails during a write it may leave some disks updated, while others are not. In this case the parity will not be consistent with the contents of the stripe, leaving the stripe unprotected in the case of a disk failure. After the machine reboots it must verify the parity of the stripe it was writing at the time of the crash, if any. To do so the system must keep track of the stripe it is writing so that its parity can be verified after a reboot, either by storing the information on disk or in non-volatile memory. If either of these approaches cannot be used then the parity of all of the stripes in the array must be verified after the reboot.

The third problem with the RAID architecture is that all the disks are attached to a single machine, so its memory and I/O system are likely to be a performance bottleneck. For example, a SCSI I/O bus can accommodate up to eight disks, each with a bandwidth of 1-2 Mbytes/second, but the SCSI bus itself has a total bandwidth of only 2-10 Mbytes/second. Additional SCSI busses can be added, but data must also be copied from the SCSI channel into memory and from there to a network interface. On a DECstation 5000/200 workstation, for example, these copies only proceed at about 6-8 Mbytes/second. The Berkeley RAID project has built a special-purpose memory system with a dedicated high-bandwidth path between the network and the disks [Drapeau94] but even this system can support only a few dozen disks at full speed.

2.2.2 Disk Performance Optimization

The second way of improving the performance of a disk subsystem is to optimize the way in which the file system uses its disk. The seek time and rotational latency of an access vary substantially depending on the starting location of the arm and rotational position of the disk. Performance can be improved significantly by avoiding long seeks and rotational latencies. There are two types of optimizations that are used to achieve these goals. The first is to schedule outstanding disk accesses in such a manner as to minimize the time it takes for them to complete. This optimization is referred to as *disk scheduling*. The second is to lay out data on disk so that it can be accessed with a minimal amount of overhead.

2.2.2.1 Disk Scheduling

Disk scheduling is done by ordering pending accesses so that seek times and rotational latencies are minimized. For example, consider a disk whose arm is at cylinder 1, and which needs to access sectors in cylinders 2 and 1000. Accessing cylinder 2 prior to cylinder 1000 results in seek distances of 1 cylinder and 998 cylinders, for a total of 999 cylinders. Accessing them in the reverse order results in seek distances of 999 cylinders and 998 cylinders (1997 total), roughly doubling the amount of seek time required to access the sectors. A recent study [Seltzer90] found that by intelligently scheduling long sequences of random requests the disk bandwidth can be improved from about 7% to 25% of the disk's raw bandwidth.

Disk scheduling works best in environments where there are many pending disk accesses to be scheduled. For example, the best results in the Seltzer study occurred with queue lengths of 1000. Systems with many users and many running applications per disk might generate lots of simultaneous disk accesses that can be effectively scheduled, but it has been shown that in the UNIX environment 70% of the disk accesses encounter an idle disk, and the average queue length is less than ten [Ruemmler93]. Maximum queue lengths of over 1000 were measured on a file server serving 200 users, but queues this long were seen by less than 1% of the accesses. Maximum queue lengths on workstations were less than 100. Thus, for most UNIX disk accesses the disk queue length is too short to take advantage of better scheduling policies.

2.2.2.2 File Allocation

Another technique used to improve disk performance is to lay out data on the disk so that they can be accessed efficiently. For example, in the UNIX office/engineering environment files are usually read sequentially from start to finish. If the file is laid out on the disk contiguously (termed *contiguous allocation*), then the file can be read with a minimal number of seeks: one potentially long seek to the cylinder containing the start of the file, followed by short seeks to adjacent cylinders until the end of the file is reached. Thus contiguous allocation results in the minimum seek cost when accessing the file sequentially.

Contiguous allocation is not without its drawbacks, however. The biggest problem is that contiguous allocation can result in significant amounts of disk space being wasted due to *external fragmentation*. External fragmentation refers to the free space on the disk that is in pieces too small to be used to store files. Consider what happens when a new file is to be stored on the disk. A contiguous region of free space must be found that can store the file. It is unlikely that a region will be found in which the file will fit exactly; probably there will be free space leftover. Furthermore, the UNIX semantics of not specifying the file size when it is created and allowing files to grow by appending makes it difficult to choose a free region of the proper size. Space must be left for potential appends, even though it may be left unused. As more and more files are stored in the system it becomes harder and harder to find space to store them. Eventually the file system may find itself unable to store a file because there isn't a big enough free region, even though there is plenty of free space on the disk.

An example of an existing file system that uses contiguous allocation is Bullet [van Renesse88]. Bullet does not provide UNIX semantics for the files it stores, which makes it easier to implement contiguous allocation. Space is preallocated for file data by specifying the ultimate size of a file when it is created. Files cannot be grown by appending, nor can they be modified. Fragmentation is reduced by reorganizing the disk during off-peak hours, or as necessary. During reorganization files are moved around on the disk to eliminate any space lost due to external fragmentation.

The drawbacks of contiguous allocation have led to the development of extent-based file systems. Examples include DTSS [Koch87] and EFS [McVoy91]. An *extent* is a fixed-sized contiguous region of the disk. The idea is that an extent is large enough so that the cost to seek to its start it negligible when amortized over all of the bytes that are subsequently transferred. This allows extent-based file systems to approach the performance of contiguous allocation. Each file is stored in a small number of extents. If a file grows in size beyond the end of its last extent then another extent is added to it. Extent-based systems avoid external fragmentation because disk space is allocated and deallocated in fixed-sized units, rather than in variable-sized files.

Extents may eliminate external fragmentation, but they introduce the problem of *internal fragmentation*. Internal fragmentation is space within an allocated extent that is not used to store file data. On average one-half of the last extent allocated to a file will be left unused. This space cannot be used by another file because it is smaller than an extent.

The UNIX Fast File System (FFS) [McKusick84] strikes a compromise between contiguous allocation and extent-based allocation by allocating the disk in units of file blocks, but allocating blocks contiguously when possible. When a block is appended to a file its location is chosen based upon the location of the previous block. Ideally the next contiguous block will be used. This allows many of the benefits of contiguous allocation to be achieved without causing external fragmentation. The internal fragmentation present in extent-based systems is avoided by allowing disk blocks to be subdivided into 1-kilobyte size *fragments*. The last block of a file is allowed to occupy a fragment if it is too small to fully occupy a disk block. This reduces the average internal fragmentation per file to one half of a fragment, or 512 bytes.

2.2.2.3 File Clustering

Contiguous allocation and extent-based systems are effective at improving the performance of accessing data within a file, but they do little to improve the performance of workloads that use many small files. In order to speed up these workloads the file system must take advantage of patterns in the accesses between files. This is commonly done by *clustering* files that are used together into the same region of the disk. If the clustering accurately reflects the usage patterns of the files then the overhead of accessing the files will be reduced.

UNIX FFS achieves clustering by dividing the disk into disjoint sets of contiguous cylinders, called *cylinder groups,* and storing all of the files within a directory in the same group. Thus files in the same directory are only short seeks from one another. If applications tend to access files in the same directory within a short period of time then the overheads of doing so are reduced. Different directories are placed in different cylinder groups to spread them out and ensure that the entire disk is used.

2.2.3 Log-Structured File Systems

The allocation and clustering schemes described in the previous sections are intended to reduce both the number of seeks required to access a single file, and the length of the seeks required to access different files. Despite these improvements, it still takes at least one seek to access a file. A seek must be done to the beginning of each file, and in the case of writes, additional seeks may need to be done to write the file’s inode and any indirect blocks. For workloads that contain many small files, as is the case in the UNIX office/engineering environment, this lower limit of one seek per file may be a performance bottleneck. To improve the performance of these workloads it is necessary to design a file system that can access many small files, and their corresponding metadata, in a single transfer to the disk.

The desire to allow many files to be accessed in a single transfer led to the development of the log-structured file system (LFS) [Rosenblum91], which is one of the underlying technologies in Zebra. A log-structured file system treats the disk like an append-only log. When new file data are created or existing files are modified, the new data and their corresponding metadata are batched together and written to the end of the log in large sequential transfers. LFS is particularly effective for writing small files, since it can write many files in a single transfer; in contrast, traditional file systems require at least two independent disk transfers for each file. Rosenblum reported a tenfold speedup over traditional file systems for writing small files.

LFS is also effective at clustering related files together so that they can be accessed efficiently. The append-only nature of the log causes files that are written at about the same time to be clustered on the disk. This style of clustering is markedly different from that used by other file systems such as UNIX FFS. FFS uses *logical clustering,* in which files that are close together in the file system name space (i.e. in the same directory) are stored close together on the disk. LFS, on the other hand, uses *temporal clustering,* in which files that are written at the same time are stored close together. Both styles of clustering assume that the locality of reference when reading the files matches the clustering scheme. In FFS it is assumed that files in the same directory are read together; LFS assumes that files that are written together are read together. If it is the case that files in the same directory tend to be written together then both clustering schemes will achieve the same effect.

LFS obtains its substantial improvement in write performance by transferring large amounts of data to the disk in a single access. These large transfers necessarily require large contiguous regions of free space on the disk. Free space is created as file blocks become unused due to either deletion or modification, but unfortunately there is no guarantee that this free space will naturally coalesce into large contiguous regions. LFS solves this problem through the use of a *segment cleaner,* which is responsible for garbage-collecting the free space and coalescing it into large contiguous regions. The segment cleaner operates in the following way. The log is divided into fixed-size regions called *segments*. Free segments (segments that do not contain any live file data) are used to store new portions of the log. The cleaner generates free segments by copying live data out of existing segments and appending them to the end of the log. Once the live data are copied out of a segment the entire segment is marked as free and can be reused.

In addition to improving disk performance during normal operation, LFS is also able to recover from a crash more quickly than most file systems. Its append-only nature ensures that only the tail of the log can be affected by a crash, therefore only the tail of the log needs to be examined to make sure it is consistent. Most file systems must examine the entire disk because they cannot tell which part of the disk was being written at the time of the crash. LFS uses a checkpoint and roll-forward technique to find the end of a log after a crash and verify the consistency of the metadata. At regular intervals LFS forces all of its metadata to the disk to ensure it is consistent, and stores a pointer to the current end of the log in a checkpoint region in a reserved location on the disk. After a crash the reserved location is used to find the end of the log as of the most recent checkpoint. LFS then runs through the log starting from the checkpoint and brings its metadata up-to-date. By checking only the portion of the log created since the last checkpoint LFS is able to recover from a crash in significantly shorter time than traditional UNIX file systems. Rosenblum reported recovery times on the order of one second for LFS [Rosenblum92], as compared to many minutes for UNIX file systems.

LFS has two features that make it especially well-suited for use on a RAID: large writes, and the ability to find the end of the log after a crash. LFS writes to the disk in large transfers. By making these transfers larger than the RAID stripe size, LFS can avoid the overhead associated with partial stripe writes to a RAID. Writes almost always span all of the disks in the array, so parity can be computed efficiently. Furthermore, after a crash LFS knows where to find the end of the log. By integrating LFS and RAID the overall crash recovery of the system is simplified. During the roll-forward phase of crash recovery the parity of the stripes can be verified, eliminating the need to build this mechanism into the RAID itself.

2.3 Network File Systems

A network file system is one in which the disk that stores a file and the application program accessing it are on separate machines connected by a network. The machine that runs the application programs is called the *client* and the machine with the disks is called the *file server.* The file system defines a protocol whereby the clients can access the files from the file server over the network. In many network file systems clients cache file data in their memories in order to improve performance and reduce the load on the file server. This introduces a consistency problem, however, since a file can be cached by several clients at once; modifications to the file must be propagated to all of the cached copies. One of the biggest differences between network file systems is the way in which they keep the client caches consistent.

2.3.1 Network Disks

The simplest form of network file system is a network disk, in which the network is interposed between the file system and the disk device driver. The file system is run on the client as if the disk were local, except that disk requests are forwarded over the network to the server. The server simply accesses the requested data on the disk and returns them to the client. A network disk is a simple way to implement a network file system, because it only requires the insertion of a network communication layer between the existing file system and disk device driver layers. The file system is unaware that its disk is remote, and similarly the disk device driver is unaware that the file system is remote.

The biggest disadvantage of a network disk is that it is difficult for clients to share files. Each client runs its own file system; if care is not taken they will interfere with each other as they access and modify the disk. For this reason network disks are rarely used. Instead, most network file systems use a higher-level protocol in which the file server manages the disk layout and clients communicate in terms of logical file blocks, rather than disk blocks.

2.3.2 File Block Access

Most file systems provide an interface for clients to access data based on logical file blocks rather than physical disk blocks, since this makes it simpler for clients to share files. Using this interface the clients read and write logical file blocks, and the file server uses the block maps to access the correct disk blocks. Thus the disk layout and metadata management are encapsulated in the file server and hidden from the clients. Since clients do not access the disk directly there is no danger of them modifying the disk in conflicting ways, as is possible with a network disk.

There are some file servers, however, that provide neither a logical file block nor physical disk block interface. First, some network file systems, such as Bullet [van Renesse88], require clients to read and write whole files. The problem with this approach is that clients cannot access files that are larger than they can store. Second, Zebra uses a log abstraction similar to that used in LFS between the clients and servers. Clients read and write portions of a log, rather than file blocks. This arrangement allows clients to read and write multiple file blocks in a single transfer, and is described in more detail in Chapter 3.

2.3.3 File System Namespace

The protocol between the clients and the file server must also include management of the file system namespace. In a local file system the application program gives the desired file name to the file system via a system call. The file system then uses the name to perform the name lookup by accessing in turn each of the directories in the path. In a network file system the application is separated from the disk by a network, and thus there are two places in which the name lookup can logically occur: on the client or on the file server. In the NFS [Sandberg85] network file system the clients do the name lookup by traversing the path themselves. At each level in the path the client sends a *lookup* request to the file server that includes a reference to the current directory (called a *file handle*) and the name of the desired entry in the directory. The file server returns a file handle for the entry. The client repeats the operation until the end of the path is reached. In Sprite [Ousterhout88], the clients send the entire pathname to the file server, which does the name lookup and returns the resulting file handle. In either case the lookup results in a file handle to be used by subsequent read and write operations to the file.

2.3.4 Client Cache Consistency

One way of significantly improving the performance of a network file system is to cache file data on clients. Client file caches provide many of the same benefits as a disk cache: the cache absorbs some of the file traffic, replacing costly server accesses with inexpensive local accesses. For writes it is also desirable to age data in the cache before writing it to the server, since this filters out short-lived data. Client caching introduces a potential cache consistency problem, however, if clients share files. *Read-sharing,* in which several clients have the file cached for reading, does not require any special handling. The clients can read from their individual copies of the files without affecting one another. *Write-sharing*, on the other hand, requires coordination of the client caches. Write-sharing occurs when several clients have a file cached and at least one of them modifies it. If the contents of the caches are not synchronized a client may read obsolete file data from its cache because it didn't realize that another client modified the file (this is called a *stale data error*).

It is useful to distinguish between two different forms of write-sharing: *sequential write-sharing* and *concurrent write-sharing*. In sequential write-sharing only one client has the write-shared file open at a time. Several clients may be reading and writing the same file, but these accesses do not overlap. When a client writes to a file the cached copies on the other clients become obsolete; without synchronization the clients may use these stale copies during subsequent reads.

Concurrent write-sharing occurs when the accesses to the write-shared file overlap, because multiple clients are reading and writing the file simultaneously. Concurrent write-sharing is more problematic than sequential write-sharing because with the latter it is sufficient to verify that the cached copy of a file is current at the time the file is opened; concurrent write-sharing can cause the cached copy of a file to become obsolete while an application is reading from it.

Sequential write-sharing is the most common form of sharing, accounting for at least 80% of all write-sharing [Baker91], and it is also the easiest to handle since clients only need to verify that the cached copy of a file is current when they open it. Concurrent write-sharing, on the other hand, occurs infrequently yet is expensive to handle since clients are simultaneously reading and writing a file. The net result is that a single solution that handles both forms of sharing will be invoked frequently, yet will be expensive. Thus it may be beneficial to handle each form separately.

There are several ways of dealing with write-sharing. The most common approaches are time-based and token-based. A time-based solution allows client caches to become inconsistent, but only for a limited period of time. When a client modifies a file block it sends a copy of the block to the file server within a time limit. Clients that are caching the file block periodically check with the file server to verify that their copy is current. If not, they discard their copies and fetch new ones from the server. The interval between checks can be varied according to how often the file is modified. The advantage of this scheme is that the file server need not keep track of which clients are caching a file. The disadvantage is that a client may occasionally read stale file data from its cache. An example of a network file system that uses a time-based approach is NFS [Sandberg85].

An alternative is to use *tokens* to ensure that clients never cache obsolete versions of files. Each file in use has two types of tokens associated with it: *read* tokens and *write* tokens. Prior to accessing a file a client must hold the correct type of token for the access. Clients obtain tokens from the file server, and the server maintains the consistency of the client caches by coordinating the distribution of tokens and revoking them from clients when necessary. The invariant maintained by the file server is that a client may possess a write token for a file only if no other client possesses a token for the same file. There may be any number of read tokens for a file, but only one write token. If a client wishes to write a file and another client already has a read token then the read token must be revoked by the file server. The mechanism employed to do this revocation is called a *callback*. The server sends an unsolicited message to the client telling it that its token is no longer valid. Revocation of a read token causes the client to discard its cached copy of the file. Revocation of a write token is more complex because the client's copy of the file has been modified and cannot simply be discarded. This is typically handled by having the client write the modified blocks back to the file server when the write token is revoked.

The disadvantage of using tokens to ensure client cache consistency, instead of a time-based approach, is the complexity that it adds to the system. The clients must be capable of receiving unsolicited callbacks, which violates the client/server structure of the system because in a callback the server makes a request of a client rather than the other way around. The file server must keep track of all of the tokens and issue callbacks when appropriate. Furthermore, the state of the tokens must not be lost by a machine crash. After a client crash the server must clean up the state of the client's tokens. After a server crash its token state must be recovered. The details of server crash recovery are covered in the next section.

One variation on the token-based approach is to handle concurrent write-sharing by revoking all tokens, as is done in Sprite [Ousterhout88]. In this scheme concurrent write-sharing causes the server to revoke all tokens for the file, which in turn causes the clients to forward to the server all application read and write requests to the file. Since the server has the only copy of the file the clients are guaranteed to see consistent views of it. The advantage of this scheme is that it simplifies the token implementation since clients only have to obtain tokens when a file is opened, and it has little effect on system performance since concurrent write-sharing is infrequent [Baker91].

2.3.5 Server Crash Recovery

A network file system is a collaboration between clients and servers: each performs some of the functions required for applications to access files. This collaboration leads to dependencies in the states of the machines involved. For example, when a client possesses a token for a file both the client and the server must keep track of this fact; if either one forgets it then inconsistencies may occur, such as clients reading stale file data. Interdependencies in the states of the system's components represent the *distributed state* of the system.

Distributed state is used to improve the performance and correctness of the file system. By knowing the state of the other components in the system each component can optimize and coordinate its activities. For example, distributed state allows a token mechanism to be used to keep client caches consistent, which in turn improves the performance of the system without sacrificing correctness.

The biggest problem with distributed state is that a machine crash causes some of this state to be lost. When the machine reboots it will no longer contain the distributed state that it had before the crash, leaving it inconsistent with the rest of the system. Either the state of the system must be adjusted to account for the lost state, or the lost state must be recovered. The former approach is typically used to deal with client failures. When a client crashes and reboots the server reinitializes the state it has associated with the client, so that after the reboot the client and server states agree.

This approach doesn't work so well for dealing with server crashes since it would mean that all the clients must be rebooted whenever the server crashes. This ensures that the clients' states agree with the server's, but is disruptive for the users. Sprite originally used this technique for handling server crashes, but it quickly became apparent that having to reboot the entire system when the file server crashes is unacceptable, particularly if there are a large number of clients.

There are several ways of handling server crashes that don't require rebooting the clients. The first is to design the file system so that it contains no distributed state. If the states of the clients and servers are not dependent in any way, no changes to the clients' states are required when the server crashes. This type of network file system is referred to as being *stateless,* and is best represented by NFS. Servers do not keep track of things such as which clients are caching which files, allowing servers to crash and reboot without affecting the clients (other than a pause in service while the server is down). The drawbacks of building a stateless file system are that the server cannot ensure the consistency of client caches, nor can it store any information in its main memory that must not be lost in a crash, such as dirty file blocks. These limitations mean that neither the clients nor the servers can use write-back caches, reducing the effectiveness of the caches and decreasing the overall system performance.

Most network file systems use distributed state to improve performance and correctness, but incorporate recovery protocols that allow a server to recover its state after a reboot. One example is Sprite, which uses the clients' states to reconstruct the state of the server. The server learns from the clients which files they have cached, and which files they have open. The server uses this information to initialize its cache consistency mechanism before resuming file service to the clients. Another approach which promises higher performance and security has been proposed by Baker [Baker94]. The server's distributed state is stored in the volatile main memory of the server in such a way that it can be reused after a crash. This allows the server to recover its state without interacting with the clients.

2.3.6 NFS

The de facto standard network file system in the UNIX workstation environment is Sun's Network File System (NFS) [Sandberg85]. NFS was designed to be simple, and as a consequence uses stateless file servers to avoid the complexity associated with maintaining and recovering distributed state. For example, servers do not keep track of the contents of the clients' caches. This allows the system to recover relatively quickly from server crashes. The clients simply wait until the server reboots, at which time their pending requests complete. No recovery protocol is needed because there is no state to be recovered.

NFS pays a price for its simplicity, however. The servers cannot guarantee client cache consistency because they do not keep track of the cache contents. This means that it is possible for a client to access stale file data from its cache (a stale data error) because another client has modified the file without its knowledge. NFS reduces the likelihood of this happening by limiting the amount of time after a file is modified during which other clients may still access the old version of the file. By making this interval small enough the probability of a stale data error occurring can be made acceptably small.

There are two mechanisms used by NFS to limit the interval during which a stale data error can occur. The first is that clients periodically poll the file server to ensure that the files they are caching are not obsolete. For each cached file the client retrieves status information from the server. This status information includes the last modification time for the file, and by comparing this time with the modification time of the cached copy the client can determine if its copy is out of date. If so, the client discards its cached copy of the file and fetches a new copy from the server. When the client accesses a file it checks to see how long it has been since it last retrieved the file’s status. If it has been longer than a time limit then the new status is fetched.

Determining the rate at which a client should poll the server poses a dilemma. A client will read stale file data if a cached file has been modified since it was last polled and the polling interval has not expired. Thus it is desirable to poll frequently. On the other hand, polling consumes server cycles and slows down applications. The more frequently a client polls, the more load on the server. Thus it is desirable to poll infrequently. NFS resolves this conflict by using an adaptive polling interval that varies between 3 seconds and 60 seconds, and is based upon the rate at which a file is modified. Files that are modified frequently are polled frequently, and the opposite is true for infrequently modified files. This allows clients to quickly discover modifications to files that are changing, without unnecessary polling for stable files.

Client polling only solves the cache consistency problem if the server always has the most recent version of a file. If another client has modified the file but not told the server, then polling done by the other clients in the system will not detect the modification. Therefore it is important that clients let the file server know of file modifications in a timely fashion. As a result, NFS uses a write-through-on-close policy, in which clients write back all dirty data associated with a file when they close it. A block write to the server is initiated when a block is modified, but the close does not complete until any pending writes have completed. In this way the server receives the most recent version of a file soon after the client has modified it.

The use of write-through-on-close to send dirty data to the server is not without drawbacks. First, it does not handle the case in which another client reads the file during the time it is open for writing. The file server will not know that the file has been modified and the second client will read stale file data. Second, it reduces the effectiveness of the client cache at filtering short-lived file data because most files are open a very short period of time [Baker91] and it is unlikely that the data will be deleted before it is written to the server. Third, it reduces application performance because the application must wait for dirty file blocks to be written to the server when the file is closed.

Another source of performance problems in NFS is multiplicative writes. When a client writes a file block to the server the server must immediately write the block to disk, to ensure that the block is not lost in a server crash. Unfortunately it is not sufficient to write only the block. If the block is being added to a file then the file’s inode and any affected indirect blocks must be written too. This means that the single block write requires at least two disk writes to complete, further reducing the performance of the server. Modern NFS servers address this problem with non-volatile memory (NVRAM). Modified inodes and indirect blocks are buffered in the NVRAM before being written to disk. This allows multiple modifications to the same file to be filtered by the NVRAM, and allows the inodes and indirect blocks be written to disk in an efficient order [Moran90].

2.3.7 Sprite

The Sprite [Ousterhout88] network file system is designed to provide both high performance and perfect cache consistency. Both of these goals require file servers to be stateful, i.e. to keep track of which clients are caching which files. This allows clients to use write-back caches while avoiding stale data errors. The downside of these goals is that the Sprite servers are more complicated than NFS servers and must go through a complex recovery protocol after a reboot.

Sprite clients use a write-back file cache, rather than the write-through-on-close cache used in NFS. A Sprite client does not write dirty data through to the server until one of four things happens: the cache becomes full, the data reaches 30 seconds of age, an application program forces the data to be written, or the server uses a callback to force the client to write the data. A write-back cache has several benefits. First, short-lived data can be deleted from the cache without being written to the server, thereby improving file system performance and reducing the server's load. Recent measurements of a Sprite system found that between 36% and 63% of newly written data dies before it reaches 30 seconds in age [Baker91][Hartman93]. Second, the write-back cache does not force the application to wait for the data to be written when it closes a file. It can continue processing and the data will be written back later, improving the application performance.

A comparison of Sprite and NFS performance made several years ago found that Sprite was 30-40% faster than NFS [Nelson88]. A more recent study on faster workstations found that Sprite's performance improved to 50-100% faster than NFS [Ousterhout90]. This is because NFS's write-through-on-close policy ties application speed to disk speed, whereas Sprite's write-back policy decouples the two. Processor speed is increasing at a faster rate than disk speed, allowing Sprite to make better use of newer hardware. NVRAM closes the gap somewhat, but still causes high server and network loads.

Sprite's use of a write-back cache comes with a price, however. It makes it more difficult to maintain client cache consistency. For example, if NFS's polling mechanism were used there could be up to a 30 second delay after an application writes a file before it is written to the server, increasing the probability of stale data errors. One of Sprite's design goals was to eliminate stale data errors completely. To achieve this Sprite's file servers are not stateless, unlike NFS's servers. Each Sprite server keeps track of which clients are accessing which files, and uses this information to ensure that write-sharing does not cause a stale data error.

Sprite uses a token-based mechanism for ensuring the consistency of client caches, although tokens are not explicitly passed between the clients and servers. Clients contact the server when they wish to open or close a file; these requests implicitly transfer tokens.

The server keeps track of which files are open and which clients are caching dirty file blocks to ensure that stale data errors do not occur.

Sprite uses different mechanisms for handling sequential write-sharing and concurrent write-sharing. For the former a combination of file version numbers and server callbacks are used to ensure that clients always access the most recent version of a file. Each file has a version number associated with it that is incremented each time the file is modified. When a client opens a file it sends an open request to the file server, and in its reply the server returns the current version number for the file. The client compares this version number with its cached copy of the file (if any), and if the version numbers agree the cached copy can be used. Otherwise the cached copy is invalidated. Thus the file version numbers ensure that clients do not use cached copies of files that are older than those stored on the file server.

File version numbers are not sufficient for preventing stale data errors, however, because there is no guarantee that the version number returned by the server is in fact the most recent version of the file. The write-back caches on the clients make it possible for the most recent copy of a file to be stored in a client cache, so that the server’s copy is out-of-date. Sprite solves this problem through the use of callbacks. When a client closes a file for which it has dirty blocks it notifies the server of this fact. If another client subsequently opens the file the server does a callback to the client with the dirty blocks to force it to write them from its cache back to the server. This ensures that when a client opens a file the server always returns the version number of the most recent copy, guaranteeing that stale data errors do not occur.

The use of callbacks to flush dirty blocks from client caches may handle sequential write-sharing, but it does not solve the concurrent write-sharing problem. For this reason, Sprite uses a separate callback mechanism to handle concurrent write-sharing. These callbacks cause the clients to disable caching of a file. When the server detects that an open will cause concurrent write-sharing it uses callbacks to notify all clients that have the file open that they can no longer cache the file. The clients flush all dirty blocks for the file out of their caches and invalidate their cached copies. All subsequent accesses to the file are sent through to the server, ensuring that the clients always access the most recent contents of the file, even if the file is being concurrently write-shared.

2.3.8 AFS/DEcorum

The Andrew File System (AFS) [Howard88] is a distributed file system intended to support very large numbers of clients. Clients cache file data on their local disks to improve performance and to reduce the load on the servers. Clients also cache file naming information and file attributes, so that files can be accessed without contacting the file server. Both of these techniques help reduce the server load and improve the scalability of the system.

AFS uses a callback mechanism to keep the client caches consistent. When a client caches a file it assumes that the cached copy is up-to-date unless told otherwise by the server. The server keeps track of which files each client caches so that it can notify them when their cached copies become obsolete. The server detects that cached copies have become obsolete when a client writes back dirty blocks for the file. To ensure that the server detects the new version of a file in a timely fashion the client caches are write-through-on-close. When a client closes a file it has modified it writes all of the dirty blocks back to the server so that the server can then notify other clients that have the file cached that their copies are out-of-date and should be invalidated.

One of the problems with AFS's cache consistency mechanism is that it does not handle concurrent write-sharing. The server does not notice that a file has been modified until it is closed, so that it cannot invalidate other cached copies of a file while the file is being modified. During that time the other clients will use out-of-date data from their caches.

Transarc's DEcorum [Kazar90] is a commercial version of AFS that provides better consistency guarantees and higher performance. Perfect client cache consistency is guaranteed through the use of tokens that are explicitly passed between the clients and the server. A client cannot read from a file unless it has a read token, and cannot write to it unless it has a write token. The server ensures that no other client possesses a token for a file if one client has a write token for it. To do so, the server may have to revoke read tokens from clients, which causes each client to flush any dirty blocks for the file from its cache and invalidate its cached copy of the file. By controlling the issuance of tokens in this manner the server is able to ensure that stale data errors do not occur, even if the file is undergoing concurrent write-sharing.

Since DEcorum uses a token mechanism to ensure that client caches are consistent it does not need to use write-through-on-close caches on the clients. Instead DEcorum uses write-back caches in which a client only writes back a files's dirty blocks to the server when the file's token is revoked. This provides higher performance than a write-through-on-close cache because file blocks are only written to the server when necessary to avoid stale data errors, allowing more of the file data to die in the client cache.

2.4 Network File System Performance Limitations

A network file system as described in the last section, and as represented by NFS, Sprite, AFS, and DEcorum, suffers from two major performance limitations: file server bottlenecks and unequal server loads due to non-uniform file accesses. The next sections describe these limitations in more detail and the techniques used by current network file systems to overcome them.

2.4.1 File Server Bottlenecks

In a traditional network file system each file is stored entirely on a single server. Although there may be many file servers in the system, when a client accesses a particular file it interacts only with the single server that stores the file. This means that the speed at which a file can be accessed is limited by the performance characteristics the file server, including its memory bandwidth and the speed of its processor, network interface, I/O busses, and disks. For example, on a DECstation 5000/200 workstation the processor is rated at 20 SPECmarks, memory to memory copies run at 12 Mbytes/second, copies to and from the I/O controllers run at 8 Mbytes/second, and a single disk transfers at 1.6 Mbytes/second. The saturation of any one of these server resources will limit the overall system performance. Thus it is important that the server's capabilities be carefully matched with the clients and workloads it must serve. Increases in bandwidth demands, either through improvements in client performance or changes in workloads, will require a corresponding increase in server performance. This may be as simple as adding disks or I/O busses to the server, but eventually the maximal server configuration will be reached beyond which its performance cannot be improved.

The importance of file server performance and the difficulty of improving it leads to network file servers that are high-performance and expensive machines. File server performance determines system performance, therefore high-performance systems require high-performance servers. Often special-purpose machines are used as file servers in order to get higher performance than could be delivered by a general-purpose computer. These machines are tailored to their file server task by having high-bandwidth data paths connecting the network to the disks and multiple processors for handling client requests. The following sections give some examples of high-performance file servers.

2.4.1.1 Auspex NS 6000

The Auspex NS 6000 [Nelson93] is a special-purpose computer designed to provide high-performance NFS file service. The main focus is on supporting high aggregate loads generated by large numbers of clients and networks. Thus the performance available to any one client may not be especially high, but the Auspex can provide this level of performance to many more clients than a general-purpose computer used as a file server.

The Auspex is a functional multiprocessor: it contains several processors, each of which is responsible for managing a different task related to NFS file service. The *network processor* handles all network communication and implements both the underlying network and NFS protocols. The *file processor* manages the file system metadata as well as a cache of file blocks in a memory buffer. The *storage processor* controls the I/O channels to the disks. These processors, and the cache memory, are connected by a high-speed bus. The division of tasks between separate processors allows overlapping of the tasks so that system performance can be improved in a balanced manner. For example, network bottlenecks can be alleviated by adding networks and network processors without affecting the rest of the server configuration.

2.4.1.2 RAID-II

RAID-II [Drapeau94] is a research project at U.C. Berkeley to build a high-performance network file server from a disk array and a host workstation. In a traditional file server the disk array and the network would be connected to the I/O system of the host workstation, and data would be transferred between the network and the disk array via the workstation's backplane and memory system. RAID-II avoids the potential performance bottleneck presented by the host backplane via a high-performance data path that connects the disk array directly to the network. File data move between the network and the disk array over this path, without entering the host. The host is responsible for setting up the data transfers, but it does not manipulate the data directly. This control function requires relatively low bandwidth, avoiding a bottleneck on the host backplane.

2.4.1.3 DataMesh

The DataMesh [Wilkes89][Wilkes91] is a storage server design consisting of an array of processors and disks connected by a high-performance interconnect, much like a parallel machine with disks attached to each node. Some of the nodes in the array are network nodes that connect the DataMesh to the outside world and its clients. File requests are received by the network nodes and directed through the interconnect to the appropriate disk node. File service performance is improved by having multiple disk nodes participate in a file transfer in parallel. A potential bottleneck, however, is the network node connected to the client. If the network node saturates then it will be necessary to connect the client to multiple network nodes. The DataMesh architecture does not address the issue of coordinating multiple network nodes to participate in handling a single access by a client.

2.4.2 Hotspots

One way of improving the aggregate performance of a network file system is to add file servers. This does not improve the performance of accessing a single file, but it does increase the total storage capacity and bandwidth of the system. It may be difficult to balance the loads among the servers, however, since file accesses are not distributed evenly among the files. Popular files represent *hotspots*, or data in the file system that are heavily used. An example is the directory that contains the system binaries, which is likely to be accessed frequently. There are a couple of current techniques for balancing server loads. The first, and most common, is to manually distribute files among the servers so the loads are balanced. This is undesirable because it requires human intervention and the balance is only temporary since access patterns change and hotspots move about. The second is to replicate the files in the system so that more than one server stores a copy of each file. Accesses to a file can be spread out over the replicas of the file, distributing the load across the servers. The drawbacks of using replication are that it requires additional storage, a mechanism for ensuring the consistency of the replicas, and a policy for deciding when and where to replicate what files so that server loads are balanced.

2.5 Network File System Availability

As described, the file servers in a network file system represent single points of failure. If a server crashes, all of the files that it stores will be inaccessible while it is down. Clients that need to use any of the files on the server must wait until it reboots. Even worse, it is possible for files to be lost if a server suffers a hardware failure. Two existing techniques of avoiding the single points of failure presented by file servers are file replication and dual-ported disks.

2.5.1 File Replication

The most common solution for providing highly available and reliable file service is to replicate each file. Each replica of a file is stored on a different server. When a server fails, the replicated copies of its files can be used to provide uninterrupted service to its clients. Replication is expensive, however, both in the additional storage and server resources needed to store the replicated copies, and in the cost of keeping the replicas consistent.

A major difference between the different network file systems that replicate files is the way in which they maintain the consistency of the replicated copies. If a client modifies a copy of a file then that modification must eventually be propagated to all of the replicas. Immediate propagation ensures that all of the replicas remain consistent, but it is an expensive operation since the application must wait until all of the copies are updated. For this reason most replicated file systems update the copies asynchronously. This is a trade-off of performance for availability, however, because the replicas will be left inconsistent if the server containing the modified copy fails before the replicas are updated. Some systems allow the user to specify the number of replicas and how they are to be updated to adjust the performance and reliability trade-off to the user’s liking.

One of the biggest advantages of a replicated file system is that it allows the system to survive network partitions. If a network failure occurs the system may become partitioned into separate groups of machines, with each member of a partition able to communicate with the other members of the same partition but not with any other partition. In a non-replicated file system only the machines in the same partition as the file server will see uninterrupted file service. Clients in other partitions will have to wait until the network reconnects for file service to be restored. In a replicated file system each partition containing copies of the needed files can continue operation. When the partitions are reconnected the replicas must be brought up-to-date and any conflicting modifications resolved. Many conflicts can be resolved automatically, but some, such as a simultaneous modification of a file and a deletion of the same file, may have to be left to a human to sort out.

Some examples of network file systems or file server architectures that provide replicated copies are Locus [Walker83], Coda [Satyanarayanan90], Ficus [Guy90][Page91], Echo [Hisgen89], Harp [Liskov91], and Deceit [Siegel90]. Locus, Coda, and Echo are complete network file systems in the sense that they define a client/server protocol for accessing files. The rest of the systems do not define a particular protocol; rather, they are layered on top of existing protocols, such as NFS, and focus on improving the reliability and availability of the file server.

Locus allows a file to be replicated among a set of *storage sites* (SS), coordinated by a *current synchronization site* (CSS). A client contacts the CSS on each open and close and the CSS directs the request to a storage site containing the most recent version of the file. When a file is modified on an SS it notifies the other SS's, and they bring their copies up-to-date in the background. The CSS ensures that the clients do not access the obsolete copies during the update. Locus can tolerate network partitions by allowing each partition to have a CSS. This can cause the replicas of a file to receive conflicting modifications during the network partition, since each partition has its own CSS. These conflicts are resolved once the partitions are reconnected. Conflict resolution for common types of files is handled automatically, otherwise the user must resolve the conflict.

Ficus uses a replication scheme similar to Locus. Ficus updates file replicas asynchronously, so that an application that modifies a file can continue processing while the modifications are propagated to the other replicas in the background. This also means that inconsistencies are possible. Periodically a reconciliation algorithm is run to ensure that the replicas are consistent. It is possible for stale data to be accessed during the time before reconciliation occurs, but the designers believe this to be a fair trade-off of performance and scalability vs. consistency. Conflicting modifications discovered during reconciliation are handled automatically when possible, otherwise the conflict is reported to the user.

Coda uses a read-one, write-all replication scheme in which clients can read any replica of a file, but must write them all. This does not guarantee that every replica will be up-to-date, because network partitions can cause some servers to miss an update to a file. To avoid this problem the client contacts all of the servers to determine which has the most recent copy, and reads from that one. In the normal case they all should have the most recent copy, allowing client accesses to be distributed among the servers. Coda also allows clients to cache files and to operate in a disconnected manner. Servers callback clients when cached files become obsolete. If the client is disconnected it will obviously miss these callbacks; in this case the conflict is resolved when the client reconnects to the network.

The rest of the replicated systems use a primary copy scheme, in which one replica of a file is designated as the primary copy and all accesses are directed to it. Backup replicas are only used if the primary fails, in which case a backup is promoted to be the primary. Once the primary becomes available again it is brought up to date with the backup. Echo uses a write-ahead log to keep track of file system modifications. The primary server writes this log to disk when a file is modified. This log is also written to the backups and is used to recover from a failure of the primary server. Harp also uses a write-ahead log, except that an uninterruptible power supply is used to ensure that the log is reliable, rather than writing it to disk. This allows file modifications to complete without an expensive disk access.

Deceit is an NFS-based file server that supports replicated copies. Clients access a Deceit server as if it were a standard NFS file server. Deceit servers are pooled together to provide highly-available service. Files can be replicated and the degree of replication and the update propagation policy specified by the user. If a client contacts a server that does not store a replica of the desired file the request is forwarded to a server that does. Updates to replicas are propagated using the ISIS distributed programming toolkit [Birman84].

2.5.2 Dual-Ported Disks

Another means of providing highly available file service is to connect dual-ported disks to two servers, as shown in Figure 2-6. Each disk can be accessed by either server, preventing a server failure from making a disk inaccessible. During normal operation each disk is accessed by only one of the servers. This server is the primary server, or owner, of the disk. Clients access the files on a disk through its primary server. When the primary server fails the other server takes over ownership of the disks and begins servicing client requests. Disk failures can be tolerated using mirrored disks. When the primary server writes to a disk it also writes to a backup disk. A hardware failure of the primary disk causes the system to switch over to using the backup disk.

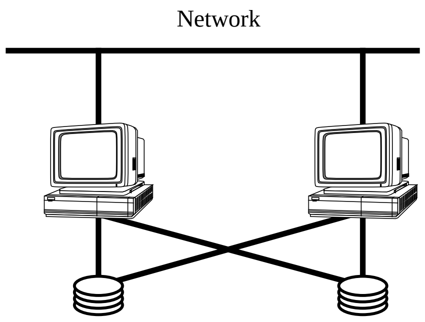

Figure 2-6. Dual-ported disks.

Each disk is connected to both servers, allowing either server to service requests to a disk. During normal operation one of the servers is designated the primary server for each disk; the other server only services requests if the primary fails. Each disk may contain its own file systems, or for added reliability, the disks may be mirrored to guard against disk failures.

There are several advantages of using dual-ported disks rather than replicated files. First, a server failure can be tolerated without having replicated copies of the files it stores. The dual-ported disks allow the backup server to directly access the failed server's disks. Second, disk failures can be tolerated using mirrored disks, rather than a more complicated replication scheme. The primary server simply writes to both disks, avoiding the communication costs and computation on the backup servers needed to update the replicas of a file.

The disadvantages of dual-ported disks are that they do not allow the system to survive network partitions, and the failover from the primary to backup server can be complex. Care must be taken that only one server at a time thinks it is the owner of a disk or chaos will ensue.

Two examples of highly-available file servers that use dual-ported disks are Echo [Hisgen89] and HA-NFS [Bhide91a][Bhide91b]. Echo uses a combination of replication and dual-ported disks to provide highly available file service. HA-NFS uses dual-ported disks to provide highly available NFS file service. They are very similar in their implementation. Both allow disks to be attached to multiple servers, both use special hardware to ensure that there is only one owner of a disk at a time, and both use a write-ahead log on the disk to record file system modifications. Changes to the file system metadata are written to the log before being applied to the metadata. If a failure occurs before the metadata has been updated the log is used by the backup server to apply the necessary changes.

2.6 Striped File Systems

A striped file system *stripes,* or distributes, its file data across multiple storage devices, allowing a single file access to be served by more than one device. This allows the system's file access bandwidth to scale with the number of devices that participate in a transfer. The storage devices can be disks, I/O nodes in a parallel computer, or file servers in a network file system. Striping across disks is usually implemented in the disk storage system, as in RAID. The disk array appears to the file system as a single large and fast disk. Consecutive disk blocks may in fact be stored on separate physical disks, but the file system is not aware of this fact. The advantage of this approach is that the file system does not need to be modified. RAID was described in Section 2.2.1.1. The rest of this section describes striped file systems for parallel computers and networks of workstations.

The terminology I use to describe striped file systems is very similar to that for RAID. A collection of data that spans the servers and over which parity is computed is called a *stripe*, and the portion of a stripe stored on a single server is a *stripe fragment*. Stripe fragments that contain file system data are called *data fragments*, whereas the fragment that contains the parity of the data fragments in a stripe is called a *parity fragment*.

2.6.1 File-Based Striping

One feature shared by all existing striped file systems is that they stripe each file across the storage devices independently, as shown Figure 2-7. I refer to this as *file-based striping*, because each data fragment contains a set of blocks from a single file. In contrast, a RAID implements *disk-based striping*, because the data fragments contain blocks of a virtual disk. Thus, the striping and parity functions in a file-based system do not span files, since each stripe contains the blocks of a single file.

While conceptually simple, file-based striping has two drawbacks. First, small files are difficult to handle efficiently. If a small file is striped across all of the servers as in Figure 2-8(a) then each server will only store a very small piece of the file. This provides little performance benefit, since most of the access cost is due to network and disk latency, yet it incurs overhead on every server for every file access. If the client workload consists of mainly small files then striping files in this manner may actually reduce the overall system performance, rather than improving it. Thus it seems better to handle small files differently than large files and to store each small file on a single server, as in Figure 2-8(b). This leads to problems in parity management, however. If a small file is stored on a single server then its parity will consume as much space as the file itself, resulting in high

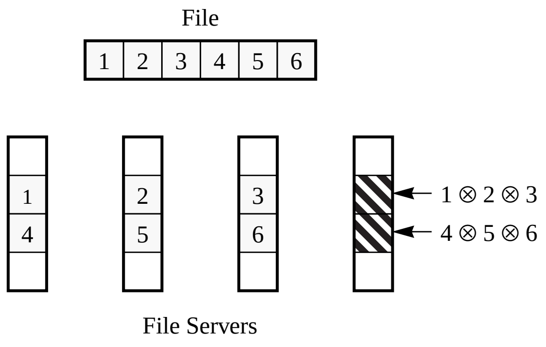

Figure 2-7. File-based striping for a large file.

The file is divided up into stripe units that are distributed among the servers. Each stripe contains one parity fragment.

storage overhead and double the amount of data written by a client when it writes the file. In addition, the approach in Figure 2-8(b) can result in unbalanced disk utilization and server loading. Care must be taken to distribute the small files over the servers so that the loads and utilizations do not vary too much.

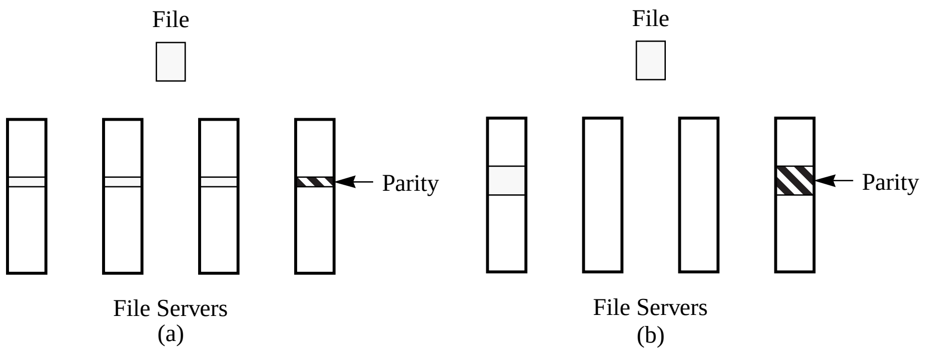

Figure 2-8. File-based striping for a small file.

In (a) the file is striped evenly across the servers, resulting in small fragments on each server. In (b) the entire file is placed on one server but the parity takes as much space as the file.

The second problem with file-based striping is that partial stripe writes are expensive and complicated to implement. If an existing file is modified then its parity must be updated to reflect the modification. As in a RAID, the writes of the file block and the corresponding parity block must be atomic; if only one completes the contents of the stripe, and therefore the file, may be lost. Unlike a RAID, the storage devices involved are not connected to the same computer, requiring an atomic commit protocol between the computers involved. There exist protocols for ensuring that two writes to two different servers are carried out atomically [Bernstein81], but they are complex and expensive.

Despite the disadvantages of file-based striping, its conceptual simplicity and ease of implementation have caused it to be used in all existing striped file systems. Its major drawbacks are related to difficulties in managing parity, making it ideal for those systems that do not maintain parity. The following subsections describe the existing parallel file systems and network file systems that use file-based striping to allow clients to access files in parallel from multiple servers.

2.6.2 Parallel File Systems

A parallel file system is one that stripes file data across I/O nodes in a parallel computer. The purpose of the striping is to improve the bandwidth of file accesses. The striping is usually done by storing consecutive blocks of a file on different I/O nodes. Parallel file systems have a very different set of design criteria than network file systems, because a parallel machine is a much more tightly-coupled environment than a network. Parallel file systems do not typically worry about failures in the interconnect, processors, or storage systems. Examples of parallel file systems are CFS [Pierce89], *sfs* [LoVerso93], and Bridge [Dibble90]. All three support standard UNIX semantics as well as parallel access modes that allow multiple processors to access a file concurrently.

CFS (Concurrent File System) is designed for the Intel iPSC/2 parallel computer. CFS has a single name process that manages the file system name space. Each I/O node runs a disk process that manages the disks attached to the node. The metadata for each file in the system consists of a *file structure* that contains information about the file, such as its size, and a list of pointers to the file's data blocks (or indirect blocks in the case of large files). A library is used to translate I/O operations by application programs into requests to the name process and the appropriate disk processes, as specified by the contents of the file structure.

CFS does not allow the processing nodes to cache file data. All file read and write operations result in an access to at least one disk process. This eliminates the need for a cache consistency protocol, but results in increased latencies, communication costs, and load on the disk nodes. Interestingly, CFS does allow processing nodes to cache copies of the file structure to avoid fetching it on every file access. These copies are kept "adequately up to date" through "simple lazy conventions" [Pierce89].

The *sfs* file system [Lo Verso93] provides parallel file access on the CM-5 parallel computer. The NFS protocol is used to communicate between the processing nodes and a centralized processor that runs the file system. This allows sharing and caching of files, but requires additional latency when accessing a file since processing nodes can't access the I/O nodes directly. They must first contact the file system node which uses the file metadata to send an I/O request to each of the I/O nodes storing the file. Data are then transferred between the I/O nodes and the processing nodes. The file system node is notified when the I/Os are complete so it can update the file metadata.

Bridge [Dibble90] is designed to allow multiple processes in a parallel application to access a file concurrently. A centralized process called the *Bridge server* maintains the file system metadata. One common mode of operation uses the Bridge server to initiate parallel transfers. One of the nodes in a parallel application opens the file, thereby becoming the *job controller.* The job controller provides a list of all of the application's nodes to the Bridge server. Subsequent read and write operations by the job controller cause parallel transfers to occur to and from all of the application's nodes.

Bridge applications can also determine where the blocks of a file are stored to better configure themselves to minimize data transfers. The most obvious use of this knowledge is to run the job that will access a particular file block on the node that stores that block. For example, copying a file can be implemented by doing a block copy locally on each node storing a block. Bridge reports nearly linear speedup in applications that access file blocks in this manner.

2.6.3 Swift

Swift [Cabrera91] improves the performance of network file systems by striping files across its file servers. Clients can access a file in parallel from the servers, improving the performance of file access. Swift also provides highly-available file service, either by replicating the files it stores or by storing parity in much the same manner as a RAID.

Swift consists of four components: *clients*, *storage agents*, *storage mediators*, and *distribution agents*, as shown in Figure 2-9. Clients run application programs that read and write files. The storage agents store file data. The storage mediator is responsible for managing the storage and network resources. When a client wishes to access a file the storage mediator reserves the appropriate storage and network capacity and creates a transfer plan. For example, to write a file the storage mediator would decide the appropriate striping unit, preallocate space on the storage agents, and reserve network capacity for the transfer. The storage mediator is also responsible for ensuring the consistency of client caches by using call-backs to flush client caches when appropriate.

A *distribution agent* is responsible for implementing the transfer plan created by the storage mediator. Its most important function is to stripe file data over the storage agents

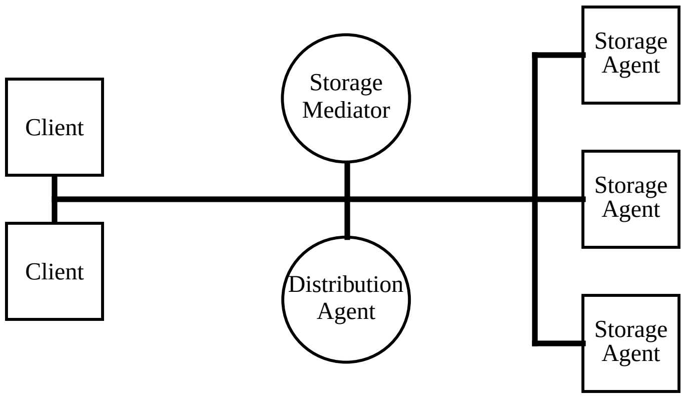

Figure 2-9. Swift architecture.

Clients run applications, storage agents store file data. The storage mediator allocates storage and network resources. The distribution agent carries out the transfer plan created by the storage mediator to transfer data between the clients and the storage agents.

as specified by the transfer plan. During writes the distribution agent breaks the file into blocks which are striped across the storage agents, and during reads the distribution agent reassembles the file from its constituent blocks on the storage agents. Thus the distribution agent is a potential performance bottleneck because all data read and written by the clients must pass through it, but the Swift architects expect to avoid this problem by having many distribution agents in the system, each of which is co-resident with the clients it serves.

The storage mediator is a central point of failure because the metadata it contains is needed to formulate the transfer plans. Swift makes the metadata highly reliable by storing it on the storage agents, which in turn are made highly reliable through replication or parity.

An initial prototype of the Swift architecture was implemented as a proof of concept. It did not support any of the reliability mechanisms. The storage mediator and distribution agent functions were implemented in libraries linked into the application program. Files were striped uniformly across the storage agents. This prototype has shown nearly linear speedup of file reads and writes when the number of storage agents is increased from one to three.

Recently the Swift prototype has been reimplemented to incorporate the reliability mechanisms [Long94]. The prototype can now support a variety of parity organizations. Measurements indicate the parity computation incurs a significant overhead, so that the resulting performance of a five-server system with parity enabled is only 53% of the original Swift prototype with the same number of servers. In other words, parity requires less storage than replication, but it does not provide better performance.

2.7 Summary

There are many issues in designing a high-performance and highly-available network file system. At the lowest level the system's performance is dependent on the performance of the file server's disk subsystem. Techniques such as RAID can be used effectively to improve the disk subsystem performance, by aggregating many disks into a single logical disk. LFS improves the efficiency of the disk subsystem through the use of temporal clustering, which allows small writes to be batched together and written to the disk in large, efficient transfers. File server performance is further improved by caching file data in its main memory, so that some disk accesses are avoided and those that cannot be are scheduled efficiently.

At the next level in the system performance is improved by defining efficient protocols between the clients and servers, and by avoiding file server accesses via client file caches. The data transfer protocol in most network file systems is based on logical file blocks. A client request specifies a logical file block, and the file server uses the file's block map to determine the corresponding disk block. This approach allows many clients to share files and hides the details of the block map from the clients, but it requires the file server to process each individual file block that is accessed, limiting the server performance.

Client file caching improves system performance by eliminating file server accesses. The cache allows some of the reads and writes by application programs to be satisfied locally, without contacting the server. This not only improves the file system performance, but reduces the load on the file server and allows it to service more clients that would be possible without client caching.

At the highest level the design of a network file system must be concerned with availability and scalable performance. The standard technique for providing highly-available file service is to replicate files. This allows the clients to continue accessing files even if a server crashes. The disadvantage of replication is that it requires additional storage space to contain the replicas, and it requires a consistency mechanism for ensuring that an update to a file is propagated to all of its replicas. For these reasons the use of parity to provide highly-available network file service seems appealing. RAID has shown that parity can be used to build highly-available disk subsystems; it seems worthwhile to investigate its use in network file systems. The Swift system has already demonstrated that the use of parity in a network file system does reduce the amount of storage required to provide highly-available service, but Swift's parity implementation does not provide significant performance improvements over replication.

Swift has also demonstrated that striping can be used to provide scalable performance in a network file system. In a traditional network file system the performance obtained when accessing a file is limited by the resources of the file server that stores it. Additional servers only improve the aggregate transfer bandwidth of the system, not the individual transfer bandwidth. By striping file data across servers the file access performance is decoupled from the server performance, allowing both the aggregate and individual transfer bandwidths to be improved by adding servers. More servers means a file can be accessed at a faster rate.

The focus of the Zebra design effort has been to use striping and parity to improve the performance and availability of network file systems. Zebra's advantage over previous efforts in this area is that it borrows ideas from both RAID and LFS. As will be seen, the combination of striping with parity and log-based storage allocation leads to a highly integrated system that has high server efficiency, uses striping to provide scalable performance, and uses parity to provide high availability. The details of Zebra's striping and parity mechanisms are found in the next chapter.

3 Zebra Fundamentals

Distributed systems are at the same time better and worse than their centralized counterparts. A distributed system is better than a centralized system because it offers scalable performance and high availability, but the inherent complexity of distributing a computation makes it more difficult to implement. Much of this complexity comes from the need to synchronize accesses to shared resources, so that each component of the system sees a consistent view of each resource. For example, if the clients of a network file system are sharing a file the accesses to the file must be synchronized so that the clients always read the most recently written data. Thus there is a tension in the design of a distributed system between the desire to distribute functionality to improve scalability and availability, and the need for synchronization to permit sharing and ensure consistency.

There are several manifestations of this tension in the design of a striped network file system. First, the striping and parity functions that in a RAID are handled by the centralized host processor must be distributed. The manner in which this distribution is done can have a significant impact on the resulting system's performance and synchronization complexity, however. All other existing striped network file systems use file-based striping, in which individual files are striped across the servers. As described previously, this form of striping has several drawbacks, including excessive parity overheads for small files and the need for partial stripe writes to be atomic. Thus, while file-based striping promises to work well in workloads dominated by large files and full stripe writes, it may have excessive overheads on workloads that don't have these characteristics.

The inherent limitations of file-based striping led to the development in Zebra of an alternative striping scheme called *log-based striping*. Log-based striping borrows ideas from log-structured file systems (LFS) [Rosenblum91], so that instead of striping individual files, as is done in file-based striping, Zebra interposes a log abstraction between the files and the disks and stripes the logs. Zebra can be thought of as a log-structured network file system: whereas LFS uses the logging approach at the interface between a file server and its disks, Zebra uses the logging approach at the interface between a client and its servers. Each Zebra client organizes its new file data into an append-only log which it then stripes across the servers, as illustrated in Figure 3-1. Thus

Zebra clients stripe logs containing files, rather than individual files. Each client creates its own log and computes the parity of its log as it is written to the servers.

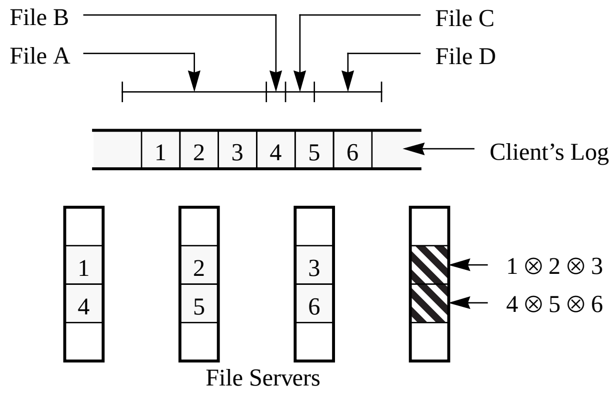

Figure 3-1. Log-based striping.

Each client forms its new file data into a single append-only log and stripes this log across the servers. In this example file A spans several servers while file B is stored entirely on a single server. Parity is computed for the log, not for individual files.

Log-based striping alone does not solve all of the distribution and synchronization problems in a striped network file system, however. First, clients must be able to share files in a consistent manner. In Zebra this mechanism is provided by the *file manager*, which is responsible for providing client access to the file system name space, for managing the file block maps, and for ensuring the consistency of client caches. Second, the contents of the storage servers must agree with the file system metadata managed by the file manager. To achieve this goal, Zebra clients store additional information in their logs called *deltas*, which describe changes in the state of the file system and are used by the file manager to ensure the consistency of the file system after a crash. Finally, the file system must keep track of its free space and allocate it to store newly-written data. In Zebra this means that the system must reclaim the free space found inside of existing stripes and coalesce it into large contiguous regions in which to store new stripes. This functionality is provided by the *stripe cleaner.*

The remainder of this chapter looks at the design choices made in Zebra, and how they interact to provide a striped network file system that requires less overhead and synchronization than a file system that uses file-based striping.

3.1 Parity Computation by Clients

The parity computation in a striped network file system must be distributed because, unlike a RAID, there is no centralized resource that contains all of the data needed to compute a stripe's parity. In a RAID, all of the data written by the clients pass through the RAID host, making it easy for the host to manage the parity of the stripes. For full stripe writes the host simply computes the parity of the stripe and writes it along with the data, and for partial stripe writes the host reads back information from the stripe as needed to compute the parity. A striped network file system, however, lacks a comparable centralized resource that has easy access to the data being written and the current contents of the stripes. One could be added, but it would be a performance bottleneck and single point of failure, since all data written would pass through it.

One possibility is to distribute the parity computation among the servers. After data have been written to a stripe, the servers exchange information to compute and store the stripe's new parity. The simplest way of doing this is to number the servers, then have each server receive a partially computed parity fragment from the previous server, XOR in its data fragment, and pass the result on to the next server. When the parity server gets the parity fragment it simply stores it. Partial stripe writes can be implemented in a similar fashion, except that parity is only exchanged between those servers that participate in the write. When a server receives a partially computed parity fragment it XORs in both the new data fragment and the old data fragment (which it reads from the disk), and sends the result on to the next server. The parity server XORs in the old parity fragment to the parity fragment it receives before storing it. This method of computing parity results in one extra network transfer for each data fragment written, since each server that receives a data fragment must also send a partially computed parity fragment to the next server in sequence.

There are several drawbacks to performing the parity computation on the servers, however. First, computing the parity for a stripe requires an extra network transfer for each server participating in the write, doubling its cost. This is true even if a full stripe write is performed, which is particularly inefficient because the client writing the stripe has all of the data needed to compute its parity. For a full stripe write it makes more sense to compute the parity on the client and avoid the overhead of exchanging data between the servers.

Second, computing parity on the servers consumes server resources and reduces server performance. For each data fragment a server receives, it must also receive and transmit a partially computed parity fragment, as well as read the old contents of the data fragment from its disk. The overhead represented by these activities limits the available write bandwidth provided by the server, thereby limiting the throughput that the server can support.

To avoid these problems, Zebra performs the parity computations on the clients, which reduces the number of network transfers required to compute parity, reduces the load on the servers, and allows the performance of the parity computation to scale with improvements in client performance. In particular, full stripe writes require fewer network transfers than if parity were computed on the servers. When a client writes a full stripe it can easily compute the stripe's parity and write it along with the stripe. This increases the number of network transfers required to write a full stripe write by only 1/N, where N is the number of servers participating the write, instead of by the N extra transfers required if parity is computed on the servers.

Computing parity on the clients does not improve the performance of partial stripe writes, however. In fact, partial stripe writes become slightly more expensive. Partial stripe writes require one extra network transfer per fragment written if the servers compute parity; not only does computing the parity on the clients require one extra network transfer per fragment written, to read back the old contents of the fragments, but it also requires two extra network transfers for the client to read and write the parity fragment. Thus the overhead of computing the parity on the client during a partial stripe write depends on the number of data fragments written. If a single fragment is written the overhead is twice that required if parity is computed on the servers. The larger the number of data fragments written, the smaller the overhead.

The higher performance of full stripe writes versus partial stripe writes makes it desirable to favor the former over the latter. This does not mean that applications can be forced to always write in units of full stripes, however; instead the file system must have a way of batching together unrelated data from many small writes by application programs into full stripe writes to the servers. The technique that makes this possible in Zebra is the use of non-overwrite updates of file blocks, as described in the next section.

3.2 Non-Overwrite

One of the most important features of Zebra is that it updates file blocks by writing new copies, rather than updating the existing copies. File systems traditionally modify a file block by modifying its storage location on the disk, so that the old contents of the block are overwritten with the new. This technique is known as *overwrite* or *update-in-place*. The result is that once a storage location has been allocated for a block, the block stays in that location until it is deleted. In Zebra, however, clients do not modify file blocks via update-in-place. Instead, a client modifies a file block by appending a new copy of the block to the end of its log, then updating the file's block map to point to the new copy rather than the old.

The use of non-overwrite to update file blocks has several advantages. First, it allows the clients to batch together blocks from different files and write them to the servers in a single transfer. Since file blocks are not modified in-place, there is no reason a client can't store blocks from several files in the same stripe. This not only amortizes the cost of writing over more bytes, but it also allows clients perform full stripe writes instead of the more expensive partial stripe writes. Second, the elimination of update-in-place frees clients to write data to any stripes they wish; just because a file block is currently stored in a particular stripe doesn't mean that the new copy of the block has to be written to the same stripe. This means that even if several clients simultaneously modify the blocks contained in a single stripe, they can write their modified blocks to different stripes and avoid the synchronization that would be required if they were to write to the same stripe.

These advantages of using non-overwrite instead of update-in-place are described in more detail in the following sections.

3.3 Write Batching

By using a non-overwrite method of updating file blocks, Zebra allows clients to batch together file blocks from different files and write them to the servers in large, efficient transfers. This improves the write performance of the system in two ways. First, large transfers amortize the overhead of performing a data transfer and allow parity to be computed over more bytes of data, reducing the per-byte costs. Even if an application performs small writes the client is able to batch these writes together into a large write to the same stripe. This allows more data to be transferred in each server access, and it avoids having to perform a partial stripe write for each file block written. If file blocks are updated in-place, then modifications to a collection of blocks each located in a different stripe will require a partial stripe write for each block. By using non-overwrite to modify file blocks, clients can write unrelated file blocks to the same stripe and perform a single parity update that covers all of the blocks. This same technique used in the write-anywhere file layout (WAFL) [Hitz94] to reduce the cost of partial stripe writes.

Second, non-overwrite makes it possible to batch together enough data to perform a full stripe write, so that not only are the transfer and parity overheads amortized over more bytes, but the overheads are reduced because additional network transfers are not needed to compute the stripe's parity. If update-in-place is used a full stripe write is only possible if the applications that a client is running happen to modify all of the file blocks in a stripe. Without update-in-place, however, full stripe writes are more likely because unrelated file blocks can be written to the same stripe.

3.4 Virtual Stripes

The second advantage of not using update-in-place to modify file blocks is that clients can write their modifications to different stripes and avoid simultaneously modifying the same stripe. If file blocks are updated in place, then two clients that wish to modify blocks in the same stripe would have to synchronize their actions so that parity is computed correctly. In Zebra, however, the clients can simply write their modified blocks to different stripes, so that they have exclusive access to the stripes they are modifying and don't need to synchronize during the parity computation.

This does not completely solve the synchronization problem, however, since the system must guarantee that only one client at a time will write its blocks to a particular stripe. Zebra provides this guarantee through the use of *virtual stripes*, which provide a level of indirection between the stripes accessed by the clients and the physical storage on the storage servers. In a RAID, striping is done based upon physical disk blocks, so that all of the blocks numbered zero from each disk form stripe zero, blocks numbered one form stripe one, etc. With physical striping the number of stripes in the system is fixed, as is the mapping from a stripe's name (address) to the disk blocks that store it.

It isn't difficult in a striped file system, however, to add a level of indirection to this mapping so that the disk blocks that store a stripe aren't fixed. Given the name of a stripe a lookup must be done to determine which disk blocks store it. Any collection of the system’s disk blocks can be used to hold the stripe, as long as it spans the servers. Once this indirection exists it is easy to expand the stripe name space so that there are more valid stripe names than there are disk blocks to hold them. The only limitation is that the number of existing stripes cannot exceed the size of the storage space.

The indirection provided by virtual stripes has several advantages in Zebra. First, it allows each client to create new stripes to hold the data it writes while guaranteeing that two clients do not create stripes with the same name. The stripe name space is simply partitioned into disjoint sets, and each client assigned to a different set. Because clients create stripes in their own partition there is no danger of two clients choosing to write the same stripe simultaneously, thus eliminating the need to synchronize when choosing a stripe to write.

Partitioning the stripes is also feasible with physical stripes, but it limits the amount of data that a single client can write. Each client can only write as much data as fits in the stripes in its partition, preventing a client from writing more data if its partition fills up, even though there may be plenty of free space in other stripes. Partitioning based on virtual stripes does not suffer from this same problem, however, since the number of virtual stripes allocated to each client can be much larger than the physical storage space in the system. The only limitation is that the total number of virtual stripes that can exist at any one time cannot exceed the storage capacity of the system.

Another advantage of using virtual stripes is that they are initially empty when they are created, making partial stripe writes less expensive. The contents of a new stripe are logically zero, as is its parity fragment. Therefore, if data are appended to a new stripe in a partial stripe write, there is no need to read the current contents of the stripe to compute the new parity, since the current contents are known to be zero. Thus, by using virtual stripes, partial stripe writes can be completed without reading data from the stripes, thereby improving their performance.

Finally, the use of virtual stripes allows the allocation of file blocks to stripes to be decoupled from the allocation of stripes to storage space. Clients are responsible for laying out file blocks in the stripes they create, which they do by forming the blocks into a log, and storing the log in the stripes. The servers, on the other hand, are responsible for taking the stripes they are given and storing them on their disks. The actual storage location of a stripe isn’t of concern to the clients, as long as the servers know where it is stored and return its contents on subsequent reads. Thus virtual stripes allow clients to allocate stripes and store file data in them, while the servers manage their own storage space.

3.5 Append-Only Writes

The combination of non-overwrite and virtual stripes allows Zebra clients to write file blocks to the servers without synchronizing with each other before doing so. While this is true no matter which stripes each client chooses to write, there is an advantage to treating the storage space in the virtual stripes as a log, as is done in Zebra. In log-based striping, the file blocks are formed into a log that is striped across the servers. Thus the data written by each client are appended to the end of its log. One implication of this mechanism is that a partial stripe write can only affect the stripe that contains the end of the client's log. If the client caches the parity fragment for the end of its log, a partial stripe write can be performed without reading the old contents of the stripe's parity fragment. While it is possible for clients to cache parity fragments in a system that is not append-only, doing so will be much less efficient since partial stripe writes may occur to any stripe in the system. In Zebra, however, if a client performs a partial stripe write it is guaranteed to be targeted at either the stripe that contains the current end of the client's log, or, if that stripe is full, a new stripe. In either case the client does not have to read the parity fragment, in the former instance because it has the parity cached, and in the latter because the parity for a new stripe is logically zero.

Another advantage of confining partial stripe writes to the tails of the client logs is that there is no need to make them atomic. The failure of a partial stripe write will only affect the tail of the log. Recovery from the failure is simply a matter of finding the tail of the log and verifying that the parity for the stripe that contains it is correct. If it is not, then a write was in progress at the time of the crash and the stripe's parity is corrected simply by recomputing it from the existing fragments of the stripe. This effectively truncates the log, and may leave the offending write in an incomplete state due to the crash. This behavior is no different from other file systems that maintain UNIX semantics.

The net result of using append-only writes and virtual stripes is that partial stripe writes in Zebra are no more expensive to perform than full stripe writes. Because writes are append-only, clients can cache the most recent parity fragment and avoid having to read it to perform a partial stripe write. Since clients write virtual stripes there is no danger of two clients simultaneously modifying the same stripe. Thus the overhead associated with a partial stripe write is simply the cost of writing the parity fragment, the same as in a full stripe write. This does not mean, however, that the performance of partial stripe writes is the same as full stripe writes. A partial stripe write transfers a smaller amount of data, so that the per-byte overheads are higher, leading to reduced performance. Even though partial stripe writes aren't as costly in Zebra as they are in other systems, it is still advantageous for each client to batch together its data to form full stripe writes.

3.6 Stripe Cleaning

One of the implications of using append-only logs to store file data is that the system must delete stripes at the same rate as new stripes are created to hold the logs, otherwise the system will run out of storage space. Clients create new stripes to store the data they write, instead of using the free space found in existing stripes. This means that although free space will appear within existing stripes as the file blocks they contain are either superceded by new copies or deleted, this free space cannot be used by the clients directly. The only way to reuse the free space in a existing stripe is for all of the data in the stripe to be unused, so that the entire stripe can be deleted and the space it occupies reclaimed by the storage servers and reused for new stripes.

The problem, therefore, is one of ensuring that there is a steady supply of empty stripes to be deleted to provide space to hold new stripes. This task is made more difficult due to internal fragmentation of free space within existing stripes. In the steady state, file data are deleted at the same rate at which they are created. This means that free space will appear in existing stripes at the same rate at which data are written to new stripes, but there is no guarantee that this free space will naturally coalesce into large regions that span entire stripes. If it does not, the system will run out of empty stripes to delete, even though there is plenty of free space in existing stripes.

In Zebra the tasks of reducing internal fragmentation of free space and producing empty stripes to be deleted are handled by the *stripe cleaner*. The stripe cleaner operates by *cleaning* stripes so that they do not contain any data that are in use, thereby producing empty stripes to be deleted and reducing the internal fragmentation of the free space. A stripe is cleaned by copying any live data it contains to the end of the stripe cleaner’s client log, which moves the data to a new stripe and leaves the old copies unused. The functioning of the stripe cleaner is described in greater detail in the next chapter.

3.7 Centralized Metadata Management

Zebra’s log-based striping makes it possible for multiple clients to share a collection of storage servers without requiring synchronization when writing stripes, but it does not address the issue of allowing clients to share files. There are three problems that must be solved. First, the system must provide a mechanism by which a file block written by one client can be read by another client. When a client writes a file block to its log, only that client knows which stripe stores the block. If another client wishes to read the block it must somehow discover the block’s storage location from the client that wrote it. Second, the contents of the client caches must remain consistent, so that clients see consistent views of the file contents. A client should not read obsolete file data from its cache, for example. Third, clients must be able to share the file system name space in a consistent fashion. For example, two clients should not be able to create two different files with the same name.

In Zebra these tasks are handled by the *file manager*. The file manager’s role is to manage the file system metadata. It synchronizes client accesses to the metadata so that the clients see consistent views of the file system. First, the file manager manages the block maps that keep track of where each file block is located. When a client writes a file block it notifies the file manager of the block’s new location, allowing other clients to determine the block’s storage location by querying the file manager. In this manner the file manager is able to ensure that clients always read the most recent copies of file blocks. The file manager operation is described in more detail in Section 4.4.

The file manager also ensures that the client caches remain consistent. The mechanism for doing this is based on the Sprite cache consistency protocol. When clients open and close files they send messages to the file manager so that it can keep track of which clients are modifying which files. If a client tries to open a file for which it has an obsolete cached copy it is notified by the file manager to discard the copy. Thus clients never access out-of-date file data from their caches.

Finally, each operation on the file system name space is handled by the file manager. If a client wishes to create a file, for example, it sends a request to the file manager. The file manager can therefore synchronize accesses to the file system name space to ensure that inconsistencies do not occur.

3.8 Consistency via Deltas

One of the problems of having the file manager maintain the file system name space is that file blocks are stored on the storage servers, but the block maps that keep track of their locations are managed by the file manager. Any inconsistency between the two can lead to lost file data. For example, if a file is written to the storage servers, but a failure prevents the file manager from updating its block map, the blocks of the file may become inaccessible.

The solution to this problem to make the act of writing file data and updating the file's block map atomic. To do this, Zebra borrows a technique from database systems known as *write-ahead logging*. A write-ahead log is used to record a set of actions that are about to be performed, before actually performing them. Once this information is safely stored in the log, the actions are then undertaken. If a failure prevents all of the actions from occurring, the log can be used to return the system to a consistent state. This is done by using the contents of the log to finish the operations or to undo the effect of those that have already been done.

A similar solution can be used to ensure that writing a file block and recording its location is atomic. Furthermore, it is simple to integrate with log-based striping since the system already contains a log. In addition to storing file blocks in the client logs, Zebra also stores descriptive information about the blocks called *deltas*. Each block stored in a log has a delta stored along with it which describes the current and previous storage locations of the block. This information is used after a crash to ensure that the file's block map has been updated properly to reflect the newly written block. The details of how deltas are processed by the file manager are given in Section 4.4.2.

3.9 Summary

The most novel feature of Zebra is its use of log-based striping to store file data, rather than the file-based striping preferred by other striped file systems. Zebra's log-based striping is advantageous in several ways. First, it prevents clients from having to synchronize before writing to a stripe. Each client writes its own log to its own set of virtual stripes, so there is no danger of multiple clients modifying the same stripe simultaneously. Second, partial stripe writes are confined to the ends of the client logs because the logs are append only. This means that a partial stripe write need not be atomic, since a failure can be corrected by verifying the parity of the stripe at the tail of the client's log and truncating the log if necessary. Third, since the logs are append only and are stored in virtual stripes, there is no need to read information from a stripe during a partial stripe write. Data can only be written to a part of the stripe that was previously empty, and therefore logically zero, and the client can cache the parity fragment for the last stripe in its log to avoid reading it during a partial stripe write. Finally, log-based striping allows clients to batch together file data from different files and write them to the servers in large transfers. This not only improves server efficiency, but also increases the likelihood that clients will be able to perform full stripe writes, instead of the more costly partial stripe writes.

The remaining issues of free space management and file system metadata management are handled in Zebra by the stripe cleaner and the file manager, respectively. The stripe cleaner produces empty stripes whose storage space can be reused to hold new stripes. It does this by copying live data out of existing stripes and appending them to the tail of its log. The file manager maintains the file system metadata, such as its name space and block maps. Clients interact with the file manager to access the metadata, allowing the file manager to ensure that the clients see consistent views of the file system. This does present a problem, however, since the file blocks stored by the storage servers must agree with the block maps maintained by the file manager. Zebra solves this consistency problem through the use of deltas, which allow changes in the state of the file system's blocks to be recorded in the log so that the state can be reconstructed after a crash.

4 Zebra Architecture

This chapter describes the Zebra architecture, including descriptions of the components of a Zebra file system and how they interact to provide file service to the application programs, how the system tolerates failures, and how each component recovers from a failure. The Zebra components are shown in Figure 4-1, and consist of *clients*, which are the machines that run application programs; *storage servers*, which store file data; a *file manager,* which manages the file system metadata; and a *stripe cleaner,* which reclaims unused space on the storage servers. More than one of these components may share a physical machine. The file manager and stripe cleaner are system services that may run on a single client, for example. It is also possible for a storage server to be a client. In the figure, however, the storage servers and clients are shown as separate machines.

A single Zebra file system is defined to be a directory hierarchy that is managed by a single file manager. There may be multiple clients, storage servers, and stripe cleaners in the system, but only one file manager. Several file managers may be composed into a larger file system by piecing together their directory hierarchies into an overall hierarchy,

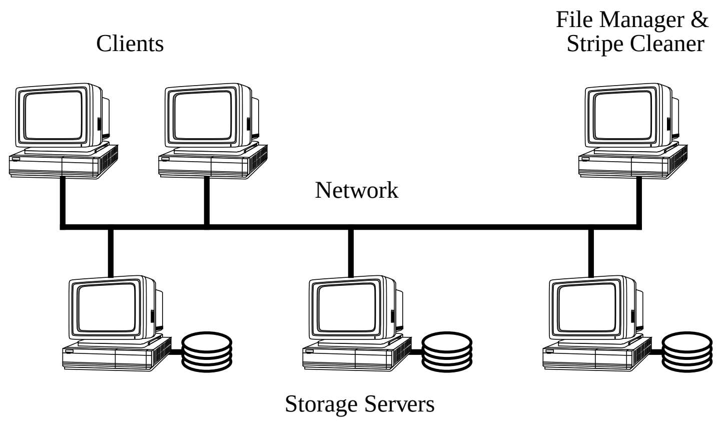

Figure 4-1. Zebra components.
Clients run applications; storage servers store data. The file manager and stripe cleaner can run on any client in the system, although it is likely that one client will run both of them.

but this does not affect the design of the individual file systems. For this reason Zebra is described as if there is only a single file manager. A similar simplification is made for the stripe cleaner. There may be several stripe cleaners in a file system, but it easier to describe the operation of the stripe cleaner if it is assumed that only one exists.

The Zebra architecture is also described under the assumption that each storage server has only a single disk. However, this need not be the case. For example, storage servers could each contain several disks managed as a RAID, thereby giving the appearance to clients of a single disk with higher capacity and throughput.

4.1 Log Addresses

One of the consequences of using log-based striping instead of file-based striping is that the storage servers provide a different storage abstraction than a traditional file server. In a traditional network file system the file server interface provides a logical file abstraction. Clients access data stored on the server by reading and writing file blocks, and the file server uses the files' block maps to determine which disk blocks to access. Each request specifies the file ID and logical block number to be accessed. The file server uses the file ID to find the file's block map (inode), then uses the block map to find the disk block that contains the desired logical file block.

In contrast, the storage server interface in Zebra implements a log abstraction, so that the data on a storage server are accessed by their locations in the client logs, rather than their locations in files. The location of a block of data within the client logs is specified by its *log address*, which includes the identification of the log that holds the data and the offset of the data within the log. Thus in Zebra the mapping from logical file block to physical disk block is divided into two phases: in the first phase the client converts the logical file block into its log address, and in the second phase the server converts the log address into a physical disk address.

Log addresses are easily parsed to determine which client produced the data stored at a given address, and which server stores them. This parsing is shown in Figure 4-2. The log ID identifies which client created the log. The stripe index is the sequence of the stripe within the log, and the fragment index is the sequence of the fragment within the stripe. The fragment index can therefore be used as a server index. The combination of the log ID and stripe index is called the *stripe ID*, since it uniquely identifies each stripe of the system. The stripe ID plus the fragment index is called the *fragment ID* because they uniquely identify the fragment.

4.2 Storage Servers

The storage servers are the simplest components of Zebra: they are merely repositories for stripe fragments. To a storage server a stripe fragment is simply a collection of bytes identified by its log address. Each storage server keeps a fragment map that maps the fragment ID of each fragment to its disk address.

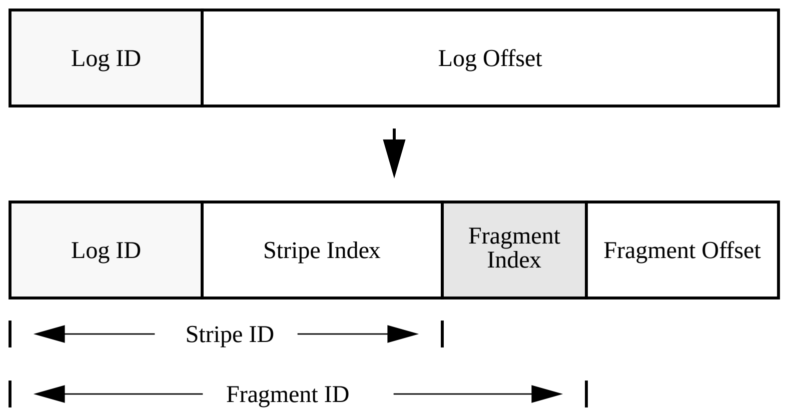

Figure 4-2. Log address parsing.

A log address can be broken down into an index of the stripe within the log, an index of the fragment within the stripe, and an offset within the fragment. The log ID plus stripe index is the *stripe ID* and uniquely identifies a stripe; the stripe ID plus the fragment index is the *fragment ID* and uniquely identifies a fragment plus the server that stores it.

The use of log addresses to access data on the servers results in less server overhead than if logical file addresses were used. The server does not interpret the contents of the fragments that it stores, reducing the per-file or per-block overheads on the storage server when reading or writing data. Since a fragment is much larger than a file block or an average file (a fragment is 512 Kbytes in the prototype) the load on the server CPU is substantially reduced.

In addition, since the server does not interpret the contents of the fragments it stores there is no need for the fragments to actually cross the host backplane at all. Some server architectures, such as RAID-II [Drapeau94], implement a high-performance data path between the network and the disk subsystem. This data path allows data to flow between the networks and the disks without being copied to the host memory system across the host backplane, as would be the case in a traditional file server architecture. This increases the file server performance by avoiding the bottleneck presented by the host backplane, but in a traditional file system it complicates the file server software because the data being stored by the server is not easily accessible to the host CPU. The Zebra architecture, on the other hand, circumvents this problem because there is no need for the host to actually look at the contents of the fragments it stores.

4.2.1 Functionality

Storage servers provide six operations on fragments as illustrated in Table 4-1. The Zebra architecture distinguishes between those fragments that store file data, called *data fragments*, and those fragments that store the parity of a stripe, called *parity fragments*. The Zebra storage servers handle data fragments and parity fragments in a slightly different manner. A data fragment is created via a *store* operation and may be appended to via *append* operations (this allows clients to store amounts of data smaller than the size of a fragment). It is an error to attempt to store a data fragment that already exists; this ensures that existing data in a data fragment cannot be overwritten.

Operation	Parameters	Effects
Store Data Fragment	Fragment ID Size Checksum	Stores the fragment on the disk. *Checksum* is the checksum of the fragment. It is an error if the fragment already exists.
Append to Data Fragment	Fragment ID Size Checksum	Appends data to an existing data fragment. *Checksum* is the checksum of the entire fragment including the appended data.
Store Parity Fragment	Fragment ID Size Sequence Checksum	Same as for storing a data fragment, except that if the fragment already exists the new copy replaces the old. The *sequence* must increase for each new copy.
Retrieve Fragment	Fragment ID Offset Size	Returns *size* bytes starting at *offset* within the fragment. A list of offset/size pairs may be specified for the same fragment.
Delete Fragment	Fragment ID	Deletes the fragment.
Last Fragment	Client ID	Returns the fragment ID of the most recent fragment stored or appended to by the client.

Table 4-1. Storage server interface.
The set of operations used by clients to store, retrieve, and delete fragments on a storage server.

Parity fragments, on the other hand, can be overwritten but cannot have data appended. A parity fragment may be overwritten when a stripe is created via a series of partial stripe writes (a write that does not span an entire stripe). Each partial stripe write can cause a new version of the parity fragment to be stored. If a parity fragment is stored when a copy already exists the new copy replaces the old. The *sequence* parameter increases for each new copy of a parity fragment stored and is used during crash recovery to distinguish between multiple copies of the parity fragment (see Section 4.2.2).

All storage server operations are synchronous and atomic. A *store* operation, for example, does not return until the fragment is safely stored on the disk. Furthermore, a server failure during an operation does not leave the operation partially completed. During recovery the server will determine whether the operation completed successfully. If the operation only partially completed before the crash, the effects of the operation are undone. Thus a failure during an append operation either results in all of the data being appended or none of them.

Failures external to the server may also cause an operation to fail. If the client fails to send all of the data associated with a store operation then obviously the operation cannot succeed. A communication failure of this sort causes the storage server to abort the operation.

4.2.2 Crash Recovery

When a storage server crashes and recovers it must ensure that its state is both internally and externally consistent. *Internal consistency* means that the on-disk data structures maintained by the server are consistent with one another. These data structures may be inconsistent if the crash occurred during a store or append operation. *External consistency* means that the contents of the server are consistent with the contents of the other servers in the system. In particular, after a crash a server will lack fragments from any stripes written while it was down. As part of recovery the server must reconstruct and store these fragments.

There are two ways in which a storage server may be internally inconsistent after a crash. The first is that a store or append operation may have been in progress at the time of the crash, leaving the new data only partially written to disk. This problem is solved through the use of fragment checksums. The store and append operations take a fragment checksum as a parameter as well as the data to be written. This checksum is compared with the contents of the fragment after a crash. A mismatch indicates that the operation did not succeed. It is only necessary to verify the checksums of those fragments that were being modified at the time of the crash, although a naive implementation can simply check all of the fragments stored on the disk. The next chapter explains how the storage servers in the Zebra prototype limit the number fragments that need to be checked after a crash.

It is important that a crash during an append operation not cause the entire fragment to be lost, as would be the case if the old fragment checksum were overwritten by the new. Should a crash occur between writing the checksum and data the checksum would no longer correspond to the contents of the fragment, requiring the entire fragment to be discarded. This problem can be avoided by storing the new checksum for an append operation in a non-overwrite manner. Should a crash occur, the server can thus revert to the old checksum for the fragment and avoid losing its previous contents.

The second possible internal inconsistency is that a crash may result in the server having two copies of the same parity fragment. New copies of a parity fragment are stored in a non-overwrite manner, so that a crash during the store of a new copy does not leave the previous version of the fragment corrupted. Once the new copy has been safely stored the server updates its fragment map to point to the new copy and discards the old copy. If a crash should occur, however, between storing the new copy and updating the fragment map there will be two copies of the same parity fragment. This ambiguity is solved using the sequence numbers associated with parity fragments. Each new copy of a parity fragment is assigned a higher sequence number than the current copy. After a crash all copies of the fragment are discarded except for the copy with the highest sequence number. It should be noted, however, that while this mechanism ensures that the storage server always ends up with the most recent copy of a parity fragment, it does not ensure that the contents of the parity fragment are consistent with the other fragments in the stripe. Such an inconsistency may occur if a client crashes while it is writing a stripe, and only writes some of the stripe's fragments. Zebra's handling of this potential inconsistency due to a client crash is described in Section 4.4.5.

After a recovering storage server verifies that its state is internally consistent it must verify that the fragments it stores are consistent with the fragments stored on the other servers. Once again there are two possible inconsistencies: first, the server will not contain the fragments of any stripes created while it was down; and second, the server will still contain fragments of stripes that were deleted while it was down. The former is corrected by the server itself during recovery. The recovering server invokes the *last fragment* operation on the other servers in the system. The server uses this information to determine the most recent stripe written by each client and compares them to the most recent fragments that it stores. Any missing stripe fragments are reconstructed using the contents of the other fragments in the stripe and stored by the server, bringing itself up-to-date. Each fragment contains a descriptive block of data that makes it possible to detect whether or not a reconstructed fragment is valid; if the reconstructed fragment has not yet been written by the client the result will be a block of zeros without a valid description, and is ignored by the server. Details of the fragment format in the prototype are given in Section 5.2.1.

A more difficult problem is that some stripes may have been deleted while the server was down. The only way a recovering server can detect this inconsistency is to compare a complete list of the fragments it stores with the corresponding lists on the other servers. For large systems this comparison may require a significant amount of resources. For this reason Zebra storage servers depend on an external agent to replay the delete messages after a server recovers. If a delete fragment operation fails because the server is down the operation must be retried once the server reboots. This is described in more detail in Section 4.5.6.

4.3 Clients

Clients are the machines where application programs execute. The client retrieves the appropriate fragments when an application reads from a file, and uses log-based striping to stripe newly written file blocks across the storage servers. The mechanics of reading and writing files during normal operation (no failures) are outlined in the following two sections, followed by a section describing how fragments are reconstructed during a server failure.

4.3.1 Reading Files

The interaction between a client and the storage servers when reading files is relatively simple. To read a file block the client obtains the file's block map from the file manager and uses it to determine the log address of the desired block. The details of how the client obtains the block map are presented in Section 4.4.3. Once the block's log address is known it is parsed to determine which fragment contains the desired portion of the log and which storage server stores the fragment. A retrieve operation is then used to obtain the desired data from the server. For large files read-ahead is used to keep all of the storage servers busy. This ensures that file blocks are being transferred from all of the storage servers concurrently, and keeps all of the servers' disks busy reading the next fragment from disk while the previous one is being transferred over the network to the client.

A Zebra client does not attempt to optimize reads of small files: each file is read from its storage server in a separate operation, just as for a non-striped file system. However, it is possible to prefetch small files by reading entire stripes at a time, even if they cross file boundaries. If there is locality of file access so that groups of files are written together and then later read together, this approach might improve read performance. I speculate that such locality exists but I have not attempted to verify its existence or capitalize on it in Zebra.

4.3.2 Writing Files

For Zebra to run efficiently, clients must collect large amounts of new file data and write them to the storage servers in large batches (ideally, whole stripes). Zebra clients use write-back caches that make this batching relatively easy to implement. When an application writes new data, they are placed in the client's file cache and aren't written to the server until either (a) they reach a threshold age (30 seconds), (b) the cache fills with dirty data, (c) an application issues an `fsync` system call to request that data be written to disk, or (d) the client cache consistency protocol requests that data be written in order to maintain consistency among client caches. In many cases files are created and deleted before the threshold age is reached so their data never need to be written at all [Baker91].

When information does need to be written to disk, the client forms the new data into one or more stripe fragments and writes them to the storage servers. The client computes the parity as it writes the fragments and at the end of each stripe the client writes the parity to complete the stripe. To benefit from multiple storage servers it is important for a client to transfer fragments to all of the storage servers concurrently. A client can also transfer the next stripe fragment to a storage server while the server is writing the current stripe fragment to disk, so that both the network and the disk are kept busy.

If a client is forced to write data in small pieces (e.g., because an application invokes `fsync` frequently) then it fills the stripe a piece at a time, appending to the first stripe fragment until it is full, then filling the second fragment, and so on until the entire stripe is full. When writing partial stripes the client has two choices for dealing with parity. First, it can delay writing the parity until the stripe is complete. This is the most efficient alternative and it is relatively safe since the client's copy of the unwritten parity fragment can be used to reconstruct stripes in the case of a server failure. The contents of a stripe will only be lost if a disk fails and the client crashes before writing the parity.

For even greater protection the client can store a new copy of the stripe's parity fragment each time it appends to the stripe. The storage server will replace the old copy of the fragment with the new. This alternative is slower because it requires the parity to be written for each partial stripe write, but it will only lead to data loss if two disks fail, which is even less likely than a dual failure involving a disk and a client.

The rate at which applications invoke `fsync` will have a large impact on Zebra's performance (or any other file system's) because `fsync`s require synchronous disk operations. Baker et. al [Baker92b] found that under a transaction processing workload up to 90% of the segments written on an LFS file system were partial segments caused by an `fsync.` Such a workload would have poor performance on Zebra as well. Fortunately, they found that on non-transaction processing workloads `fsync` accounted for less than 20% of the segments written. The average size of these partial segments was about 20 Kbytes. Based upon measurements of the Zebra prototype presented in Chapter 6, the bandwidth of writing partial segments of this size is only half that of writing full segments, so that `fsync` reduces the Zebra write bandwidth by less than 10%.

4.3.3 Storage Server Crashes

Zebra's parity mechanism allows the clients to tolerate the failure of a single storage server using algorithms similar to those described for RAIDs [Patterson88]. To read a file while a storage server is down, a client must reconstruct any stripe fragment that was stored on the down server. This is done by computing the parity of all the other fragments in the same stripe; the result is the missing fragment. The format of log addresses makes it simple to find the other fragments in the stripe, since all of the fragments in the same stripe have the same stripe ID. Writes intended for the down server are simply discarded; the storage server will reconstruct them when it reboots, as described in Section 4.2.2.

For large sequential reads reconstruction is relatively inexpensive: all the fragments of the stripe are needed anyway, so the only additional cost is the parity calculation. For small reads reconstruction is expensive since it requires reading all the other fragments in the stripe. If small reads are distributed uniformly across the storage servers then reconstruction doubles the average cost of a read.

4.4 File Manager

The previous section on reading and writing files neatly avoided the issue of managing the file system metadata. In a UNIX file system the metadata includes file attributes such as protection information, disk addresses of file blocks, directories, symbolic links, and special files for I/O devices. The metadata for each UNIX file consists of an inode containing the file attributes and disk addresses for the file blocks, as described in Section 2.1. Zebra stores the file metadata in a similar inode structure, except that the block addresses contained in the inode are log addresses instead of disk addresses, since log addresses are used to access data on the servers. I refer to these log addresses as *block pointers*. Clients use the block pointers to read data from the storage servers and must update the pointers after writing to a file, since writing a file block causes it to be appended to the client's log and thereby changes its log address. Furthermore, these accesses to the block pointers must be synchronized, otherwise it is possible for clients to see inconsistent views of the file system by using out-of-date block pointers.

The need for synchronization applies not only to the block pointers, but the file system metadata as a whole. Unsynchronized accesses to the file system name space or file attributes, for example, can lead to inconsistent views due to the use of out-of-date metadata. For example, without synchronization it is possible for two clients to simultaneously create two files with the same name. In most network file systems the synchronization is done implicitly, because only the file server accesses the metadata directly, allowing the server to serve as a synchronization point for the client accesses. In a striped network file system, however, there may not be a corresponding centralized location to perform the synchronization.

Zebra solves the metadata synchronization problems via a centralized service called the *file manager* that manages the file system metadata. The file manager performs many of the usual functions of a file server in a network file system, such as name lookup and maintaining the consistency of client file caches. However, the Zebra file manager doesn't store any file data; where a traditional file server would manipulate data the Zebra file manager manipulates block pointers. The file manager can be thought of as a librarian for these pointers. If a client wishes to read a file block and it doesn't know where the block is located, it asks the file manager for the block's log address. Similarly, when a client writes a file block it must notify the file manager of the block's new log address so that the block pointers can be updated. For example, consider a read operation: in a traditional file system the client requests the data from the file server; in Zebra the client requests block pointers from the file manager, then uses the block pointers to read the data from the storage servers. Figure 4-3 illustrates this sequence of events.

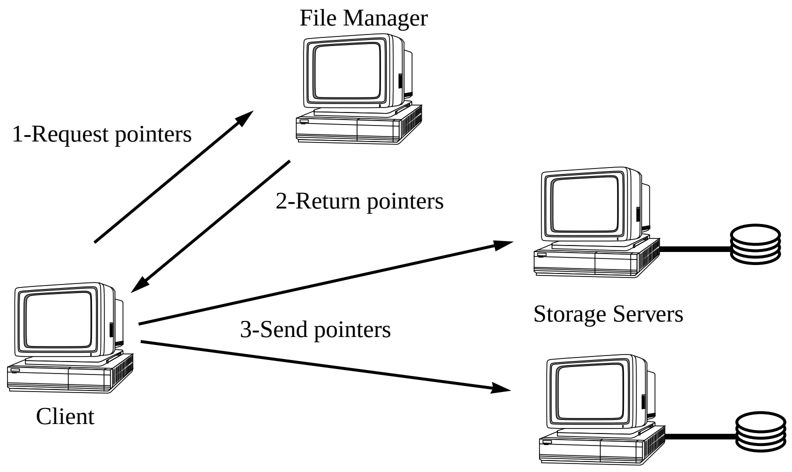

Figure 4-3. Reading from a file.

To read from a file the client first requests the file's block pointers from the file manager (assuming it doesn't have them cached), then uses the block pointers to retrieve the appropriate fragments from the storage servers.

The use of a file manager to manage the file metadata serves two purposes in Zebra: it eliminates the need for the clients to understand the metadata format, and it provides a central synchronization point for metadata modifications. To access a file a client only needs to get a list of block pointers from the file manager; it doesn't need to understand the format of an inode. To write a file a client only needs to give the new block pointers to the file manager, instead of modifying the inode directly. This allows the format of the metadata to be changed without modifying the clients. Second, the file manager synchronizes modifications to the metadata. Clients send changes to the block pointers to the file manager, which can ensure that simultaneous modifications do not occur.

4.4.1 Client Cache Consistency

If a network file system allows clients to cache file data and also allows files to be shared between clients, then cache consistency is a potential problem. For example, a client could write a file that is cached on another client; if the second client subsequently reads the file, it must discard its stale cached data and fetch the new data. Zebra's cache consistency mechanism is similar to that used in Sprite [Nelson88]. Clients notify the file manager when they open and close files. This allows the file manager to keep track of which clients are caching which files, and whether or not they have any dirty blocks for the file. If a client opens a file for which another client has dirty blocks the file manager first notifies the latter client to flush the dirty blocks out to the storage servers. The file manager then applies the new block pointers to the file's metadata, and allows the former client to complete the file open. The client doing the open can then request the updated block pointers from the file manager. Similarly, if during an open it is discovered that the client's cached copy of a file is obsolete the file manager tells it to discard its cached copy and to fetch new block pointers.

The situation is a bit more complex if a file is simultaneously open on several clients and at least one of them is writing to it (concurrent write-sharing). In Sprite this causes the file server to notify all of the clients that the file is not cacheable and all read and write accesses must go directly through to the server. This ensures that the clients see a consistent view of the file because they are all accessing the same copy. Zebra, however, doesn't have a file server that can fill this role. One possibility is to force the file to be cached on the file manager and have it handle all of the reads and writes. Unfortunately, this solution doesn't scale well to larger number of clients. Alternatively, one of the clients can be chosen as the synchronization point. Only that client is allowed to cache the file and all read and write requests by the other clients must be sent to the anointed client. This solution scales better since it avoids a central service that must handle all cases of concurrent write-sharing.

The cache consistency mechanism must tolerate client and file manager crashes. In particular a file manager crash should not cause it to lose track of which files clients are currently caching and accessing. The mechanism used by Zebra to recover the cache state is similar to that used in Sprite. During file manager recovery the clients inform the file manager of which files they have open or have dirty blocks for in their caches. The file manager reconstructs the cache consistency state from this information.

4.4.2 Updating Block Maps via Deltas

When a client writes a file block to its log, and thereby changes the block's log address, it must communicate this change to the file manager so that the file's block map can be updated. A failure to do so leaves the newly written block inaccessible by the file system, even though the block may have been successfully written to the storage server. Thus a mechanism is needed for ensuring that writing a block and updating the file's block map are atomic events: either they both happen or neither do. At first a solution would seem to require a two-phase commit protocol to coordinate writing both the file block and the block map. One of the biggest breakthroughs in the Zebra design was the realization that the logs themselves can be used as reliable communication channels, avoiding the need for a complex two-phase commit. The logs are reliable because they are striped across the servers and backed up by parity. The logs can be used for communication because they are append-only. Messages written to the log are easily read back in the same order. Thus a client can communicate block pointer changes to the file manager simply by writing them to its log and allowing the file manager to read them as it needs to. The change in a block's pointer is stored in the same stripe fragment as the file block itself, which guarantees that they are written atomically because stripe fragment writes are atomic.

To make it possible to use them as reliable communication channels the logs contain two kinds of information: file blocks and *deltas*. A delta identifies a change in a file block's storage location, and is used to communicate this change between the client that wrote the block and the rest of the system. For example, when a client writes a file block to a stripe fragment it also stores a delta for the block in the same fragment. The file manager subsequently reads the delta from the log and uses it to update the block pointer for the block. Deltas are created whenever blocks are added to a file, deleted from a file, or overwritten. An overwritten file block requires a delta because the new copy of the block is appended to the client's log, changing the block's log address and leaving the old copy of the block unused.

There are several types of deltas in Zebra. Deltas generated by clients as described are called *update deltas,* since they describe an update to a block's location. Other types of deltas are used for communication between other components of the system and are described later in this chapter. Update deltas have the following contents:

Field	Bytes	Description
File ID	4	Identifies the file to which the delta applies.
File Version	4	Version of the file to which the delta applies.
Block Number	4	Index of the block within the file.
Block Size	4	Size of the block, in bytes.
Old Block Pointer	8	The block’s old log address (NULL if the block is new).
New Block Pointer	8	The block’s new log address (NULL if the block has been deleted).
Modification Time	4	The time when the block was last modified.

Table 4-2. Update delta format.
The contents of an update delta.

The deltas in the logs represent a history of the file system: by looking at the deltas for a file block one can follow the changes made to that block and learn of its current storage location from its most recent delta. The old block pointer for each delta will match the new block pointer of the previous delta, much like a sequence of dominoes. In theory the file system does not need to maintain any block maps for the files it stores: a file block can be found simply by searching the logs for its most recent delta. In reality this is a rather inefficient way of accessing a file block; it is much faster to keep the block map for the file up to date so that the block can be found without searching the logs. The block map itself is kept current, however, by processing the logs. The file manager reads deltas as they appear in the client logs and applies them to the block map.

Having the file manager process the deltas to keep the block maps up to date does introduce a complication: when a client asks for the block pointers for a file how does the file manager know that the file’s block map is current and that there aren’t any outstanding deltas yet to be processed? The answer is that the cache consistency mechanism keeps track of client modifications to files and can therefore be used to determine if there are outstanding deltas for a file. As part of maintaining client cache consistency the file manager keeps track of which files each client has modified. If another client opens one of these files the file manager must first notify the client with the dirty blocks to flush them to the storage servers (if it hasn’t done so already), then the file manager must process the deltas for the modified file before allowing the open to complete. Thus when the file manager knows that a client has modified a file it also knows that it must process the deltas associated with the modification.

As described, the system for updating the block maps is reliable but relatively inefficient. The clients write deltas to their logs, and the file manager reads them back and applies them to the block maps. This is much more expensive than simply having the clients send the deltas directly to the file manager because reading the deltas from the logs requires accesses to the storage servers and possibly disk accesses. To avoid this performance hit Zebra uses two mechanisms for getting the deltas to the servers. The primary transfer method is driven by the clients and provides unreliable yet efficient service. After a client has successfully written a fragment to a storage server it sends the deltas from the fragment to the file manager. The file manager stores them in a buffer of deltas it has received but not yet processed. When the file manager needs to apply the deltas to the block maps (as determined by the cache consistency algorithm) it simply reads them from this buffer. If they are not found in the buffer then an error has occurred in sending them from the client to the file manager. In this case the file manager uses the previously described mechanism for accessing deltas, which is reliable but inefficient. It simply fetches the relevant portions of the client's log from the storage servers, extracts the deltas, and applies them to the block maps. During normal operation the primary path is used, providing an efficient way of updating the block maps, but should an error occur the backup path can be used to reliably access the deltas.

4.4.3 Reading and Writing Revisited

The reading and writing of a file as described in Section 4.3 glossed over the handling of the file's block map. These scenarios can now be updated to include the file manager and how it manages the block map. Reading a file block is a two-step operation. First the client must fetch the block pointer from the file manager, then use it to fetch the block itself from the appropriate stripe fragment. This results in an extra network message relative to a non-striped file system. For large files the overhead of this message is negligible when compared to the overall number of messages required to transfer the file data. For small files the overhead of fetching the block pointers can be reduced by returning the first few block pointers of the file as the result of the open operation. Block pointers are relatively small compared to the blocks themselves, so that the pointers for many blocks can be returned in the result, with minimal impact on the open performance.

When writing a file the client must store deltas for the file blocks in its log along with the blocks themselves. These deltas contain the old block addresses as well as the new, so if the client is overwriting existing blocks it must first fetch the block pointers from the file manager so that the deltas can be filled in correctly. After the client has written a stripe fragment to a storage server it sends the deltas from that fragment to the file manager to be processed.

4.4.4 Performance

The performance of the file manager is a concern because it is a centralized resource. It must perform name lookups for all of the clients and provide access to the file block pointers, and hence may be a performance bottleneck. There a several ways of avoiding this bottleneck, however. The first is to allow clients to cache block pointers. Once a client has retrieved the block pointers for a file there is no need to fetch them from the file manager again until they become obsolete. Second, clients can cache naming information so that the file manager need not be contacted for most opens and closes. Client-level name caching has been used successfully in the AFS file system [Howard88] and Shirriff found that a name cache occupying only 40 Kbytes of a client's memory can produce a hit rate of 97% [Shirriff92]. Client name caching has not been implemented in the Zebra prototype described in Chapter 5 because of the difficulty in adding it to the Sprite operating system, but I would expect that a production version of Zebra would do so. And third, multiple file managers can be used, each responsible for a different portion of the file system name space. A similar solution is used by current network file systems to avoid file server bottlenecks. While not optimal in terms of load-balancing, it should suffice for systems with only a few file managers.

4.4.5 Client Crashes

The crash of a client has two effects on the file system that must be rectified by the file manager: the contents of the client's cache and the state of its applications are lost, so that the client can no longer participate in the cache consistency protocol; and second, the client may have been writing a stripe at the time of the crash, perhaps leaving the stripe's data inconsistent with its parity. When the file manager loses communication with a client it assumes the client has crashed and cleans up the cache consistency state associated with that client. Any open files are closed, and any write tokens owned by the client are reclaimed. The file manager then processes any unprocessed deltas in the crashed client's log. Once the client's state has been cleaned up in this manner other clients can open files that were previously cached on the crashed client.

The file manager must also deal with inconsistent stripes caused by a client crash. If the client was writing a stripe at the time of the crash then only some of the fragments may have been written, leaving the stripe's data fragments inconsistent with its parity fragment. This inconsistency must be resolved as soon as possible since the stripe is vulnerable to a server failure; therefore it cannot be delayed until the crashed client reboots. Thus it falls to the file manager to fix it. When a client crashes the file manager detects the crash and verifies the consistency of the last stripe written by the client. It does so by querying the storage servers to identify the end of the client's log (using the *last fragment* request), then confirming that the last stripe of the log is complete and the parity correct. If a stripe is missing a single fragment then the missing data can be reconstructed using the other stripes in the fragment. Similarly, if the parity fragment is incorrect it is computed from the other fragments in the stripe. If a stripe is missing more than one fragment then the log is truncated to the first missing fragment, and the parity computed for the remaining portion of the stripe. This means that data being written at the time of a crash can be lost or partially written, just as in other file systems that maintain UNIX semantics.

4.4.6 File Manager Crashes

The file manager is a critical resource for the entire system because it manages all of the file system metadata, including the block maps. Without the file manager the clients cannot access the metadata, and without the metadata the files cannot be accessed. The approach employed by Zebra to ensure that the file manager is both highly reliable and highly available is to design the file manager so that it can run on more than one machine and can recover quickly from crashes. Using these techniques even a hardware failure of the machine hosting the file manager can be tolerated by starting a new file manager on another machine.

The first step in providing highly reliable and available service from the file manager is to ensure that it is not tied to one particular machine in the system. If the metadata is stored non-redundantly on the file manager (on a local disk, for example) then the file system will be unusable whenever the file manager's host is down, and the loss of the file manager's disk will destroy the file system. For this reason the file manager stores the file system metadata on the storage servers, rather than on a local disk, making the metadata both highly reliable and available. It is reliable because Zebra's parity mechanism allows the contents of the storage servers to be reconstructed in the event of a server failure. It is available because the contents of the storage servers can be accessed by any machine in the system. If the client that is running the file manager should suffer a hardware failure another client can easily take over as the file manager since it too can access the metadata. A similar approach has been proposed by Cabrera and Long for the Swift file system [Cabrera91] to make its storage mediator highly available.

The metadata is stored on the storage servers in a virtual disk implemented as a Zebra file. This file, called the *virtual disk file*, is stored in the file manager's client log just like any other Zebra file, but it has a special file number that identifies it as containing the virtual disk. The file manager reads and writes the virtual disk, and these accesses are translated into reads and writes of the underlying Zebra file. Implementing the virtual disk on top of a Zebra file not only ensures that the metadata is highly available and reliable, because the log is protected by parity, but it also improves the performance of accessing the metadata because it is striped across the servers.

The second step in making the file manager highly available and reliable is to ensure that it can quickly recover from a crash. There are three aspects to this: recovering the current contents of the virtual disk file, recovering the state of delta processing, and recovering the distributed state of the file system not related to striping and parity, such as the state of the read and write tokens. The last item is not unique to Zebra: any file system must recover its distributed state after a crash. For example, in Sprite a recovering file server determines the states of the client caches and the files each client has open by querying the clients. The issues involved in recovering the distributed state of a network file system are outside the scope of this thesis, but are covered in great detail in Baker's thesis [Baker94].

The first Zebra-related issue in file manager recovery is to recover the contents of the virtual disk file at the time of the crash. One caveat, however, is that any dirty blocks for the virtual disk file that were in the file manager's cache when the crash occurred are lost, and cannot be recovered. The best that can be done is to recover the virtual disk blocks that were written to the log prior to the crash. One way to do this, albeit a slow one, is to start at the end of the file manager's client log and work backwards through it looking for deltas for the virtual disk file. While this method does produce the correct result, it is likely to be unacceptably slow because it must process the entire log. A better solution is to periodically issue a *checkpoint* that contains a snapshot of the current block map for the virtual disk file. The file manager checkpoint is another special Zebra file, identified by its file number, that the file manager stores in its log. To create a checkpoint file the file manager first writes any dirty cache blocks for the virtual disk file to its log, to ensure that its block map is consistent with the data blocks stored in the log, then writes out a checkpoint file containing the block map. During recovery the file manager looks backwards through its log until it finds the checkpoint file. It then reads the virtual disk file's block map from the checkpoint, and processes the log from the checkpoint to the end and applies any deltas pertaining to the virtual disk file to its block map.

The second Zebra-related issue in recovering the file manager is determining where to begin processing deltas once the virtual disk file has been recovered. The Zebra file manager solves this problem by storing additional information in its checkpoint file that identifies the last delta processed for each client prior to the checkpoint. Since the virtual disk file is forced to the log during a checkpoint it is guaranteed that any deltas processed prior to the checkpoint are safely reflected in the block maps, whereas any deltas processed after the checkpoint may not be. During recovery the file manager need only apply the latter type of deltas to the block maps to bring the block maps up-to-date.

There are several complications associated with replaying deltas during recovery, however. First, the effects of some of the deltas processed since the last checkpoint may already be reflected in the block maps, and therefore should not be reapplied. This will happen if the modified block maps were written out to the log before the crash. The update deltas that should not be reapplied are easily identified because their file version numbers will be less that the current file version numbers of the files to which they apply.

The second complication is determining the order in which the deltas should be replayed. During normal operation changes in the state of the client cache consistency dictate the order in which the deltas should be applied, as described in Section 4.4.2. For example, when one client opens a file that was just written by another client, the file manager knows that it must process the deltas from the latter client before the open by the former is allowed to complete. During recovery this dynamic cache consistency information is unavailable, so another mechanism must be used to order the deltas. Zebra solves this problem using the file version numbers stored in the deltas. If the file manager encounters an update delta whose version number is greater than the file's version number then there must be an update delta in another client's log that should be applied first. The file manager delays processing the delta until all the intervening update deltas have been processed from the other client logs. Deadlock is not possible because the delta version numbers reflect the order in which the deltas were created, so that newer deltas are applied after older deltas. For a circular dependency to occur a delta would have to exist that depends on a newer delta, as shown in Figure 4-4. Without a circular dependency, deadlock cannot occur, and the file manager is guaranteed to make progress during recovery.

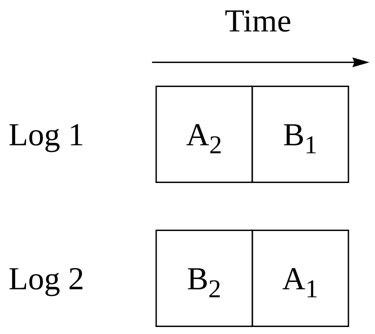

Figure 4-4. Requirements for deadlock.

Two client logs are shown, each containing one delta for two files. To process delta A_2 the file manager must first process A_1 and hence B_2 because it precedes it in the log. Processing B_2 in turn requires processing B_1, leading to a deadlock because A_2 must be processed first. Fortunately this scenario cannot occur, because it implies the deltas were created in the order B_1,B_2,A_1,A_2, yet B_1 occurs after A_2 in the log.

4.5 Stripe Cleaner

One of the implications of using an append-only log to store file data is that each client is continually creating new stripes to hold newly created portions of its log. This means that the system must continually reclaim the space occupied by old stripes, otherwise it will run out of space in which to store new stripes. Space occupied by an old stripe is reclaimed through a process called *cleaning*, in which the live data in the old stripe is copied to a new stripe, leaving the entire old stripe unused and its storage space available to hold a new stripe.

Stripe cleaning in Zebra is handled by a use-level process called the *stripe cleaner*, which is very similar in operation to the segment cleaner in a log-structured file system. It first identifies stripes with large amounts of free space, then it reads the remaining live blocks out of the stripes and writes them to a new stripe (by appending them to its client log), as shown in Figure 4-5. Once this has been done, the stripe cleaner uses the *delete fragment* operation to delete the fragments from the storage servers.

4.5.1 Identifying Stripe Contents

To clean a stripe the cleaner must know which data in the stripe are live, and which are no longer used. The cleaning algorithm also needs to know the age of the live data, to allow it to make an intelligent decision as to the order in which stripes should be cleaned. The cleaner obtains the information on stripe contents and age by processing the deltas in

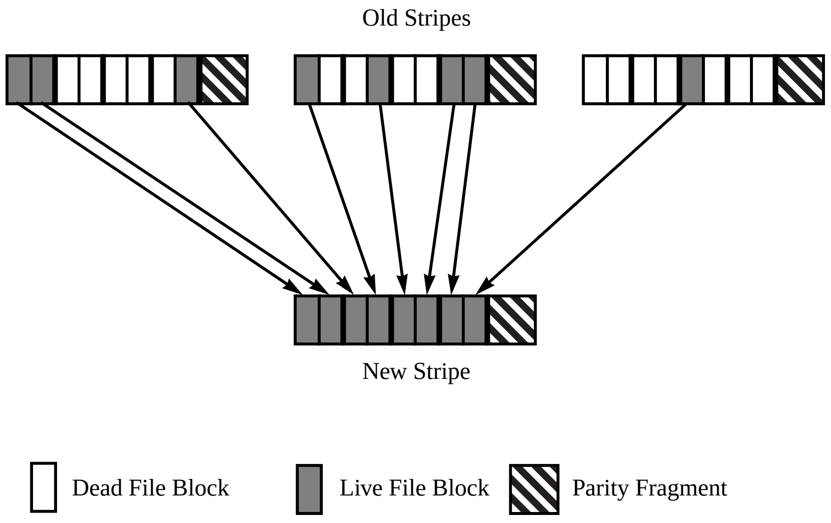

Figure 4-5. Stripe cleaning.

Live file blocks are copied out of old stripes into new, leaving the old stripes entirely empty and the space they occupy available for reuse. The parity of the new stripe is computed as the data are copied. In this example the live data in three stripes are copied to a new stripe, resulting in a net gain of two empty stripes.

the client logs, in much the same manner as the file manager updates the file system metadata by processing the deltas. The cleaner looks at each delta and updates the contents and age of the affected stripes accordingly: the data referred to by the old block pointer in a delta are no longer in use, while the data referred to by the new pointer are alive. The modification time in the delta indicates the time when the data were created, allowing the age of the data in the stripe to be computed. Note that the modification time in the delta indicates the time at which the corresponding block of data was created, not the stripe that holds it. It is possible for there to be large differences in the ages of the blocks in a stripe, particularly if some of the blocks were placed in the stripe due to cleaning since the cleaner favors cleaning old data.

In addition to using the deltas to adjust the stripe utilizations and ages, the cleaner also appends all of the deltas for a stripe to a per-stripe file, called the *stripe status file*, whose use will be described below. The stripe status files are stored as ordinary Zebra files. Note that a single delta can affect two different stripes; a copy of the delta is appended to the status files for both stripes.

The techniques used to make the stripe cleaner highly available and reliable are similar to those used by the file manager. The stripe cleaner's state consists of the stripe utilizations, stripe status files and the cleaner's progress in processing the client logs. This information is stored in normal Zebra files so that it is both reliable and available. Periodically the cleaner checkpoints these files to the servers to ensure that the copies in the log are consistent and can be recovered after a crash.

4.5.2 Choosing Stripes to Clean

The stripe cleaner chooses stripes to clean that minimize the amount of system resources consumed by cleaning. The first step in choosing stripes is to divide the stripes into three classes: those that cannot be cleaned, those that are trivial to clean because they contain no live data, and stripes that can be cleaned and contain live data. The first class of stripes are those than cannot be cleaned because they contain deltas that may yet be needed by the file manager or the stripe cleaner. This includes stripes whose deltas have not yet been processed by either the file manager and the stripe cleaner, and stripes whose deltas have been processed after the last checkpoint of each. Cleaning only applies to the blocks in a stripe; the deltas in a stripe are always discarded and never copied. This means that a stripe cannot be cleaned until there is no chance that the deltas it contains will be needed in the future. This can only happen if the deltas have been processed by both the file manager and stripe cleaner and checkpoints issued. Figure 4-6 illustrates the relationship of the file manager and stripe cleaner checkpoints and stripes that can be cleaned and those that cannot.

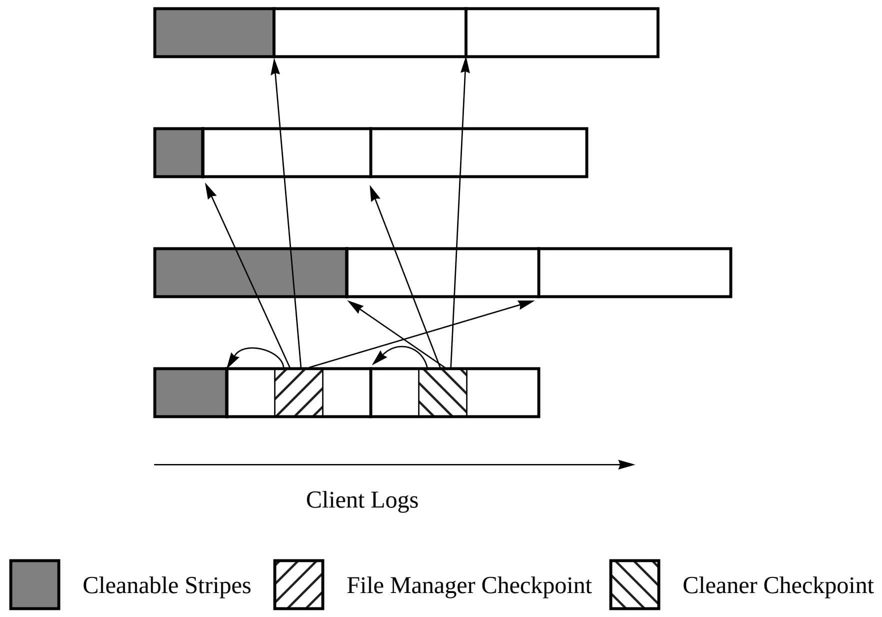

Figure 4-6. Cleanable vs. uncleanable stripes.
The cleaner can only clean stripes whose deltas have already been processed by both the file manager and stripe cleaner prior to their most recent checkpoints.

Once the cleaner has determined which stripes may be cleaned it first looks for stripes with no live data. The stripe utilization database makes this easy to do. If an empty stripe is found, and it is in the cleanable regions of the logs, the cleaner then deletes the stripe's fragments from the storage servers and also deletes the corresponding stripe status files.

This is a common occurrence since each cleaned stripe eventually becomes empty, hence this special check for empty stripes.

If there are no empty stripes and more free space is needed then the cleaner chooses one or more stripes to clean from the set of stripes that are cleanable and contain live data. The policy it uses for this is identical to the one described by Rosenblum [Rosenblum91], i.e., a cost-benefit analysis is done for each stripe, which considers both the amount of live data in the stripe and the age of the data. Each stripe in the system is given a priority defined by $\alpha(1-\mu)/\mu$, where α is the average age of the live bytes in the stripe, and μ is the utilization of the stripe (fraction of live bytes). The priority of a stripe is simply the benefit of cleaning it divided by the cost of doing so. Stripes that have a higher benefit-to-cost ratio have a higher priority, and are therefore cleaned first. The numerator in the equation is the benefit of cleaning the stripe. The amount of space reclaimed by cleaning the stripe, $(1-\mu)$, is multiplied by the average age of the live bytes. The intuition is that young bytes don't live long, so cleaning them will probably have little benefit since they are likely to die on their own soon anyway. This decreases the utilization of the stripe, and reduces the cost of eventually cleaning it. On the other hand, old bytes are unlikely to die in the near future, so that the stripe's utilization is not likely to decrease on its own. Thus there is no benefit in waiting to clean a stripe containing old data. The net result is that the benefit of cleaning a stripe is computed by multiplying the amount of free space in the stripe by its age, causing the cleaner to favor old stripes.

The cost of cleaning a stripe is in reading and rewriting the live data it contains. These reads and writes result in a transfer of 2μ bytes. The 2 is left out of the priority computation because it is a constant and only changes the absolute values of the stripe priorities and not their relative values.

4.5.3 Synchronized Cleaning

There are two steps in cleaning a stripe: identifying the live blocks, and then copying them to a new stripe. The stripe status files make the first step easy: the cleaner reads the deltas in the stripe's status file and finds blocks that were created but not yet deleted. Without the stripe status files this step would be much more difficult, since the deltas that cause blocks to become free could be spread throughout the stripes in the file system.

The second step in cleaning is copying the live blocks out of the stripe to a new stripe. This step is made more complicated by a potential race condition between cleaning a file block and modifying it. Without synchronization a client could modify the block after the cleaner reads the old copy but before the cleaner rewrites the block, in which case the new data might be lost in favor of the rewritten copy of the old data. There are two ways to avoid this race condition: locking the files to ensure exclusive access, as was done in Sprite LFS [Rosenblum91], and an optimistic approach pioneered by Seltzer et al. [Seltzer93] and used in Zebra.

The most straight-forward way of avoiding a cleaning race is to lock a file when it is being cleaned. The cleaner simply locks the file, cleans the desired blocks, and unlocks the file. By locking the file the cleaner ensures that a client cannot modify a block during cleaning.

While simple to implement, the locking approach causes lock convoys that reduce overall system performance. The cleaner in the original LFS used locking to prevent files from being modified until after cleaning was finished. Unfortunately, this produced lock convoys that effectively halted all normal file accesses during cleaning and resulted in significant pauses. Furthermore, the need for the cleaner to lock files when cleaning them results in additional messages to the file manager, reducing the performance of both the cleaner and the file manager.

4.5.4 Optimistic Cleaning

To avoid the performance problems associated with locking files to clean them the Zebra stripe cleaner uses an optimistic approach similar to that of Seltzer et al. [Seltzer93]. The idea behind optimistic cleaning is that blocks are cleaned without any synchronization with other applications. Applications may therefore cause a race by modifying a block at the same time it is cleaned. In the normal case no race will occur, since the cleaner favors old stripes and it is unlikely that the blocks they contain will be modified during the cleaning. In the unusual case that a race does occur the system detects the race and handles it by ignoring the new copy of the block produced by the cleaner. This does not mean, however, that a race prevents the cleaner from making progress. If the cleaner loses a race with an application it simply means that the application has produced a new version of the file block in question, and the new version is necessarily stored in a different stripe from the old version. Thus the old version of the block is no longer in use in the stripe being cleaned, just as the cleaner intended.

To clean a file block the cleaner brings the block into its file cache without opening the file and without changing the file's modification time. The file block is then marked as dirty so it will be appended to the end of its client log. In addition, a special type of delta is created for cleaned blocks called a *cleaner delta*, rather than an update delta. A cleaner delta differs from an update delta only in that it lacks a version number, since the cleaner does not know the current version numbers for the files it cleans because it doesn't participate in the cache consistency protocol.

In Zebra a race during cleaning is signified by two deltas for the affected block: a cleaner delta from the cleaner, and an update delta from the other client that modified the block. Both of these deltas have the same old block pointer because they refer to the same block in the log, as shown in Figure 4-7. The file manager detects a race by comparing the old block pointer in each delta with the block pointer in the file's block map. These block pointers will always agree unless a race has occurred, because the client cache consistency protocol prevents simultaneous modification of a file block by several clients.

There are four possible scenarios that can occur when the file manager processes a delta, as shown in Table 4-3. The first two scenarios represent the cases without a race: the delta's old block pointer matches the file manager's current block pointer, and the file

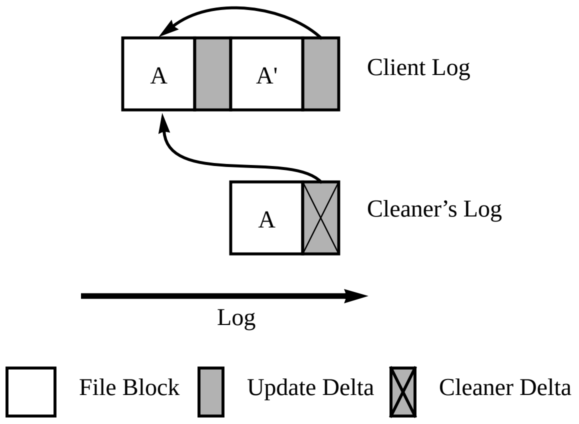

Figure 4-7. Cleaner/client conflict.

An update delta and cleaner delta have the same old block pointer, as shown by arrows. The client has modified block **A** to produce block **A'**. At the same time the cleaner has moved block **A** to a new stripe. The cleaner's copy of the block is outdated and is therefore rejected by the file manager.

manager updates its block pointer with the new block pointer in the delta. If an update delta arrives whose old block pointer doesn't match (the third scenario in the table), it indicates that a cleaning race occurred and the cleaner delta was processed by the file manager first. The file manager updates its block pointer with the new block pointer from the delta. If a cleaner delta arrives whose old block pointer doesn't match (fourth scenario), a cleaning race occurred but the update delta was processed first. In this situation the cleaner delta is ignored.

Type of Delta	Block Pointer Matches?	Update Pointer?	Issue Reject Delta?
Update	Yes	Yes	No
Cleaner	Yes	Yes	No
Update	No	Yes	Yes
Cleaner	No	No	Yes

Table 4-3. File manager delta processing.

When a delta arrives at the file manager, the old block pointer in the delta is compared with the current block pointer. If they do not match (the bottom two scenarios) then a conflict has occurred.

In both of the cases where the file manager detects a conflict it generates a *reject delta*, which is placed in the client log for its machine. The old block pointer in the reject delta refers to the cleaned copy of the block and the new pointer is null to indicate that this block is now free. The reject delta is used by the stripe cleaner to keep track of stripe usage; without it the stripe cleaner would have no way of knowing that the block it cleaned is was rendered obsolete by a concurrent modification, and is therefore no longer in use.

Another type of cleaning race is possible in which an application reads a block at the same time that it is being cleaned. For example, suppose the cleaner cleans a block after a client has obtained the block pointer, but before it has read the block. If the client then tries to use the out-of-date block pointer, one of two things will happen. If the block's stripe still exists then the client can use it safely, since the cleaner doesn't modify the old copy of the block. If the stripe has been deleted then the client will get an error from the storage server when it tries to read the old copy. This error indicates that the block pointer is out of date: the client simply discards the pointer and fetches an up-to-date version from the file manager, as it would if it didn't have a copy of the pointer in the first place.

4.5.5 File Manager Recovery Revisited

File manager recovery is made more complicated by the existence of cleaner deltas, since unlike update deltas they do not contain file version numbers. During recovery the file manager uses the version numbers in the update deltas to apply them to the block maps in the correct order. The cleaner, however, does not open a file when it cleans it, and as a result does not obtain the file version number returned in the reply to an open, and thus cannot store version numbers in the cleaner deltas it produces. Without the version numbers the file manager depends on the block pointers in the deltas to determine when to apply the cleaner deltas.

The block pointers in the deltas make it possible to order the deltas because the old block pointer in the next delta to be applied must match the new block pointer in the last delta applied, and hence the block map itself. If a delta's old block pointer does not match the block map then it is not the next delta to be applied. This means that if the old block pointer in the next update delta (as determined by the delta version numbers) does not match the block map an intervening cleaner delta must be applied first. Figure 4-8 shows an example of detecting that a cleaner delta should be applied before the next update delta.

The cleaner deltas encountered by the file manager during recovery fall into three classes: those that should be applied between two update deltas, those that should be applied after the last update delta for the block, and those that should not be applied at all because they conflict with an update delta due to a race between the cleaner and a client. The first type of cleaner deltas is applied during processing of the update deltas, as described in the previous paragraph. Once the file manager has completed processing the update deltas it will be left with those cleaner deltas that fall into the other two categories. The file manager compares the old block pointer in each delta with the corresponding pointer in the block maps; if the pointers match the delta is applied. The remaining cleaner deltas are rejected. The file manager keeps track of any reject deltas encountered while replaying the logs to ensure that duplicate reject deltas are not issued.

4.5.6 Storage Server Crashes

If the cleaner cleans a stripe while one of the storage servers is down it will be unable to delete the stripe's fragment on the unavailable server. When the server reboots it will

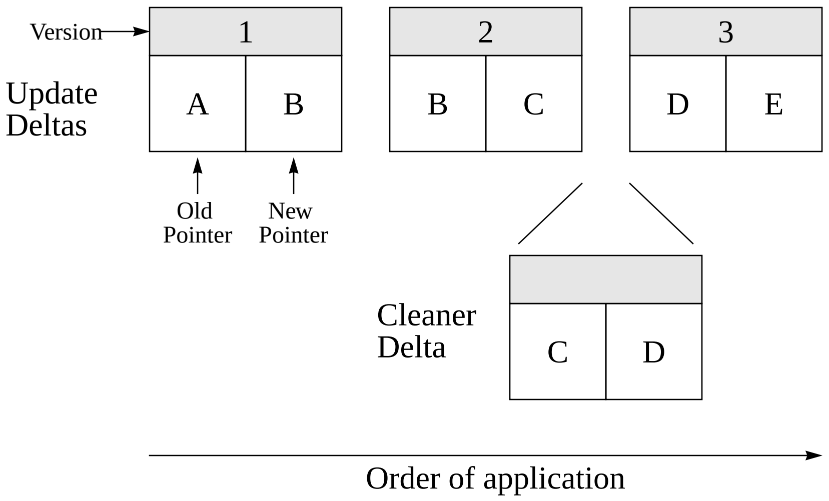

Figure 4-8. Ordering deltas by their block pointers.

The need to apply a cleaner delta is detected by a mismatch in the old and new block pointers in a sequence of update deltas. In this example the three update deltas are ordered by their version numbers. Delta 2 is applied after Delta 1 because its old block pointer B matches the new block pointer in Delta 1. The old block pointer D in Delta 3 does not match the new block pointer C in Delta 2, hence a cleaner delta must be applied whose old block pointer is C and whose new block pointer is D.

therefore contain stripe fragments for stripes that have been deleted. To avoid this waste of storage space the stripe cleaner keeps track of fragments for stripes that were cleaned but could not be deleted due to a server failure. When the server reboots the cleaner resends the delete requests to it so that the storage space occupied by the unneeded fragments can be reused.

4.5.7 Cleaning Cost

One concern about the stripe cleaner is how much of the system's resources it will consume. Cleaning cost is directly related to workload behavior: some workloads will have cleaning costs approaching zero, while in others cleaning may consume most of the system's resources. For example, a workload that creates and deletes large files that span many stripes will have a very low cleaning cost. Most stripes are either full or empty so that the cleaner does not have to copy anything to clean a stripe. While I have not measured Zebra's cleaning overhead under real workloads, it should be comparable to those for other log-structured file systems. In a synthetic transaction-processing benchmark on a nearly full disk, Seltzer found that cleaning accounted for 60-80% of all write traffic and significantly affected system throughput [Seltzer93]. However, Seltzer found cleaning costs to be negligible in a software development benchmark that is more typical of workstation workloads. Rosenblum measured production usage of LFS on Sprite for several months and found that only 2-7% of the data in stripes that were cleaned were live and needed to be copied [Rosenblum91]. Based on these measurements Zebra's cleaning costs should be low for the type of workstation workloads for which it was intended, but more work may be needed to reduce the cleaning overheads of transaction-processing workloads.

4.5.8 Distributed Stripe Cleaning

The Zebra stripe cleaner is a centralized resource, leading to concerns about its availability and performance. The former can be handled using the checkpoint and roll-forward mechanism previously described, but the latter is a more difficult problem to solve. The cleaner may consume only a small fraction of the system's resources, but as the size of the system scales the throughput requirements of the cleaner may exceed the capabilities of the machine on which it is running. If this is the case then the cleaner must be distributed across several machines. There are two options for doing so. In the first approach a set of slave cleaners run on several machines in the system, under the control of a master cleaner. The master simply processes the deltas, decides which stripes to clean, and directs the slaves to do so. Since the master doesn't actually process any data it can scale up to a much larger system. All that is needed to do so is more slave cleaners.

If the master/slave solution is unacceptable then a symmetric solution can be used. The set of stripes in the system is partitioned and a cleaner is assigned to clean each partition. For example, the stripes for each client can be numbered sequentially with each stripe assigned to a cleaner in a round-robin fashion. By adding more cleaners the total cleaning capacity of the system is scaled. The drawback of this approach is that the choice of which stripes to clean is no longer a global one. Each cleaner cleans the "best" stripes in its partition, but there is no guarantee that this set of stripes represents the best stripes overall. This results in increased cleaning costs, but I have not attempted to quantify this effect.

4.5.9 Stripe Cleaning Alternatives

There are several alternatives to having a global stripe cleaner, but all of them have serious drawbacks. The first is to eliminate cleaning altogether and simply "thread" new portions of the logs into unused pieces of previous stripes. As new portions of the logs are produced the existing stripes are examined to find empty spaces that can contain the new data. These spaces are guaranteed to exist because in the steady state empty space must appear in the file system at the same rate as which new data is produced. Threading the log avoids cleaning because stripes don't need to be entirely empty to be used to store new pieces of the log. There is a major problem with this approach, however, in that threading the log turns every log write into an expensive partial stripe write.

A second approach is to have the servers do the cleaning internally. The servers should always have enough free space to store new fragments, as long as the total amount of live data in the system is less than the servers' total storage capacity. This space may not be contiguous, however, leading to performance problems when trying to use it. The server can rectify this problem by cleaning, or garbage collecting, its storage space to make the free space contiguous, allowing new fragments to be written in efficient transfers. There are two problems with this approach, however. The first is that it requires the servers to keep track of the live data that they are storing. This means that they will have to process the deltas, increasing their complexity. The bigger problem is that a server cannot decide to reuse storage space simply because the data it contains is no longer in use. Reconstructing a stripe fragment requires all of the data in the other fragments in a stripe, regardless of whether or not that data is still in use or not. If a server discards a block of data from a stripe then the corresponding blocks in the other fragments cannot be reconstructed.

4.5.10 Log Address Wrap

One of the problems with the log addressing scheme used in Zebra is that log addresses represent offsets within client logs, and it is possible for a log to grow too large for the addresses. If a client writes enough data to its log the offset can grow too large for the fixed-size log address to represent. When the offset overflows the client will begin generating log addresses that were previously used and which may conflict with the addresses of existing stripes. The solution used by Zebra is to clean stripes in the logs prior to their reuse by the clients to avoid having the clients generate log addresses that are already in use.

The disadvantage of using cleaning to solve the log address wrap problem is that it consumes system resources because the stripe cleaner must continually copy live data to the end of the logs. The actual cost of this copying is determined by the ratio of the size of the log address space to the amount of live data in the log, since each time the log address wraps the entire contents of the affected log must be copied to its end. In other words, as a client writes to its log the cleaner must clean any live data that is already stored in the log, so that the total amount of data transferred is proportional to the amount of data written by the client (the size of the log address space), plus the amount of live data in the log. As a result, if the size of log address space is close to the amount of live data in the log the overhead of cleaning will be high, whereas an address space that is much larger than the amount of live data will have a correspondingly lower overhead. As an example, in the Zebra prototype the log address space for each client is represented by 39 bits, so that a client can write up to 8 terabytes of data before its log addresses wrap. If the size of the storage servers is 8 gigabytes, then the write overhead of cleaning to avoid log address wrap is at most 1/1000, or one-tenth of one percent of the total system write bandwidth.

An alternative to this solution is to “thread” new stripes of the log around existing stripes. In this scheme the addresses assigned to stripes do not represent offsets in the log, but instead are unique IDs that do not belong to existing stripes. This approach avoids having two stripes with the same address, but it complicates several aspects of the log mechanism. First, the ordering of stripes within the log cannot be determined from their addresses, as can be done in Zebra. Additional information is needed for each stripe that identifies the stripes that precede and succeed it in the log. Second, clients must somehow obtain unique stripe IDs to use for newly created stripes. In Zebra unique stripe IDs are easily generated by incrementing the log offset, but in this scheme clients must synchronize to ensure that existing stripe IDs are not reused nor do clients assign the same stripe ID to more than one stripe.

4.6 System Reconfiguration

This section covers changes in the configuration of a Zebra system, such as adding and removing clients, storage servers, and disks. The file manager and stripe cleaner are not covered, since a standard Zebra system will only have one of each.

4.6.1 Adding and Removing Clients

Adding a client to a Zebra system is a two-step process. First the client must be assigned a unique client ID that distinguishes the new client's log from the other logs in the system. The file manager is then notified of the new client and its ID. The file manager will then include the client in the cache consistency protocol and the client can begin accessing files.

Removing a client is done by having the client close all of its open files and flush any dirty blocks out of its cache. This cleans up its cache consistency state on the server, following which the client can then be removed from the system and its client ID reused.

It is possible treat a client crash followed by a reboot as a removal from the system followed by an addition to the system. The reasons for not doing so are purely administrative; for example, it may simplify system administration if there is a fixed pool of clients, each with a fixed client ID.

4.6.2 Adding Storage Servers

Zebra's architecture makes it easy to add a new storage server to an existing system. All that needs to be done is to initialize the new server's disk(s) to an empty state and notify the clients, file manager, and stripe cleaner that each stripe now has one more fragment. From this time on clients will stripe their logs across the new server. The existing stripes can be used as-is even though they don't cover all of the servers; in the few places where the system needs to know how many fragments there are in a stripe (such as reconstruction after a server failure), it can detect the absence of a fragment for a stripe on the new server and adjust itself accordingly. Over time the old stripes will gradually be cleaned, at which point their disk space will be used for longer stripes that span all of the servers. Old stripes are likely to be cleaned before new ones since they contain fewer live data. If it should become desirable for a particular file to be reallocated immediately to use the additional bandwidth of the new server, this can be done by copying the file and replacing the original with the copy.

4.6.3 Removing a Storage Server

Removing a storage server from operation is a three-step process. First, it must be verified that there is enough free space in the system to accommodate the loss of a server. If this is not the case then files must be deleted until the total amount of free space exceeds the storage capacity of the server. Second, the clients, file manager, and stripe cleaner are notified that stripes now have one less fragment. Once this is done any new stripes created will not use the server that is being decommissioned. Third, the stripe cleaner is instructed to clean all the old stripes. This has the effect of moving live data from the unwanted server to the remaining servers. Once this is accomplished the unwanted server will not contain any live data and can be safely removed from the system.

4.6.4 Adding Disks to Servers

The system’s storage capacity can also be increased by adding disks to the existing servers. This is easily done: the disks are initialized and the stripe cleaner notified that there are now more empty stripes in the system. There is one caveat, however. The usable capacity of the system is only increased if the disks are added to the servers in a balanced fashion. Each fragment that a server stores must be backed up by a parity fragment on another server in the system. Thus the total number of stripes that the system can store can be no greater than the number of fragments on the smallest server. It may be beneficial to add a disk to a single server since it might make it easier for that server to allocate storage for stripe fragments, but it will not increase the total number of stripes that can be stored. The total number of stripes in the system can only be increased by adding disks to the servers in a balanced fashion.

4.6.5 Removing Disks from Servers

Removing disks from servers is a three-step operation. First, there must exist enough free space in the system to tolerate the loss of the disks. If this is not the case then files must be deleted until the total amount of free space exceeds the capacity of the disks to be removed. Once this is done the stripe cleaner is instructed to begin cleaning until the total number of free stripes in the system exceeds the capacity of the disks. Third, each storage server is instructed to move any fragments stored on the unwanted disks to its other disks. When this is completed the unwanted disks can be removed from the servers. As an optimization of the last step the storage servers can be notified in the second step to avoid using the unwanted disks to store new fragments. This reduces the number of new stripe fragments that will be stored on those disks and therefore have to be moved by the servers later, but it isn’t necessary for the correct operation of the system.

4.7 Summary

The Zebra architecture defines the framework for a striped network file system that provides scalable performance and is highly available. The Zebra architecture is centered around the use of log-based striping to store file data. Each client creates a log from the data it wishes to write, and stripes the log across the storage servers. As the log is striped, the client computes and stores its parity as well. The simplest component of the system is the storage servers, which are no more than repositories for stripe fragments. The servers store the fragments written by clients, and make them available for subsequent read accesses. Should a storage server fail, the clients simply reconstruct the fragments it stores by reading the remaining fragments in the same stripe and XORing their contents together, producing the missing fragment. When the failed storage server reboots, it brings itself up-to-date by reconstructing and storing the appropriate fragments of any stripes created while it was down.

Log-based striping only provides a mechanism for storing file data on the servers; it does not define a way of keeping track of the log contents, or of the space allocation on the servers. To handle these two necessary functions the Zebra architecture uses two centralized resources, the file manager and the stripe cleaner. The file manager is responsible for maintaining the metadata of the file system, including the block maps for the files. Clients use the block maps to read blocks from files. If a client wishes to read a file block whose log address it doesn't know, it simply fetches the block pointer from the file manager.

The stripe cleaner's duty is to manage the free space on the storage servers. It does this by cleaning old stripes, so that each cleaned stripe does not contain any live data and the storage space it consumes can be reused for a new stripe.

Since the file manager and stripe cleaner are centralized resources, they represent potential single points of failure. Zebra uses two techniques for avoiding this problem. First, both the file manager and the stripe cleaner store their state information in the log of the clients on which they are running. For the file manager this state information consists of the file system metadata, and for the stripe cleaner it consists of the information about stripe contents. This allows the file manager and stripe cleaner to run on any client in the system, since the logs can be accessed by any client. Second, both the file manager and the stripe cleaner periodically checkpoint their state to the log, so that the contents of the log are consistent. After a crash, they look backwards through the log until the most recent checkpoint is found, allowing them to initialize their states and continue operation.

One of the most novel aspects of Zebra is that the mechanism used to store file data is also used as a reliable communication channel over which changes in the state of the system are transmitted. When a client stores a file block in its log, it also stores a delta for that block. The delta identifies the block, and describes both the block's old and new storage locations. The file manager and stripe cleaner processes the deltas in the client logs and use them to update their states. Since the logs are reliable and time-ordered, it is easy for the different Zebra components to agree on the distributed state of the system.

5 Zebra Prototype

The previous chapter provided an overview of the Zebra architecture, but left out the details that must be addressed in an actual implementation. This chapter fills these in by describing the implementation of a Zebra prototype in the Sprite operating system. Much of the complexity in implementing the prototype came from integrating the Zebra functionality with the existing operating system, and since this will vary from operating system to operating system, those details have been omitted. Instead, the inherent complexities in building the Zebra components are described.

One caveat of the prototype is that it does not implement all of the functionality described in the last chapter; in particular some of the availability and reliability mechanisms either require manual intervention to operate, or are not implemented at all. To reduce the overall implementation effort, I did not implement those features that are not necessary to demonstrate the scalability and availability advantages of the Zebra architecture. For example, the prototype does not behave correctly when it runs out of storage space, nor do clients automatically reconstruct fragments when a server crashes. None of the limitations of the prototype affects the validity of the performance measurements, however, so I did not implement them. I do, however, propose implementations for those features the prototype lacks.

The description of the prototype is organized into eight sections, each of which deals with a different function within the prototype. These functions are log addressing, fragment creation, data transfer, metadata management, client caching, stripe cleaning, and fragment storage. Each section describes how the system components cooperate to provide the functionality in question, and when appropriate, how the system continues to provide the functionality despite a component failure. Each section also describes in what ways, if any, the prototype implementation is inefficient or inadequate, and proposes better ways to implement the same functionality.

5.1 Log Address Format

In Zebra, file blocks are accessed by their log addresses, i.e. their offsets within the client logs that store them. Section 4.1 provided an overview of how a log address is broken-down into a log ID, a stripe index, a fragment index, and a fragment offset, but it did not specify the sizes of these components, since such information is not important when describing the architecture. The size and structure of a log address are important, however, when implementing the architecture, since they affect the size of each file's block map, as well as the interface between the clients and the storage servers. A small log address reduces the overhead of the block maps, but increases the rate at which the log addresses wrap. A large log address has the opposite effect.

The prototype strikes a balance between these two considerations by storing log addresses as two 32-bit words. The first word of a log address is called the *fragment ID*, and it consists of the log ID, stripe index, and fragment index portion of the log address. The second word of the log address is simply the offset into the fragment. Figure 5-1 shows the internal structure of a fragment ID. The log ID is eight bits long, allowing for up to 256 clients in a system. Four bits are allocated for the fragment index, allowing up to 16 fragments in a stripe, and therefore 16 storage servers in a system. The remainder of the bits (20 in all) are the stripe index.

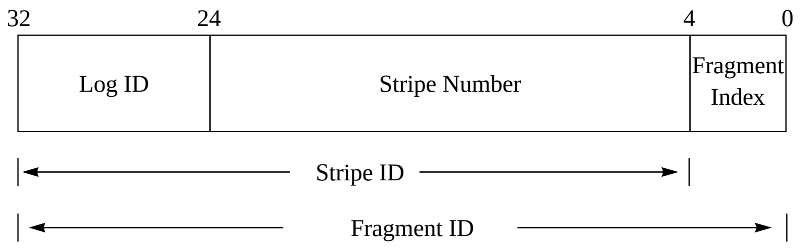

Figure 5-1. Fragment ID.

A stripe fragment ID is composed of a stripe ID plus a fragment index within the stripe. A stripe ID consists of the client ID for the client that created the stripe, plus a sequence number for the stripe that is incremented for each stripe the client writes.

The advantage of this log address format is that it allows a fragment to be uniquely identified by the single word that is its fragment ID. This simplifies the system design because a single-word fragment ID is simpler to manipulate and store than one that is longer. There are several disadvantages, however. First, each client can only create 2^{24} fragments before running out of unique fragment IDs. The fragment size in the prototype is 512 Kbytes, allowing each client to write at most 8 Tbytes of data before its log addresses wrap. The second disadvantage is that a 32-bit fragment offset is larger than is needed for any realistic fragment size. The prototype only uses 19 bits of the offset, leaving the remaining 13 bits unused. Furthermore, there is little advantage to making the contents of the log byte-addressable. If file blocks were aligned on 512-byte boundaries in the log, for example, the fragment offset would only need to be 10 bits. The underlying problem with the log address implementation in the prototype is that while it simplifies the implementation to break a 64-bit log address into a 32-bit fragment ID and a 32-bit offset, it does not make efficient use of the bits.

5.2 Fragment Creation

This section describes the format of the stripe fragments in the prototype, and how they are created by the clients.

5.2.1 Fragment Format

A stripe fragment containing file data (a *data fragment*) consists of one or more fragment *portions*, as shown in Figure 5-2. Each portion of a fragment consists of the blocks and deltas added to the fragment by a single store or append operation. Instead of storing the deltas next to the blocks they describe, the prototype stores file blocks at the beginning of a fragment portion, followed by a *delta region* that contains the deltas that describe the blocks, and concluded by a *delta trailer* that describes the delta region. The portions of a fragment are linked together by a pointer in each delta trailer that points to the delta trailer of the preceding portion of the fragment. This allows all of the delta regions in a fragment to be accessed by following the chain of pointers that starts at the delta trailer at the end of the fragment.

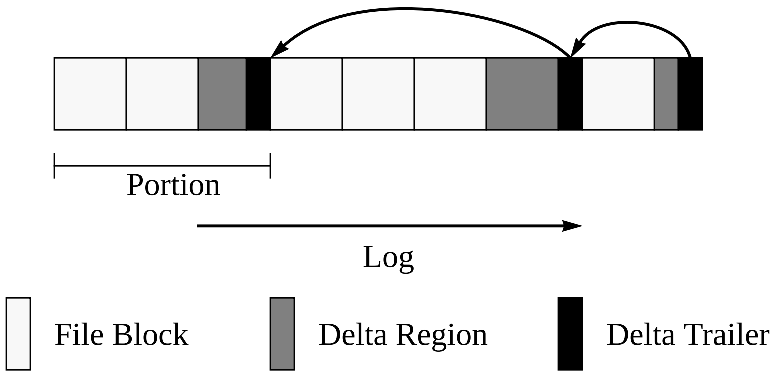

Figure 5-2. Data fragment format.

Each data fragment is divided into portions, and each portion is divided into file blocks, a delta region containing the corresponding deltas, and a delta trailer that describes the delta region and points to the previous delta trailer in the fragment.

This particular data fragment format was chosen because it allows several deltas to be read in a single operation, and it allows the log address of a file block to be determined at the time the block is assigned to a log fragment. First, the delta regions make it easy to find and read the deltas in a fragment. All of the deltas in a region can be read in a single operation, and the pointer in the delta trailer makes it easy to find the next delta region in the fragment. If each delta were stored next to the block it describes, one read operation would be required to read each delta, significantly reducing the performance of reading deltas from the log.

The second advantage of the current fragment format is that by storing the file blocks at the beginning of each portion, and the delta region and delta trailer at the end, the log address of a file block can be assigned at the time the block is placed into the fragment, rather than waiting until the fragment portion is complete. Delta regions are not a fixed size, hence putting the delta region at the beginning of a fragment portion causes the starting offset of the file blocks to be dependent on the size of the delta region, which in turn is dependent on the number of file blocks in the fragment portion. Thus the log address for each file block can not be assigned until the delta region is complete. By putting the delta region after the file blocks, however, the log addresses can be immediately assigned.

In retrospect, this advantage of the prototype's fragment format is a minor one. The intent was to allow the software that placed a file block into the log to immediately determine the block's log address, so that the file's block map could be updated accordingly. Otherwise the software would have to be notified when the fragment portion was complete, complicating the structure of the software. In reality, however, the upper-levels of the software need to be notified when a block has been stored anyway, so that the state associated with the block can be updated to reflect this fact. For example, the file cache needs to know when a dirty block has been stored so that it can unlock the block and mark it as clean. There is no reason why the routine that performs these operations cannot also update the file's block map.

There is also no compelling reason why delta regions end with a delta trailer instead of beginning with a header. The original intent was to ensure that each client log always ends with a delta trailer, making it easy to find the deltas in a fragment. To find the deltas in a client log one only needs to read the trailer at its end, then chain backwards through the delta regions using the pointers in the trailers. In retrospect it would be just as simple to start each fragment with a header that points to the first delta region, and start each region with a header that points to the next region in the fragment, thus chaining the regions in the forward direction rather than the reverse.

5.2.2 Zebra Striper

Fragments are created on the clients by a software module called the *striper*. The striper consists of a front-end and a back-end: the front-end provides an interface by which higher-level software, such as the client file cache and the virtual disk file, can store data and deltas in the log, while the back-end takes the fragments created by the front-end and stripes them across the storage servers. This section describes how data fragments are filled, how parity fragments are created, how the front-end coordinates its operation with the back-end, and how higher-level software interfaces with the striper.

5.2.2.1 Filling Data Fragments

The front-end of the striper creates data fragments from the file blocks and deltas it is given by higher-level software, such as the file cache. File blocks are located at the beginning of a fragment, allowing the striper to return the log address of a block it is given. The striper simply keeps track of how much file data the current fragment contains, and places each new file block after any existing blocks. Deltas, on the other hand, are not immediately placed in the fragment, but instead are buffered to be placed in the delta region. The striper keeps track of the size of the deltas, and uses the amount of file data and size of the deltas to determine when the current fragment is full. Once the fragment is full the delta region and delta trailer are placed after the file blocks, and the fragment given to the back-end to be stored.

Sometimes it is necessary to write a fragment to the storage servers before it is full. If, for example, an application invokes `fsync` to force a file's dirty blocks to stable storage, the striper has no choice but to write the current fragment, even if it is not yet full. This complicates the fragment format and the striper implementation because it means that fragments can have multiple delta regions and can be written in multiple operations. To write a non-full fragment the striper places the delta region and delta trailer after the fragment's file blocks, and gives the fragment to the back-end to be written to the storage servers. The difference between writing a full fragment and one that isn't is that in the latter situation the striper does not begin filling a new fragment. Instead, it continues to place new file blocks and deltas in the current fragment until it is full. The new file blocks and deltas are placed in the fragment after the data already written. Furthermore, subsequent writes of fragment use append operations rather than store operations.

One of the biggest issues in the design of the striper is its performance. All data written by a client must pass through the striper, so that the striper is a potential performance bottleneck. For this reason, the striper does not make a copy of the file blocks that it handles, but instead simply manipulates pointers to the blocks. The simplest way of filling a fragment is to allocate a buffer to hold the fragment, and copy file blocks and deltas into the buffer. Unfortunately, this solution requires the striper to copy all of the file blocks written. To avoid this bottleneck, the striper in the prototype simply keeps pointers to the data blocks in a fragment, rather than copying their contents, and thus reduces the load on the client.

While the elimination of copies within the striper improves performance, it complicates the interface between the striper and the higher-level software. Since the striper does not make a copy of the blocks it is given, the higher-level software must not modify any blocks it gives to the striper until the striper is done with them. The prototype implements this synchronization using *callbacks*. For each block given to the striper, the higher-level software also provides a callback routine to be invoked when the striper is done with the block. The higher-level software must not modify a block until its callback has been invoked. For example, the callback for a file block written by the file cache marks the block as clean and unlocks it, so that applications can resume accessing and modifying the block.

The striper handles deltas differently from file blocks, since the cost of copying a delta is small relative to the cost of issuing a callback. Thus when the striper is given a delta to store it simply makes a copy of the delta, instead of using the pointer/callback mechanism used for file blocks.

5.2.2.2 Parity Fragments

Parity fragments are fragments that contain the parity of a stripe, rather than file blocks and deltas, and as such are handled differently by the striper. When the striper begins filling a new stripe it allocates a parity fragment for the stripe, and allocates a buffer to hold the contents of the parity fragment. The contents of the parity buffer are initialized to zero. The parity buffer is filled in during the process of sending the data fragments to the storage servers, as described in the next section on data transfer. Once all of the data fragments in the stripe have been sent to the storage servers the parity fragment is given to the back-end to be sent as well. After the parity fragment has been stored its buffer is deleted.

5.2.2.3 Flow-Control

The front and back ends of the striper represent a producer/consumer pair: the front-end produces fragments, and the back-end sends them. Like all producer/consumer pairs, there needs to be flow control between the two, to prevent the front-end from producing fragments faster than the back-end can send them. Without flow control, fragments will accumulate in the back-end until the system resources are exhausted. To prevent this the front-end is allowed to get at most one stripe ahead of the back-end. In effect, there is a queue of stripes between the front-end and the back-end, and the queue has a maximum size of one. Once the queue is full, the front-end is prevented from filling any new fragments, and any requests by the higher-level software to place data or deltas in the log are blocked until the queue is emptied.

5.2.2.4 Striper Interface

The striper provides an interface that allows higher levels of the client kernel to store data and deltas in the log and receive notification once they have been stored. The routines provided by the striper are shown in Figure 5-3. Data blocks are placed into the client log via the `StriperPlace` routine. The *dataPtr* and *length* parameters describe the data being placed. The state of the striper is maintained in the structure pointed to by *striperPtr*, allowing a client to write to several logs, as would happen if a client uses several Zebra file systems. The *deltaPtr* parameter points to a partially completed delta for the data block. All of the fields of the delta (described in Section 4.4.2) are filled in by the routine calling the striper, except for the new block pointer which will be filled in by the striper when it places the block into the log. The delta is then stored in the log along with the data.

Callbacks are implemented using the *doneProc* and *doneData* parameters to `StriperPlace`. *DoneProc* points to the callback routine to be invoked. If *doneProc* is NULL a callback is not done when the data block has been stored. *DoneData* is a single word in length and is passed to the callback routine without being interpreted by the striper, allowing the higher-level software to pass information between the routine that called `StriperPlace` and the callback routine. For example, the file cache passes information about the cache block being written in the *doneData* parameter.

```
StriperPlace(striperPtr, length, dataPtr, deltaPtr, doneData,
   doneProc)
   StriperState   *striperPtr;   /* State of the striper. */
   int            length;        /* Size of data. */
   Address        dataPtr;       /* Data buffer. */
   Delta          *deltaPtr;     /* Delta info for data. */
   ClientData     doneData;      /* Data for callback. */
   void           (*doneProc)(); /* Callback routine. */

StriperDelta(striperPtr, deltaPtr)
   StriperState   *striperPtr;   /* State of the striper. */
   Delta          *deltaPtr;     /* Delta to be stored. */

StriperCallback(striperPtr, doneData, doneProc)
   StriperState   *striperPtr;   /* State of the striper. */
   ClientData     doneData;      /* Data for callback. */
   void           (*doneProc)(); /* Callback routine. */

StriperFlush(striperPtr, parity)
   StriperState   *striperPtr;   /* State of the striper. */
   Boolean        parity;        /* Flush parity too? */
```

Figure 5-3. Striper routines.

These procedures are provided by the striper module to the higher-level software. `StriperPlace` is used to store data in the log, and `StriperDeltas` is used to store a delta. `StriperCallback` registers a callback routine called when the log up to the current position has been stored. `StriperFlush` flushes any fragments to the storage servers.

In addition to the *doneData* parameter, the *doneProc* routine is also passed other information about the data block that was stored, as shown in Figure 5-4. These parameters are the status of the store operation, and the fragID and offset that represent the log address at which the block was stored.

```
doneProc(doneData, status, fragID, offset)
   ClientData     doneData;   /* Callback data. */
   ReturnStatus   status;     /* Status of the store. */
   int            fragID;     /* Data’s fragment ID. */
   int            offset;     /* Offset in fragment. */
```

Figure 5-4. Striper callback.

A callback is registered by `StriperPlace` and `StriperCallback`. The callback is associated with the current position in the log and is invoked when the log is stored up to that position.

The remaining routines provided by the striper module are `StriperDelta`, `StriperCallback`, and `StriperFlush`. `StriperDelta` is used by the higher-level software to store deltas in the log that don’t have a corresponding data block. Two examples of when this is necessary are when a file block is deleted, and when a reject delta is issued for a cleaner delta. `StriperDelta` provides an interface for these deltas to be stored in the log. The striper makes a copy of the delta, as described previously, so there is no need for a callback routine.

The `StriperCallback` routine registers a callback to be invoked when all of the data blocks and deltas that have been placed in the client log have been stored. This can be used, for example, to determine when a delta placed in the log via `StriperDelta` has actually been stored.

The `StriperFlush` routine flushes any unwritten data fragments to the storage servers, and is used to implement the `fsync` system call. If the *parity* parameter is non-zero then the parity fragment for the last stripe is written as well, even if the stripe is not full. Otherwise the last parity fragment is not written. `StriperFlush` is synchronous, and does not return until all of the fragments have been safely written.

5.3 Data Transfer

One of the goals of the Zebra project is to provide high-performance file service that scales with the number of servers in the system. To achieve this goal, not only must clients use the servers efficiently, but they must use them in parallel so that the transfer bandwidth scales. This section describes how data are transferred between the clients and the servers in the prototype, allowing data transfer to be both efficient and scalable.

5.3.1 Remote Procedure Calls

The clients and servers in the Zebra prototype communicate via Sprite remote procedure calls (RPC) [Welch86]. A remote procedure call emulates a procedure call over the network. The client sends parameters to the server, and the server sends a reply. Sending an RPC is a synchronous operation, so that the process that initiates an RPC does not continue processing until the reply to the RPC has been received. The RPC system in the Sprite kernel handles the details of the RPC implementation, including packaging the RPC data into network packets, sending the packets across the network, resending lost packets, and reassembling the RPC data on the receiving machine. The Zebra prototype makes use of this existing communication infrastructure to transfer data between clients and servers. New RPCs were added to read and write Zebra fragments, thus a client can write a fragment to a server simply by invoking the RPC that sends a fragment. The underlying Sprite RPC system takes care of getting the data to the server.

There are several complications related to using RPCs to transfer fragment data, however. The first is that the Sprite RPC system limits the maximum size of an RPC to 16 Kbytes, which is significantly smaller than the 512-Kbyte fragments used in the prototype. One option is to increase the maximum RPC size, but there are overheads in the RPC system that make this impractical, including the need to allocate buffers based upon the maximum size of an RPC. Instead, the prototype uses a sequence of RPCs to transfer file data that are too large to fit in a single RPC.

The second complication is that RPCs are intended for a request/response style of communication, and therefore lack a flow control mechanism that is needed to make data transfers efficient. During an RPC the client sends a request, and the server processes it and sends a reply. This means that the server is idle while the client sends the request, and the client is idle while the server processes the request and sends the reply. Flow control is achieved by having only one outstanding request at a time, but it limits the sum of the client and server utilizations during an RPC to at most 100%, since the client and server computations do not overlap. Higher utilization, and therefore higher performance, can only be achieved by overlapping RPCs, so that the client sends a new request as soon as it has finished sending the previous one. This enables both the client and server to remain busy processing RPCs, and improves the resulting transfer bandwidth.

The use of overlapping RPCs is only a partial solution, however, because it lacks a flow control mechanism. As described, the client simply sends a new block of data once the server has received the previous one. Without flow control, it is possible for the client to send data faster than the server can process them, causing unprocessed data blocks to accumulate on the server. Eventually the server will run out of space for new blocks, and will discard any new blocks it receives. This in turn causes the client to retransmit the discarded blocks when it fails to receive a reply for those RPCs, increasing the load on the client and the network. In effect, this causes the timeout/retransmit mechanism in the RPC system to be used for flow control, rather than its intended role masking packet loss due to network errors.

The lack of a flow control mechanism in the RPC system forces the Zebra prototype to implement its own forms of flow control. The prototype uses two different techniques, one for writing and one for reading. First, the striper limits the number of outstanding write requests there can be to any one server, so that the client does not overrun the server. In the prototype this limit is two, allowing a server to store a fragment to its disks while at the same time receiving the next fragment from the client. This solution is very simple but is of limited value, since it doesn't prevent multiple clients from overloading the same server. Fundamentally, flow control must be driven by the sink of the data, a fact that does not fit well with the use of RPCs to write data to the servers since the servers do not initiate the RPCs.

Flow control for fragment reads is easier to handle because the client is both the sink of the data and the initiator of the RPC. Clients can regulate the rate at which they receive data from the servers by limiting the number of their outstanding read requests. A server only sends data to a client in response to a read request, and as long as a client does not ask for too many blocks simultaneously it will not be overrun.

The bottom line is that flow control is necessary when transferring large amounts of data, but it is difficult to achieve using an RPC system. The prototype's flow control mechanisms are weak at best; I would expect a production version of Zebra to use a stream-based protocol for transferring file data that implements a full-fledged flow control mechanism.

5.3.2 Asynchronous RPC

A problem of using an RPC system to transfer data in Zebra is that the system requires clients to transfer data to multiple servers simultaneously. The file transfer bandwidth can only scale with the number of servers if they are used in parallel. Concurrency is difficult to achieve, however, using the normal RPC abstraction. When a process invokes an RPC, it is blocked until a reply is received. Thus a single process can only use one server at a time, defeating the purpose of striping data across servers.

The standard solution to providing concurrency in RPC-based systems is the use of multiple processes. To transfer a fragment to several servers simultaneously, the client would use one process per server, each issuing RPCs to its associated server. The problem with this approach is that it requires the system to context-switch between the processes that are transferring the data. In the best case, one context-switch per RPC is required, and in the worst-case a context-switch will be required for each network packet sent or received. Furthermore, the collection of processes must synchronize their actions, increasing the data transfer overhead on the client and reducing the transfer bandwidth.

The Zebra prototype reduces the data transfer context-switch and synchronization overheads through the use of a communication abstraction called *asynchronous RPC* (ARPC). An ARPC is similar to an RPC, except that the process is not blocked until a reply is received. Instead, the process can continue execution while the RPC is being sent and the reply received. This allows a single process to handle multiple data transfers simultaneously, without requiring context switches to do so.

While the ARPC protocol does allow a process to continue processing while waiting for a reply, it is more difficult in an ARPC system for the process to determine when the reply has been received. In a standard RPC this synchronization is easy because the process is blocked until the reply arrives. The synchronization is more complicated using ARPCs because the process does not block. In the prototype’s ARPC system this synchronization is implemented using callbacks. When a process initiates an ARPC, it specifies a callback routine to be invoked when the reply is received, thus allowing the process to synchronize with the reply. In the striper, for example, fragments are sent to the servers using ARPCs, and the callbacks are used to notify the striper when the fragments have been stored. The striper, in turn, invokes the callbacks associated with the file blocks stored in the fragment.

The prototype uses ARPC for two purposes. First, it is used to overlap requests to the same server, as described in the previous section. This provides higher data bandwidth between a client and a server than can be achieved through a series of synchronous RPCs. Second, it is used to allow a single process to send and receive fragments from several storage servers simultaneously. This allows the transfer performance to scale with the number of servers in the system, without requiring one process per server on the clients and the corresponding context switches among them.

5.3.3 Integrated Parity and Checksum Computations

The Zebra architecture requires the clients to compute the checksums and parity of each fragment written to the servers. These computations can potentially cause a significant amount of overhead on the clients, because they require that each word of data in a fragment be operated upon by the processor. In a simple implementation the checksum and parity operations would be performed separately. This requires the processor to read each word of the data fragment from memory to compute the checksum, then read each word of both the data fragment and the corresponding parity fragment to compute the new parity. Each word of the new parity fragment must also be written back to memory. Thus, the checksum and parity computations on a fragment containing N words of data require a total of 3N words of data to be read into the processor, and N words of data to be written.

The Zebra prototype reduces the checksum and parity costs by integrating these computations into the RPC and network systems. At the lowest-level of the network system each word of a packet being sent must be copied from the host memory onto the network interface board, because the network interfaces used in the prototype cannot access the host memory directly. By integrating the checksum and parity computations into this copy operation it is possible to eliminate extra reads of the data fragment to compute the checksum or parity, so that the only additional memory accesses are the N reads and N writes needed to read and write the parity fragment. Thus the number of words that must be read is reduced from 3N to N.

Although integrating the parity and checksum computations with sending of the fragment data reduced the memory bandwidth consumed, it did require significant modifications to the existing Sprite network and RPC systems. The biggest complication is that various RPC and packet headers are added to the data at different levels in the communication system. Since the parity and checksums are computed at the lowest level, during the copy to the network device, these computations will erroneously include the headers. Rather than add a mechanism by which the headers could be excluded from the computations, it was simpler to include the headers in the computations but negate their effect afterwards. This is relatively easy to do because both the parity and checksum computations are a simple XOR, so that if extra data is included in the XOR in the form of a header, it can be removed from the final result by XORing it in again. Thus each level that adds a header to the data removes the effect of the header after the data has been sent by XORing the header into the parity and checksum results returned by the lower level. The only detail is that the header changes the offset of the data within the parity buffer. When a network packet is sent that contains a piece of a fragment, it is XORed into the parity fragment at a particular offset. If a header is added to the data prior to the XOR, the offset must be adjusted accordingly.

5.4 File System Metadata

As described in Section 4.4, the metadata of a Zebra file system is similar to that of a UNIX file system, except that the block pointers in a file’s block map contain log addresses, instead of disk addresses. Because of this similarity the file manager in the prototype is a modified Sprite file server. This greatly simplified the file manager implementation because the existing Sprite mechanisms for cache consistency, name space management, and disk layout could be used without modification. The primary modifications made to the Sprite file server were support for storing block maps, rather than file data, and the addition of routines to process deltas to keep the block maps up-to-date. This section describes how the block maps are implemented on the file manager, how clients access the block maps, and how the block maps and other metadata are stored by the file manager to make them highly available and reliable.

5.4.1 Block Map Implementation

In theory, the block maps in Zebra are no different from the block maps in a standard UNIX file system, except that the block pointers contain log addresses instead of disk addresses. The Zebra block maps could therefore be implemented in the same manner as UNIX block maps, using inodes and indirect blocks. The disadvantage of this approach is that it changes the file system’s disk layout, since a log address in the prototype is two words long, compared to the one word required for a disk address. To avoid changing the disk layout, block maps in the prototype are implemented on top of “regular” Sprite files. For each Zebra file there is a Sprite file in an underlying Sprite file system, the contents of which are the block map for the Zebra file. This allows the file manager to access the block map for a Zebra file simply by reading and writing its associated Sprite file.

Although storing Zebra block maps inside of Sprite files leaves the disk layout unchanged, there are several ways in which it is less efficient than an implementation that stores the block maps directly in the inodes. First, performance is reduced due to the extra level of indirection in the block maps. The pointers in the Zebra file inodes do not point to the Zebra file blocks directly, but instead point to disk blocks that contain pointers to the Zebra blocks. Second, the underlying Sprite file system enforces a minimum file size of 512 bytes. Files smaller than that threshold are padded out to 512 bytes in size. This means that the block maps for small Zebra files consume a disproportionate amount of space, increasing their overhead and reducing the performance of accessing them. Finally, Zebra files have a delta version number associated with them, while regular Sprite files do not. Ideally, this version number would be stored in the inode for each file. In the prototype, the version numbers are instead stored in the file manager’s checkpoints, which increases the checkpoint overhead. File manager checkpoints are described in Section 5.4.4.

5.4.2 Block Maps and Clients

The client uses the block map for a Zebra file to access the correct fragments when reading blocks, and to fill in the old block pointers in the deltas when writing blocks. When a client opens a Zebra file it sends an open request to the file manager, and receives in reply a handle for the underlying Sprite file that stores the block map. To obtain the block pointers for the Zebra file, the client simply uses the handle to read them from the Sprite file. The client then uses the block pointers to access the storage servers. Since the block pointers are stored in a standard Sprite file, they are cached on the client like any other Sprite file, avoiding the need to read them from the file manager each time the Zebra file is read.

Although clients read block pointers from the underlying Sprite file directly, they do not write the block pointers in the same way. The block map for a Zebra file can only be modified by the file manager as it processes the deltas in the client logs. Clients do update their own cached copy of the block map after writing a file block, however. If they didn't then each time a client wrote a file block its own copy of the block map would become obsolete and the client would have to fetch a new copy from the file manager.

In addition to caching the block pointers for Zebra files, the clients must also keep track of the current version of each file. Every time a block is written to a Zebra file the file's version number is incremented, so that the deltas for the file can be ordered by the version number that they contain. The Sprite open RPC was modified to return the current version number when a Zebra file is opened. The client caches this version number and increments it each time it writes a block to the file.

The purpose of the version numbers stored in the deltas is to allow the file manager to determine the order in which update deltas should be applied during recovery. It is easy to order deltas that appear in the same log, since the contents of the log are time-ordered, but it isn't as simple to order deltas that appear in different logs. The file manager uses the version numbers to accomplish this task, by ordering the deltas according to their version number. It isn't strictly necessary, however, that each delta have its own version number. It is just as easy to order the deltas if the version number is incremented only when the cache consistency of a file changes, so that a different client begins modifying the file and the file's deltas now appear in a different log. Thus all of the deltas for a file that correspond to an uninterrupted sequence of modifications will appear in the same log and have the same version number. Deltas within a sequence are ordered by their position in the log, and the sequences are ordered by the version number. The advantage of this scheme is that it reduces the rate at which the file version number changes, and the processing that must be done by the clients to keep track of the version numbers.

One difficulty in using standard Sprite files to store the block pointers for Zebra files is that block deletion is not easily handled on the clients. In theory, block deletion is a special case of block modification: the client simply issues a delta that contains a special “null” value for the new block pointer. In practice it isn't that simple. When a client modifies a file block it must adhere to the cache consistency protocol, to ensure that multiple clients don't try to modify the same block simultaneously. Sprite's cache consistency protocol depends on clients notifying the file server each time they open and close files to prevent client cache inconsistencies. Unfortunately, file deletion does not require the file to be open, hence it does not invoke the cache consistency mechanism. Sprite eliminates this potential race by handling block deletion on the file server, which can ensure that no other client will modify the block as it is being deleted. This means, however, that the clients in the Zebra prototype can't delete file blocks themselves. Instead, the delete request is sent to the file manager, which issues the appropriate deltas to delete the blocks. The client sends the file’s current version number to the file manager in the delete request, and the file manager returns the updated version in the reply.

5.4.3 Metadata Storage

The Zebra metadata must be highly available, since files cannot be accessed without their block maps. To ensure that the metadata is not lost in a crash, the Zebra file manager stores the metadata in its client log so that it is striped across the servers and protected by parity. This means that the LFS that underlies a Zebra file system is built on top of the client log, rather than a local disk. Fortunately, this change in storage location can be implemented by modifying the block device driver used by LFS to access its disk, instead of by modifying LFS itself. All that is needed is a device driver that provides the same interface to LFS as a disk, but which accesses the file manager’s client log rather than a disk. Hence the device driver provides a virtual disk interface to the client log.

The blocks of the virtual disk used by the file manager are stored in a special Zebra file called the *virtual disk file*. The virtual disk file has a particular file number that distinguishes it from “regular” Zebra files. The device driver used by LFS translates disk block requests into reads and writes to the virtual disk file. Thus the virtual disk file contains the blocks of a virtual disk used by LFS to store the metadata for the Zebra file system. Since the virtual disk file is stored in the client log, it is striped across the servers and protected by parity.

The virtual disk file introduces a circular dependency, however, because it stores the block maps for Zebra files, yet the virtual disk file itself is a Zebra file. The virtual disk file cannot be accessed without its block map, so clearly the virtual disk file cannot store its own block map. The circular dependency is broken by handling and storing the block map for the virtual disk file differently from other Zebra files. The file manager keeps the block map for the virtual disk file in an in-memory array, instead of accessing it from a file stored in the LFS. Clients never access the virtual disk file directly, so this special-handling on the file manager is not a problem for clients. The virtual disk block map is stored in the file manager checkpoints, as described in the next section. Figure 5-5 shows how the virtual disk block map is used to map virtual disk sectors to the log addresses where they are stored.

One concern about the virtual disk block map is its size, since it must fit into the main memory of the file manager. Block pointers in the prototype are two words long, so that there is 64 bits of overhead per 4-Kbyte file block. This implies that the size of the virtual disk file must be about 1/512 or 0.2% of the size of the file data stored, which in turn implies that the virtual disk block map is about 0.2% of the size of the virtual disk file, or about 0.0004% of the size of the file data. For example, a file system that contains 1 Tbyte in file data will have a virtual disk file about 2 Gbytes in size, and the virtual disk block map will be about 4 Mbytes. Thus the virtual disk block map can easily fit into the file manager’s main memory, even for file systems that are quite large by today’s standards.

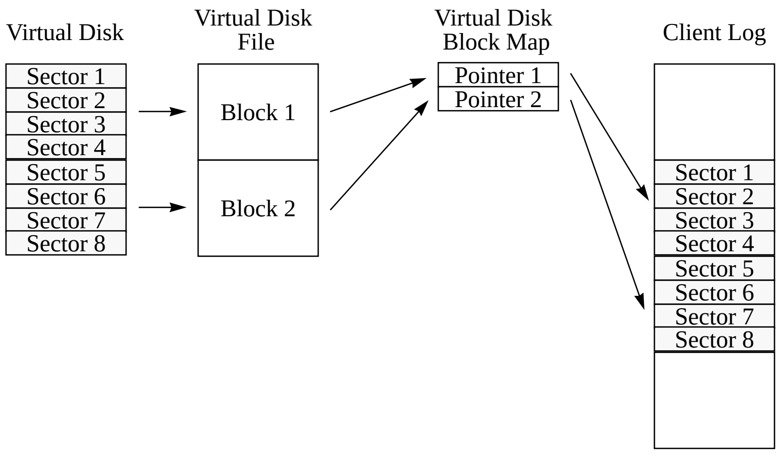

Figure 5-5. Virtual disk implementation.

Sectors of the virtual disk are stored in blocks of the virtual disk file. The virtual disk file blocks are stored in the client log at locations pointed to by the virtual disk metadata. The virtual disk metadata is stored in the file manager checkpoint (not shown) which is also stored in the client log. This diagram is simplified to show only four sectors per block; in the prototype there are eight.

5.4.4 File Manager Checkpoints

At regular intervals during its operation the file manager checkpoints its state by writing it out to its client log. A checkpoint provides a consistent snapshot of the file manager's state, so that recovery from a file manager crash does not require reprocessing all of the logs in the entire system to bring the block maps up-to-date. Instead, the file manager only needs to recover the state stored in the last checkpoint, then roll forward from that point.

A file manager checkpoint consists of a header followed by three pieces of information: a list of the file manager's current positions in processing the client logs, the array of file version numbers, and the block map for the virtual disk file. Details of the format are shown in Figure 5-6. The header contains the size of each of the subsequent regions. To create a checkpoint the file manager gathers all of this information together and writes it out to the log via `StriperPlace`. The deltas for the checkpoint are filled in so that the checkpoint blocks appear as the blocks of a special file (like the virtual disk file, it has a unique file ID). Thus to the other entities in the system that interpret the log contents, such as the striper and the stripe cleaner, the checkpoint appears as any other file. The file manager can easily find it, however, by looking for deltas that contain the checkpoint's unique ID.

The primary reason for the decision to store the file version numbers and the block map for the virtual disk in the checkpoints was that it simplified the file manager implementation. An unfortunate consequence of this decision is that file manager checkpoints can be unreasonably large. For example, in a system with 1 Tbyte of storage space and an average file size of 20 Kbytes, the file manager checkpoint would exceed

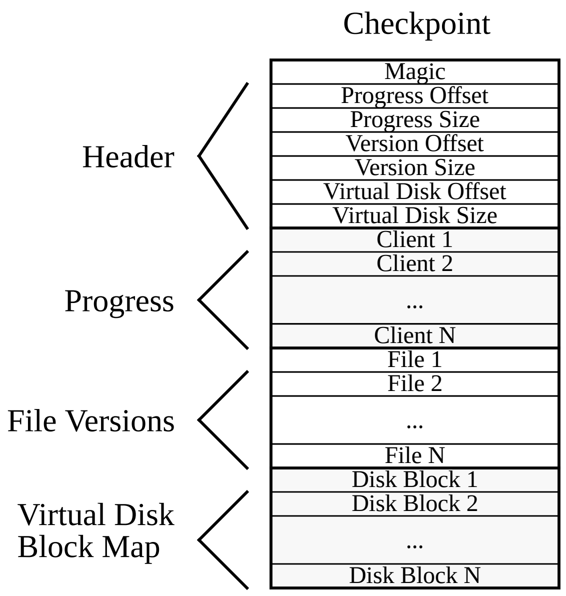

Figure 5-6. File manager checkpoint.

A checkpoint contains a header that describes the format of the rest of the checkpoint, followed by a list of log addresses indicating the file manager's progress in processing the client logs, a list of file version numbers, and the block map for the virtual disk file.

200 Mbytes in size, which is clearly too large to be practical. A better solution is to store the file version numbers in the file inodes themselves, and to use another level of indirection to store the virtual disk block map. Instead of storing the block map in the checkpoint directly, the block map is stored in another special file and that file's block map is stored in the checkpoint. This would reduce the size of the checkpoints by at least two orders of magnitude.

5.4.5 File Manager Recovery

To recover from a crash, the file manager searches backwards through its client log to find the most recent checkpoint, initializes its state based upon the checkpoint contents, then rolls forward through the client logs and processes the deltas to bring its state up-to-date. In the prototype the file manager finds the end of its log by sending *configuration* RPCs to the storage servers to determine the newest fragment that it stored before the crash. It then begins reading the delta regions in the log until it finds deltas that contain the file ID of the checkpoint. The checkpoint header contains the number of blocks in the checkpoint, making it easy to ensure that all of the checkpoint blocks were written before the crash. If the most recent checkpoint is found to be incomplete, the file manager simply continues to search backwards until it finds the previous checkpoint. Once a complete checkpoint is found its blocks are read and the file manager's state initialized based upon their contents. The file manager then rolls forward through the client logs and processes the deltas as described in Section 4.4.6.

The file manager in the prototype is assumed to always run on the same client, so that during recovery the file manager always looks in the log of the client on which it is running for the last checkpoint. This reduces the file manager’s availability since the client on which it runs could suffer a hardware failure that would prevent it from recovering quickly. This problem is easily rectified by specifying a client ID when initializing the file manager that tells it which client log to search for the checkpoint. The file manager can then be moved to another client and still be able to find the most recent checkpoint and reprocess the client logs.

A more pressing problem caused by moving the file manager to another client is that the underlying Sprite operating system will become confused by the move. The Sprite file system would have to be modified to allow clients to recover with a different server from the one with which they were communicating before the crash. In particular, the new file manager needs to recover the cache consistency state that was lost in the crash. In Sprite this information is provided by the clients, but a highly available file manager would require the clients to send this information to a different server instead. I did not make these modifications to Sprite, so the Zebra prototype does not implement a highly available file manager.

5.4.6 Delta Buffer

Delta processing on the file manager is driven by the cache consistency mechanism: the file manager only processes the deltas for a file when a client opens the file and the file’s block map is out-of-date. During the interval between their arrival and when they are processed the deltas are stored the *delta buffer*. The delta buffer accumulates deltas, allowing the file manager to process more than one at a time and reducing the processing overhead. The delta buffer is also used by the stripe cleaner to obtain deltas for processing, so that the file manager and stripe cleaner share copies of deltas, thereby reducing the amount of storage needed for the deltas. Once a delta has been processed by both the file manager and the stripe cleaner it is deleted from the buffer.

Although the delta buffer allows the file manager and stripe cleaner to batch together deltas for processing, the rate at which they process deltas must equal the rate at which deltas arrive in the buffer. If they do not, the delta buffer will grow excessively large. To avoid this problem, the delta buffer in the prototype has a fixed size, and once it is full the file manager and stripe cleaner begin processing the deltas it contains. Each time deltas are placed in the buffer its size is checked, and if it is approaching the maximum size the file manager and stripe cleaner are notified to begin processing deltas so that their space can be reused.

In retrospect, the delta buffer is an unnecessary complication in the file manager implementation. Instead of using a delta buffer, the file manager and stripe cleaner could just process deltas as they arrive. If it is desirable to reduce processing overheads by batching deltas together, this batching can be done internally to the file manager and stripe cleaner. This solution might not be as efficient as the delta buffer, but it would greatly simplify the implementation.

5.5 Client Cache

Each Zebra client keeps a cache of file blocks in its main memory. This cache serves four purposes: it allows multiple reads and writes to the same file block to be handled in the cache without accessing the server; it filters out short-lived data written by applications so that they die in the cache without being written to the server; it allows newly-written file blocks to be batched together and written to the servers in large, efficient transfers; and it serves as a read-ahead buffer to reduce the latency and improve the performance of file reads. The following sections describe in greater detail how the Zebra client cache uses the striper to write file blocks to the servers, and how it implements read-ahead.

5.5.1 Cache Block Writes

The Zebra file cache uses the striper module described in Section 5.2.2 to batch together dirty file blocks and stripe them across the storage servers. Dirty blocks are written out of the cache when the cache fills with dirty blocks, the blocks reach 30 seconds of age, or the application uses `fsync` to force a file's blocks be flushed. To write a dirty block, the cache first locks the cache block so that it is not modified during the I/O operation. Then the cache initializes an update delta for the block by filling in its old block pointer and file version number. The cache block and update delta are passed to `StriperPlace`, which places the cache block into the client log and fills in the new block pointer in the update delta. The new block pointer is then used by the cache to update the block map for the file.

The callback routine passed to `StriperPlace` is used to update the state of the cache block after its fragment has been stored on the storage servers. The callback data parameter is a pointer to the cache block. The callback routine uses this pointer to mark the cache block as no longer dirty, and to unlock it. Thus the cache block remains locked by the cache until the striper has written it to the storage servers. Since the striper fills complete fragments before sending them to the servers there may be a long delay between the time a block is given to the striper and when it gets stored on the servers. For this reason, the cache calls `StriperFlush` after it has written out all of dirty blocks that it wishes. This ensures that the blocks are written to the servers in a timely fashion and the cache blocks are unlocked and can be reused.

The cache uses three techniques to optimize the layout of file blocks within the log. First, it segregates dirty blocks based upon the files to which they belong. All of the dirty blocks from one file are placed in the log before placing blocks from another file. Second, the dirty blocks within a file are placed in sequential order. In the UNIX office/engineering environment files are typically read and written sequentially and their entirety [Baker92]. By placing the blocks in the log in sequential order they will end up in the log contiguously, improving the performance of reading them back. Third, files written by the cleaner are segregated from files written by application programs. The intent is to segregate file blocks by their expected lifetimes. If file blocks are clustered in the log based upon how long they will live, all of the blocks in each fragment will die at about the same time. This makes the fragments efficiently utilized before their blocks die, and easy to clean once they do. The expected lifetime of a file block in the UNIX environment has been measured to be proportional to its age [Baker92], so that older blocks tend to live longer than newer blocks. Thus the cleaner overheads can be reduced by segregating blocks being cleaned, which tend to be old because the cleaner favors cleaning old fragments, from blocks newly created by applications programs, so that file blocks are clustered based upon their expected lifetimes.

5.5.2 Cache Block Reads

When an application reads a block that is not in the cache, the file's block map is used to fetch the missing block. The details of how the client gets the block map are found in Section 5.4.1, but once the client has the block map it is a simple matter for it to read the block. The fragment ID is extracted from the block pointer and parsed to determine which server stores the fragment, and a retrieve RPC is sent to the storage server to obtain the file block.

The client caches in the prototype use read-ahead to improve the performance of file reads. When an application opens a file and reads the file's first block it is assumed that the application will read the file sequentially and in its entirety. The cache then begins reading file blocks from the storage servers before they are read by the application. Read-ahead serves two purposes. First, it allows the reading of file blocks from the storage servers to be overlapped with computation by the application program. Read-ahead ensures that when an application reads a block it is found in the cache, so that the application does not have to wait for it to be read from the servers. Second, read-ahead allows file blocks to be transferred from the storage servers in parallel. When an application reads from a file there is no guarantee that it will read enough data to span the servers. If, for example, read-ahead were not used and an application read only one block at a time, the servers would be accessed one-at-a-time, rather than in parallel. By reading far enough ahead in the file, the cache is able to ensure that the servers are kept busy transferring file data to the client.

The read-ahead mechanism in the prototype is implemented partially by the client cache, and partially by the storage server. During read-ahead of a file, the client cache is responsible for ensuring that the storage servers are accessed in parallel. To do this the client issues asynchronous retrieve RPCs to the servers, based upon the contents of the file's block map. When a retrieve completes the client issues a new retrieve request to the same server. This mechanism does not ensure that individual server are fully utilized, however, because it does not initiate concurrent read requests to the same server. The bandwidth attainable from a single server is maximized if both its disk and its network interface are fully utilized, by overlapping network and disk transfers. In the prototype the storage server is responsible for overlapping network and disk transfers, as described in Section 5.8.5. This division of labor is merely an artifact of the prototype, since it simplified the implementation. I expect that in a production version of Zebra the read-ahead mechanism would be implemented entirely on the clients.

5.6 Fragment Reconstruction

When a storage server is down, it is the clients' responsibility to reconstruct any fragments they need from that server. Fragment reconstruction is performed by reading the other fragments in the stripe and XORing them together. When a read is made to an unavailable fragment, the client allocates buffers for the other fragments in the stripe and reads the fragments into the buffers in parallel. The buffers are then XORed to produce the missing fragment. As an optimization, the file's block map is used to copy any file blocks that appear in the stripe's fragments out of the buffers where they are stored and into the file cache. This allows subsequent accesses to file blocks in the same stripe to be satisfied in the file cache.

Reconstruction is not automatic in the prototype: clients do not automatically reconstruct a fragment if they cannot contact a server. Clients must be manually configured to reconstruct fragments from a particular server. Automatic reconstruction only requires integration of the reconstruction facility with the RPC system so that a loss of communication triggers reconstruction, but I have not done this.

5.7 Stripe Cleaner

The stripe cleaner's task is to reclaim unused space on the storage servers. It does this by moving live data out of old stripes into new stripes, thereby leaving the space occupied by the old stripes free of live data and available for reuse. To perform this task the stripe cleaner must keep track of the state of the system's stripes. First, to clean a stripe it needs to know which of the stripe's blocks are in-use, and which are not. Second, to intelligently choose which stripes to clean it needs to know the age of the live data in the stripes. Third, to decide when to clean it needs to know how much of the system's storage space is in-use. The stripe cleaner obtains this information by processing the deltas in the client logs, as described in Section 4.5.1. The remainder of this section describes how the stripe cleaner is implemented in the prototype.

The stripe cleaner in the prototype is a user-level process that runs on the same machine as the file manager. The cleaner maintains three pieces of information about the existing stripes: the number of free stripes, a stripe statistics database, and a collection of stripe status files. The first is the number of stripes that can be written before the storage servers become full. The stripe cleaner decrements this number each time it processes a delta for a new stripe, and increments this number each time it cleans a stripe. The stripe statistics database contains the number of live bytes and the average age of those bytes for every stripe in the system. This information is maintained in a hash table indexed by stripe ID, and stored in the stripe cleaner's checkpoints. The cleaner also maintains a stripe status file for each stripe in the system. Each stripe status file is a regular Zebra file that contains descriptions of changes in the states of the blocks of a stripe, and is named by the stripe's ID. There are two types of state changes: block creations caused by a client writing a file block to the fragment, and block deletions caused by a client deleting or overwriting a file block. The description of each state change is 32 bytes in size. Since a block may change state at most twice, once to be created and once to be deleted, the resulting space overhead of the stripe status files is 64 bytes per 4-Kbyte file block, or about 2%. This overhead could be reduced by having state changes apply to a range of blocks for a file, so that a single entry in a status file describes a change to several blocks of the stripe.

5.7.1 Cleaning

The cleaner starts cleaning when the number of free stripes falls below a specified low-water mark, and it ceases cleaning when the number rises above a high-water mark. The algorithm used by the cleaner was described in Section 4.5. The cleaner first deletes any empty stripes, then prioritizes the remaining stripes and cleans them based upon their priority. An empty stripe is one that doesn't contain any live bytes. The cleaner uses the stripe statistics database to determine which stripes are empty. The cleaner then processes the stripe status files for the empty stripes to verify that they are indeed empty, then uses the *delete fragments* RPC to delete the fragments on the storage servers. After a stripe has been deleted, the cleaner updates its state by incrementing the number of free stripes, removing the deleted stripe from the stripe statistics database, and deleting the stripe's status file. If the deletion of empty stripes raises the number of free stripes above the high-water mark the cleaner ceases operation until the low-water mark is again reached. Otherwise the cleaner begins to clean stripes that contain live data.

The cleaner uses a special kernel call to clean a stripe. The parameter to the kernel call is a list of block pointers for the blocks to be cleaned. The kernel uses these block pointers to read the blocks from the storage servers into the file cache, then it marks the blocks as dirty so that they are written out of the cache. The kernel call for cleaning blocks has the same effect as reading and rewriting the blocks except that (a) it doesn't open the file or invoke cache consistency actions, (b) the file blocks are read using the block pointers provided, rather than the file's block map, (c) the data are not copied out to the user-level stripe cleaner process and back to the kernel again, (d) the last-modified time and version number for the file are not modified, and (e) cleaner deltas are created when the dirty blocks are written out of the cache, rather than update deltas. The system call is not synchronous; it only causes the file blocks to be brought into the cache and marked as dirty. The blocks will be written out of the cache according to the cache write-back policies described in Section 5.5.1.

Because the cleaning system call is not synchronous, the cleaner cannot assume that a stripe has been successfully cleaned once the system call returns. Instead, the cleaner continues to process deltas and waits for the number of live bytes in the cleaned stripe to drop to zero. Cleaner deltas will be generated when the cleaned blocks are written to the client log, and when the cleaner processes these deltas the utilization of the stripe will fall to zero. Once this happens the stripe is empty and can be deleted, freeing up the space that it occupied.

As described previously, the stripe cleaner uses an optimistic cleaning algorithm that does not prevent a race from occurring between the cleaning of a block and a modification of the same block by a client. This does not mean, however, that the cleaner does not make progress if a race occurs. The user process modified the block, so it has necessarily moved to another stripe, leaving the copy of the block in the original stripe unused. Thus once the stripe cleaner has invoked the system call to clean a stripe, it is guaranteed that the utilization of that stripe will eventually drop to zero, provided of course that the machine on which the cleaner is running does not crash while processing the system call. Stripe cleaner crashes and recovery are covered in the next two sections.

5.7.2 Cleaner Checkpoint

During normal operation the cleaner checkpoints its state to its client log at regular intervals to allow it to quickly recover from a crash. This state includes the cleaner's progress in processing the client logs, the stripe statistics database, and the stripe status files. The first step in creating a checkpoint is to force any dirty blocks for the stripe status files out to its client log. The cleaner then writes a checkpoint out to a regular Zebra file, alternating between two files to avoid corruption caused by a crash during a checkpoint. The checkpoint file contains both the pointers to the last delta processed in each client log and the stripe statistics database. Once the checkpoint file has been written to the cache, it is forced out to the log and the cleaner continues processing deltas.

5.7.3 Cleaner Recovery

After a crash, the cleaner recovers its state from the most recent checkpoint file. If the most recent file is found to be corrupted, the older checkpoint is used instead. The contents of the checkpoint file are used to initialize the state of processing the deltas, and to initialize the stripe statistics database. The cleaner then begins processing deltas. During this recovery phase it is possible for a delta to be applied to the statistics database and the stripe status files more than once, since a stripe cleaner crash may occur after processing a delta but before a checkpoint. If this happens, the stripe statistics database will be incorrect, and the affected status file will contain duplicate entries. The incorrect stripe statistics may cause the cleaning priorities to be computed incorrectly, leading to reduced cleaner performance. I have not quantified the effect of these errors, however. The duplicate entries in the stripe status files are easily detected when the status files are processed, so that there is no danger of duplicate entries causing file data to be lost.

5.7.4 Cleaner Improvements

The prototype stripe cleaner has several shortcomings that preclude its use in a production Zebra system. First, the cleaner does not handle log address wrap, as described in Section 4.5. Zebra requires the cleaner to clean stripes to prevent clients from generating log addresses that conflict with stripes that already exist. This requires synchronization between the clients and the stripe cleaner to ensure that stripes are cleaned before their stripe IDs are reused. Second, the cleaner will deadlock when the storage servers run out of free space. The cleaner produces free stripes, but requires free stripes to do so. If the system runs out of free stripes the cleaner will not be able to copy live data into new stripes, and the system will come to an abrupt halt. Third, the prototype cleaner is confined to run on a single machine. It cannot be started on a different machine if its original machine should fail. Finally, the cleaner does not retry delete fragment operations that failed due to a crashed storage server. None of these changes is particularly difficult to implement; I chose to leave them unimplemented since doing so reduced the overall implementation effort, and since they are not crucial to the validity of the prototype.

5.8 Storage Server

A Zebra storage server is simply a repository for stripe fragments. The interface to a storage server consists of operations to store a new fragment, append to a fragment, read from a fragment, and delete a fragment. The Zebra architecture requires that these operations be both durable and atomic. To be durable the effects of an operation must be permanent, barring a disk failure. When a fragment is stored, for example, it should continue to be stored by the server until deleted by a client. Thus the state of a storage server's fragments must be unaffected by a server crash. The operations must also be atomic, meaning that the failure of an operation due to a crash cannot leave the state of the fragments only partially modified. For example, a crash during an atomic append operation must result in either in all of the data being appended, or none of them.

In addition to these correctness constraints, the prototype storage server also has two performance goals. First, the number of disk accesses required to implement an operation should be minimized. Ideally, each operation should require only a single disk access. Second, crash recovery should not be proportional to the number of fragments stored by the server or the size of the disk. In particular, it should not be necessary to scan the entire disk after a crash to initialize the state of a storage server.

5.8.1 Volumes

The Zebra architecture was described as if each storage server stored a single fragment of each stripe. While this is likely to be the most common configuration, the prototype does provide a level of indirection between the clients and the servers that allows a single server to store multiple fragments of a stripe. This indirection is provided by an abstraction called a *volume*. Fragments are striped across volumes, rather than servers. This allows the system to be configured in a variety of ways, from having one disk on each server, to having all of the disks on a single server and configured as a RAID and thus a single volume, to having all of the disks on a single server but visible to the clients as separate volumes. Each volume in the system is identified by a unique integer called the *volume ID*. An RPC that accesses a fragment must specify the ID of the volume that stores the fragment. For the remainder of this section a volume will be treated as if it were a single disk, and the terms will be used interchangeably.

5.8.2 Storage Management

This section describes the data structures used by the storage server to keep track of the fragments it stores and to manage its disk space. The data structures allow the server to efficiently and reliably perform fragment operations as requested by clients.

5.8.2.1 Frames

Zebra uses a very simple scheme for storing fragments in a volume. Most of the space in a volume is divided into fixed-size *frames*, each of which is large enough to store an entire fragment. A frame stores at most one fragment, and a fragment is stored in a single frame. In addition to fragment data, each frame also contains a header that describes the fragment. This scheme makes it easy to manage the storage space in the volume and keep track of where fragments are stored, since there is a one-to-one mapping between fragments and frames, and the frames are a fixed-size. Accessing a fragment is as simple as determining which frame it is stored in, and allocating and freeing frames is easily implemented using a list of free frames.

One consequence of using frames for storage management is that fragments that do not fill a frame result in wasted space. Clients, however, do not start a new fragment until they have filled the current one, so the wasted space is minimal. If a client is forced to store a partial fragment (because of an `fsync`, for example) the client uses append operations to continue to add data to the fragment until it is full.

The header in each frame contains information about the fragment stored in the frame, as shown in Table 5-1. The use of the **index** field is described below. The **fragment ID** contains the unique ID used to identify the stored fragment, and the **fragment size** field contains the number of bytes in the fragment. The **fragment checksum** contains the checksum for the fragment computed by the client, and is used to verify the consistency of the frame header and the fragment after a crash. The **timestamp** contains the time when the frame header was written. The **complete** field indicates whether or not the fragment is in the process of being filled. When a client stores or appends to a fragment it indicates whether or not it intends to append more data to the fragment. This information is used by the storage server during crash recovery, as described in Section 5.8.3. The **complete** field is needed in addition to the **fragment size** field because a client may not fill a fragment completely before beginning a new fragment. This can happen when there isn't enough space remaining in a fragment to hold a complete file block. The **parity** and **parity sequence** fields indicate whether or not the fragment is a parity fragment, and if so, its sequence number. This information is used by the server during recovery to eliminate duplicate copies of a parity fragment (also described in Section 5.8.3).

The header for a frame is stored adjacent to the frame data, making it possible to store a new fragment in a single disk access that writes both the header and the fragment data. Append operations to existing fragments cannot take advantage of the same optimization because the existing fragment data are stored between the header and the data to be appended. Zebra assumes that clients normally write full fragments and rarely append to

Field	Bytes	Description
Magic	4	Distinctive value used for debugging
Header Index	4	Index of the header, either 0 or 1.
Fragment ID	4	ID of the fragment stored in the frame
Fragment Size	4	Number of bytes stored in the fragment.
Fragment Checksum	4	Checksum of the fragment.
Timestamp	4	Time when the subheader was written.
Complete	4	If true, data cannot be appended to the fragment.
Parity	4	If true, the frame contains a parity fragment.
Parity Sequence	4	Parity fragment sequence number (if applicable).

Table 5-1. Frame header.

Each frame has two headers with the format shown. The fragment checksum is used during recovery to verify the consistency of the frame header and the fragment it stores. The timestamp is used to determine which of a frame's headers is newer. The complete, parity, and parity sequence fields are used during recovery to update the server's internal data structures as described in the text.

existing fragments, so it is reasonable to optimize the storage server's performance to favor stores at the expense of appends.

Append operations are also more difficult to make atomic than store operations. The atomicity of a store operation is ensured by the checksum in the frame header. If a store operation is interrupted by a crash, the checksum in the header won't match the checksum in the fragment, and the contents of the frame can be discarded. Append operations, on the other hand, modify a frame that already contains fragment data. A failed append must not lose the previous contents of the frame. This implies that an append operation cannot overwrite the contents of the frame header, since the header could be corrupted by a crash. To avoid this problem, each frame has two headers that are written in an alternating fashion by store and append operations. The **timestamp** fields in the frame headers make it easy to determine which header is more recent. If a failed append operation leaves the header it was writing corrupted, the server simply uses the older header for the frame, causing the frame contents to revert to their state prior to the append.

5.8.2.2 Fragment Map and Free Frame List

The storage server maintains two in-memory data structures to keep track of the volume contents. First, the *fragment map* makes it possible to find the frame that stores a desired fragment. The frames in the volume are numbered, and the fragment map is a hash table that maps a fragment ID to the number of the frame that stores the fragment. When a new fragment is stored a frame is allocated for it, and the mapping added to the fragment map.

When a fragment is deleted its mapping is removed from the fragment map. Parity fragment overwrites are implemented by allocating a frame for the new copy of the fragment, writing the fragment to the frame, and updating the fragment's entry in the fragment map.

The storage server also maintains an in-memory list of free frames, appropriately called the *free frame list*. Frames are removed from the list as they are allocated and added to the list as they become free. A frame can become free either as a result of a delete operation from a client, or in the case of parity fragments, by being supplanted by a newer copy of the fragment.

Copies of the fragment map and the free frame list reside only in the memory of the storage server; they are not stored in the volume. The primary reason for this is that they are easily created from the contents of the frame headers during server recovery. The simplest technique for doing so is to read all of the frame headers from the volume, but this requires a disk access for each frame header, leading to server recovery times that are proportional to the amount of storage on the server. For this reason, the prototype storage server uses a different mechanism for creating the fragment map and free frame list that requires fewer disk accesses, as described in the next section.

5.8.2.3 Summary Table

The problem with storing frame information in a frame header is that it is efficient to access for some tasks, but inefficient for others. On the one hand, storing the information in a frame header adjacent to the fragment data allows both header and the data to be written in a single efficient transfer. On the other hand, headers make it inefficient to read the information for all of the frames, since it requires one disk access per frame. For this task it is preferable to cluster the frame information on disk, so it can be read in only a few transfers.

The prototype storage server resolves this dilemma by storing two copies of the information for each frame: one in the frame's header, and the other in a data structure called the *summary table*. The summary table summarizes the contents of the frame headers and can be read in a few large transfers, allowing the storage server to create the free frame list and fragment map without reading all of the frame headers. Of course, there is an issue of maintaining consistency between the summary table contents and the frame headers since they both describe the state of the frames. Thus, when the storage server reads the summary table it must first verify the consistency of its entries. This is done by comparing the contents of the summary table entry with the frame header it summarizes, as described in the next section.

The format of the entries in the summary table is shown in Table 5-2. Most of the fields correspond to fields in the frame headers. One exception is the **status** field, which indicates the state of the frame. The **status** field can be one of **empty**, **complete**, or **active**, and roughly corresponds to the **complete** field in the frame header, except that the status can be empty as well as complete or active. A complete frame contains a fragment that has been finished and will not be appended to by a client. The fragment in an active frame is still being filled and might have data appended to it in the future. Clients indicate whether or not the fragment is complete when they store and append to it, as described in Section 5.8.4.

Field	Bytes	Description
Magic	4	Distinctive value used for debugging.
Frame Index	4	Number of the frame described by this entry.
Status	4	Either **empty**, **complete**, or **active**.
Fragment ID	4	ID of the fragment stored in the frame.
Fragment Size	4	Size of the fragment, in bytes.
Current Header	4	Index of most recent frame header, either 0 or 1.
Timestamp	4	Time when the frame was last written.
Parity	4	If true, fragment is a parity fragment.
Parity Sequence	4	Parity fragment sequence number (if applicable).

Table 5-2. Summary table entry.

Each entry in the summary table describes the contents of a frame, and has the format shown. The status is used to build the fragment map and the free frame list. The current header is used to determine which frame header should be written next. The timestamp is used to verify the consistency of the summary table entry with the frame header it describes. The parity and parity sequence fields are used to eliminate duplicate copies of a parity fragment.

The summary table is stored in the volume it describes, and therefore has overhead associated with the storage it consumes. Each entry in the table is 36 bytes in size, and describes a 512 Kbyte frame. Therefore the storage overhead of the summary table is 0.007% of the size of the volume. For example, a 2 Gbyte disk will have a summary table that is 72 Kbytes in size.

5.8.2.4 Summary Table Consistency

The summary table makes it easy for the storage server to determine the frame contents, but it introduces a consistency problem because it duplicates the information in the frame headers. Zebra addresses the consistency issue by allowing the summary table and frame headers to become inconsistent during normal operation, and fixing any inconsistencies during crash recovery. The advantage of this approach is that fragment operations do not need to write both the summary table and the frame header. Instead, store and append operations only write the frame header, allowing stores to complete in a single transfer to the volume. Delete operations only write the summary table entry, allowing several fragments to be deleted in a single transfer. Thus store and append operations cause the summary table to become out-of-date, whereas delete operations do not.

Inconsistencies between the frame headers and the summary table entries are detected using the timestamps they contain. If the timestamp of a summary table entry is older than the timestamp in the corresponding frame header, the summary table entry is out-of-date and must be updated. A simple way of verifying that the entire summary table is consistent with the frame headers is to compare the timestamps of all of the frame headers and summary table entries. Although this approach is easy to implement, it requires the storage server to read all of the frame headers, undermining the reason for having the summary table.

A better approach, used in the Zebra prototype, is for the storage server to deduce from the summary table contents which entries might be out-of-date, and verify the consistency of only those entries. The storage server uses the status fields in the summary table entries to identify the entries whose consistency is suspect. A summary table entry can only become out-of-date if a store or append operation is done to the frame it describes, since these operations write only the frame header and not the summary table entry. This means that only those entries whose status is **empty** or **active** can be inconsistent, since those are the only types of frames that can be modified by a store or append operation. Entries for **complete** frames do not need to be checked, therefore, since they are guaranteed to be consistent. Furthermore, the checksums in the headers for complete frames do not need to be checked either, because a checksum can only become invalid if a store or append operation was in progress when a crash occurred, yet these operations cannot be done to a frame that is complete. Thus the storage server only needs to check the frame headers and summary table entries for those frames whose summary table entry says they are empty or active.

The restriction of consistency checks to only empty or active frames does not necessarily reduce the number of checks that must be done, however, since it is possible for all of the frames in the volume to be either empty or active. Active frames are a lesser concern, because they can only occur at the tails of the clients logs, and therefore there cannot be more of them then there are clients. There is no such limit to empty frames, however. For this reason the storage server places a limit on the number of empty frames that need to be checked by limiting the number of empty frames that could have been modified since their summary table entries were written. To do this, the server uses an on-disk data structure called the *free frame pool*, which is a list of free frames that is stored on the disk at the same time the summary table is written, and from which the server allocates frames. When the free frame pool runs out of frames it is filled from the free frame list and written to the volume after writing the summary table. During recovery the only empty frames the server needs to check are those that are in the free frame pool, since those are the only empty frames that could have been allocated since the summary table was written. The combination of the summary table and the free frame pool prevents the server from having to check all of the frame headers during recovery. Instead, the server needs to check at most as many frames as there are clients plus the frames in the free frame pool. The free frame pool in the prototype has 256 entries, so that on reboot the storage server has to check at most 456 frames -- one for each of the 200 clients, plus the 256 in the free frame pool.

5.8.2.5 Implementation Alternatives

Most of the complexity in the storage server implementation comes from the desire to avoid reading all of the frame headers during recovery, and to batch together fragment deletions into a single write to the volume. This desire led to the use of the summary table and free frame pool, which in turn led to the complexities in keeping them consistent with the frame headers. In retrospect these design decisions were probably not worth the complexity they introduced. It would be much simpler to read all of the frame headers during recovery and build the fragment map and free frame list from them directly. Performance may suffer, but even with the 1-Gbyte volumes used in the prototype it only requires 2048 accesses to read the frame headers. If we assume 30 ms per disk access, the total time required to read the frame headers is about one minute, which is probably acceptable given the complexities it avoids.

Another problem with the current storage server implementation is that it does not behave correctly when it runs out of free frames. If free frames cannot be added to the free frame pool because none exist, the server simply returns an error to the client that is trying to store a fragment. Unfortunately this behavior will cause the system to deadlock, because free fragments are only produced by cleaning, a process which itself consumes free fragments. One possible solution is to pre-allocate frames for the cleaner’s use, but I have not added this to the prototype.

5.8.3 Recovery

To recover from a crash, a storage server must verify that its state is both internally and externally consistent. The former refers to consistency between the data structures internal to the storage server, and was the focus of the previous section. This section merely summarizes how the data structures are made consistent after a crash, and describes how the server deals with duplicate copies of the same parity fragment. External consistency refers to consistency between the fragments stored on the recovering server and fragments stored on the other servers of the system. As described in the chapter on the Zebra architecture, a recovering server must reconstruct the fragments for stripes that were written while it was down, and delete fragments for stripes deleted while it was down. The Zebra prototype currently does not implement either of these functions, so that a failed server is not externally consistent once it has recovered.

To verify its internal consistency a recovering storage server first reads the summary table and the free frame pool from the volume. Each frame that is either in the free frame pool or marked as active in the summary table must be processed to ensure that the checksum in its header is consistent with the data in the frame, and that its summary table entry is consistent with the frame header. Once the headers and summary table have been verified, the storage server eliminates any duplicate copies of parity fragments. A server may end up with two copies of the same parity fragment if it crashes after the new copy has been written, but before the old copy has been deleted. The storage server uses the sequence number in the summary table entries to determine which copy is older and deletes it. Once these tasks have been completed, the data structures on the storage server are internally consistent and it can begin handling client accesses.

5.8.4 RPC Interface

The clients in the Zebra prototype communicate with the storage servers via remote procedure calls (RPC). The following sections describe the parameters and results of the RPCs supported by Zebra, and the functionality provided by the server in each case.

5.8.4.1 Store Fragment

The *store fragment* RPC is used to either store a data fragment, append to a data fragment, or a store a parity fragment. The parameters of the RPC are shown in Table 5-3. The **volume ID** identifies the volume on which the fragment is to be stored, and the **fragment ID** identifies the fragment itself. The **checksum** is the checksum of the entire fragment. For appends this checksum includes any data already stored in the fragment. The **flags** field contains one or more of the following flags: `append, complete`, `parity`, `first_RPC`, and `last_RPC`. The `append` flag indicates that the data should be appended to the end of an existing fragment, otherwise a new fragment is stored. The `complete` flag indicates whether or not the fragment is complete, i.e. the client will not append to the fragment in the future if the flag is set. If the `parity` flag is set, a parity fragment is being stored. This causes the storage server to store the new copy of the parity fragment in a new frame, rather than overwriting the existing copy of the fragment.

Parameter	Bytes	Description
Volume ID	4	Identifies volume in which to store fragment.
Fragment ID	4	Identifies the fragment to be stored.
Checksum	4	The checksum of the fragment data.
Flags	4	Valid flags are: `complete`, `append`, `first_RPC`, `last_RPC`, and `parity`.
Parity Sequence	4	Sequence number for parity fragment.

Table 5-3. Store fragment parameters.

The parameters for an RPC to store a data fragment, append to a data fragment, or store a parity fragment.

The `first_RPC` and `last_RPC` flags in the store RPC allow fragments to be stored that are larger than the maximum RPC size. In the prototype, fragments are 512 Kbytes, but the maximum RPC is only 16 Kbytes. Zebra solves this problem by transferring fragments that are too big to fit in a single RPC in a series of RPCs, delineated by the `first_RPC` and `last_RPC` flag. When the server receives a store RPC with the `first_RPC` flag set it knows that the data in the RPC does not constitute the entire fragment and the rest of the fragment's data will be transferred in subsequent RPCs. The server simply stores the fragment data in its memory and sends a reply the client. The same thing is done for subsequent RPCs containing data for the same fragment, until an RPC is seen that has the `last_RPC` flag set. This flag indicates to the server that the fragment is now complete and should be stored in the volume. The reply to the last RPC is not sent until the entire fragment has been safely stored in the volume.

5.8.4.2 Retrieve Fragment

The *retrieve fragment* RPC is used to retrieve data from a fragment. The parameters are shown in Table 5-4. The **volume ID** identifies the volume that contains the desired fragment, and the **fragment ID** identifies the fragment to be accessed. The **size** is the total amount of data to be retrieved, which must be less than the size of an RPC. The **flags** field may have the `read_ahead` flag set, in which case the server performs read-ahead as described in Section 5.8.5.

Parameter	Bytes	Description
Volume ID	4	Identifies volume that stores fragment.
Fragment ID	4	Identifies fragment from which to retrieve.
Size	4	Total number of bytes to be retrieved.
Flags	4	Valid flags are: `read_ahead`.
Range Count	4	Number of ranges to be retrieved.
Range List	4/range	List of ranges to be retrieved.

Table 5-4. Retrieve fragment parameters
The parameters for an RPC to retrieve fragment data from a storage server.

The data to be retrieved are described by the **range count** and **range list** parameters, which define a list of offset and length pairs (*ranges*) to be read. This allows several blocks of data to be returned in a single RPC. If a range extends beyond the end of the fragment it is truncated, so that it is not an error to attempt to read beyond the end of the fragment. This is useful during reconstruction because the client that is reconstructing a fragment does not necessarily know the sizes of the other fragments in the stripe. The client simply retrieves data from the fragments as if they were 512 Kbytes long, and the storage servers return only as much data as the fragments actually contain.

5.8.4.3 Delete Fragments

The *delete fragments* RPC is used to delete fragments on the storage server. The parameters to this RPC are shown in Table 5-5.

Parameter	Bytes	Description
Volume ID	4	Identifies volume that stores fragments.
Fragment IDs	4/ID	List of IDs of fragments to be deleted.

Table 5-5. Delete fragments parameters
The parameters for an RPC to delete fragments on a storage server.

5.8.4.4 Configuration

The *configuration* RPC serves as a “catch-all” RPC for determining the configuration of a storage server. Its most important use is to find the most recent fragment written by each client so to allow the system to find the end of each client’s log. There are several options to the configuration RPC:

Newest Fragment ID

The client specifies the volume ID and client ID, and the server returns the most recent fragment ID written by the client.

Fragment Information

Given a volume ID and a fragment ID the server returns the current size of the fragment, whether or not it is complete, and its current checksum.

Flush Fragment Cache

Causes the storage server to invalidate the contents of its fragment cache, whose functionality is described in the next section.

5.8.5 Performance Optimizations

The storage server improves the performance of fragment reads by caching fragments in its memory. The cache is read-only and is intended primarily to serve as a read-ahead buffer. A retrieve RPC that has the `read_ahead` flag set indicates to the server that the client is reading a file sequentially, and will in the future read data from the remainder of the current fragment and from subsequent fragments in the same log. If the flag is set, the server reads the remainder of the accessed fragment into the fragment cache, so that subsequent reads to the same fragment are satisfied in the cache. In addition, the server begins reading the next fragment in the same log into the cache. As a result, a client that sequentially reads from a file that is laid out contiguously in the log can do so without waiting for a disk access on each read. As mentioned in Section 5.5.2, the read-ahead mechanism in the prototype was divided between the client and the storage server because it was simpler than implementing it entirely on the client. I expect that a production version of Zebra would implement the read-ahead entirely on the client, where it belongs.

The fragment cache is read-only; writing a fragment simply invalidates any cached copy instead of adding the newly written data to the cache. This may lead to extra disk accesses to read a fragment that has just been written, but it simplifies the cache design and improves the performance of the fragment writes because they don't have to be copied into the cache.

Fragments are replaced in the cache in an least-recently-used fashion. Each fragment in the cache is tagged with the time that it was last accessed. When a new fragment needs to be brought into the cache and the cache is full, the fragment that hasn't been accessed for the longest time is discarded in favor of the new fragment.

5.9 Summary

The prototype described in this chapter serves as a proof-by-existence that the Zebra architecture can be implemented. True, the prototype lacks some features that would be needed in a production version of Zebra, but the major components of Zebra function properly. Clients can read and write files, the file manager manages the block maps and ensures that the client caches remain consistent, and the stripe cleaner cleans stripes. The file manager and stripe cleaner recover from crashes, and clients can reconstruct fragments. Most of what the prototype lacks are features related to availability, such as automatic reconstruction of fragments when a server fails, and the ability to run the file manager and stripe server on any client in the system. I chose not to implement these features because they are not necessary to measure the scalability and availability of the system and do not affect the performance measurements presented in the next chapter.

Most of the complexity in the prototype stems for the interaction between Zebra and the underlying Sprite operating system on which it is based. Storing the Zebra block maps in regular Sprite files, and thereby leaving the disk layout unchanged, greatly simplified some aspects of the implementation, but made others more complex. For example, file version numbers could not be stored in the file inodes, so they had to be handled differently from the other file attributes such as size and age. Probably the biggest difficulty encountered was the inadequacy of using RPCs to transfer large amounts of data to multiple servers in parallel. The RPC abstraction does not allow a single process to have many outstanding requests, a deficiency that was rectified by the development of an asynchronous RPC mechanism. Even more important, though, is that flow control is essential to achieving high-performance data transfers, yet RPC does not incorporate a flow control mechanism. To get around this problem I had to add very rudimentary flow control mechanisms in Zebra. In retrospect it would have been better to use a stream-based protocol such as TCP/IP to transfer file data between the clients and the servers.

While some of the implementation complexity and sub-optimal performance can be blamed on the interaction between Zebra and Sprite, there is some for which I alone am to blame. First, the storage server implementation is overly complex, as a result of my desire to minimize storage server recovery time. In retrospect, the added complexity is probably not worth the savings. Second, the checkpoints for both the file manager and storage servers contain too much information and are likely to become excessively large in systems of moderate size. Third, the delta buffer is an unneeded complication on the file manager and should be eliminated. And fourth, a fragment format that has headers at the beginning of the fragments is probably more intuitive than the current format which has trailers at the end. Implementation time was a major consideration in the design of these features, and I expect that an implementation that places greater emphasis on performance and storage overheads will avoid the mistakes made in the current prototype.

6 Prototype Measurements

This chapter presents measurements of the prototype described in the previous chapter, including its file access performance, scalability limits, and availability overheads. This information can be used not only to determine the performance of file transfers as a function of the number of clients, servers, and file size, but also to identify the bottlenecks that limit the overall scalability of the system. The overheads of providing high availability are also measured, including the cost of parity and of checkpoint and recovery in the file manager and stripe cleaner.

6.1 Experimental Setup

The performance measurements were collected on a cluster of DECstation-5000 Model 200 workstations running a version of the Sprite operating system that has been modified to include the Zebra prototype. The workstations are rated at about 20 integer SPECmarks and each contains 32 Mbytes of memory. In the benchmarks the memory bandwidth is at least as important as CPU speed; these workstations can copy large blocks of data from memory to memory at about 12 Mbytes/second, but copies to or from I/O devices such as disk controllers and network interfaces run at only about 8 Mbytes/second. The workstations are connected by an FDDI ring and each workstation has a single network interface. I measured the bandwidth and latency of Sprite RPCs over FDDI to be 3.1 Mbytes/second and 1.3 ms, respectively. A more detailed analysis of the network performance is given in the next section. Each storage server stores its stripe fragments on a single DEC RZ57 disk attached to the server via a SCSI I/O bus. The maximum disk bandwidth is 1.6 Mbytes/second, also described in the next section.

A total of nine workstations were available for running these experiments. The minimum configuration tested consisted of one client, one storage server, and one file manager. In the maximum configuration there were three clients, five storage servers and one file manager.

For comparison, I also measured existing network file systems, represented by a standard Sprite configuration and an Ultrix/NFS configuration. The Sprite system used the same collection of workstations as the Zebra experiments, except that the standard Sprite network file system was used instead of Zebra, and the Sprite log-structured file system was used as the disk storage manager on the file server. The NFS configuration used the same client configuration as Zebra, but the file server had a slightly faster server CPU and slightly faster disks. The NFS server also included a 1-Mbyte PrestoServe non-volatile RAM card for buffering disk writes.

6.2 Limiting Factors

This section presents performance measurements for the underlying communication and storage systems used in the Zebra prototype, since the performance of these resources limits the performance of the higher-level services that use them.

6.2.1 RPC Performance

The Zebra prototype uses Sprite RPCs to send control messages between components and also to transfer file data between the clients and the storage servers. For this reason, the performance of the underlying RPC system is of importance, since it provides a lower bound on the latency of control messages, and an upper bound on the rate at which file data can be transferred.

Figure 6-1 shows the bandwidth of Sprite RPCs over the FDDI network as a function of the amount of data sent in the RPC. The reply to each RPC is empty. Each data point is the average of 100 iterations of sending an RPC. The line labeled “No Parity” represents standard RPCs, which reach a peak bandwidth of 3.1 Mbytes/second for 16-Kbytes RPCs (the maximum size of a Sprite RPC). The line labeled “Parity” shows the RPC performance when the parity and checksum of the RPC data are computed. As described in Section 5.3.3, the parity and checksum computations in the prototype are incorporated into the RPC system, since the CPU must copy each packet to the network interface board anyway. The integration of the parity and checksum computations into this copy allows the CPU to read the data once and perform several computations on them, rather than reading them once for each computation. RPCs with parity have slightly reduced performance, reaching a peak of 2.8 Mbytes/second, or only 90% of the standard Sprite RPCs. These bandwidth limits constrain the file access performance of a client to 3.1 Mbytes/second for reads, since parity doesn’t need to be computed, and 2.8 Mbytes/second for writes, which require parity and a checksum.

The latency of a zero-length RPC for which parity and a checksum is computed vs. a zero-length RPC for which they are not is 1.290.01 ms and 1.280.01 ms, respectively. Thus the difference in latency between the two types of RPCs is not statistically significant. This relatively high RPC latency has at least two implications for the overall system performance. First, a client cannot open or close a file in less than 1.3 ms, limiting the rate at which a client can open files to at most 700 files/second. Second, the time to fetch a block pointer from the file manager is at least 1.3 ms, limiting the rate at which a client can read small files whose block pointers are not cached to at least 2.6 ms/file, or 350 files/second.

Figure 6-1. Sprite RPC bandwidth.

Bandwidth increases with RPC size. Dips in the performance occur when the size of the RPC requires an additional network packet. The parity computation slows down large RPCs by about 10%.

6.2.2 Disk Performance

The performance of a storage server is not only limited by the performance of the RPC system, but it is also limited by the performance of its disk subsystem. The storage servers in the prototype store stripe fragments on a single DEC RZ57 disk connected to the server via a SCSI bus. The disks have a published average rotational latency of 8.3 ms, a track-to-track seek of 4 ms, and an average seek of 14.5 ms. The disk capacity is 1.0 Gbytes, divided into 1925 cylinders. Each track is 35.5 Kbytes in size. The raw bandwidth of the disk media is 2.5 Mbytes/second, but the disk transfers data over the SCSI bus at only 1.6 Mbytes/second in asynchronous mode. The SCSI interface board in the server does not support DMA; the server must copy data between the interface board and the kernel memory. I measured the speed of this copy to be 8 Mbytes/second.

Figure 6-2 shows the measured read and write bandwidth of the disk subsystem. In these experiments data were transferred between the disk and a buffer in kernel memory. Each data point shows the average and standard deviation of 1000 disk accesses of a given size. The starting sector of each access was chosen randomly. For one-sector transfers, the read and write performances of the disk are roughly comparable. A read of one sector takes an average of 29.86.4 ms, resulting in a bandwidth of 16.8 Kbytes/second. Writing one sector is slightly faster than reading, taking an average of 27.36.5 ms, or 18.3 Kbytes/second.

Figure 6-2. Raw disk bandwidth.

Bandwidth between the DEC RZ57 disk and the kernel memory increases with the access size. The starting sector for each access is chosen from a uniform random distribution. The maximum size of a single SCSI transfer is limited to 64 KBytes by the SCSI controller. Larger accesses are divided into several SCSI transfers, but this causes large writes to miss a revolution, limiting write performance to half of the disk media bandwidth. The track buffer usually prevents missed revolutions on large reads.

Both read and write performance increase with access size until the access size reaches 64 Kbytes. At this point the write performance levels off at 1.1 Mbytes/second, while the read performance continues to improve. There are two reasons for this plateau in the write bandwidth: the SCSI protocol limits the size of a disk access to 64 Kbytes, and a disk can handle only one access at a time. This means that to access more than 64 Kbytes on the disk, a series of 64-Kbyte transfers must be done, one after the other. During writes this causes the disk to miss a rotation after every transfer, due to the delay caused by the disk notifying the host that the write has completed, and the host responding by sending another write request to the disk. During this delay the disk continues to rotate, causing the heads to be positioned after the starting sector of the next transfer. Thus the next transfer must wait for the disk to rotate back to the starting sector, causing the disk to miss a rotation after every transfer, and the write bandwidth to level off at 1.1 Mbytes/second.

The 64 Kbyte size limit does not have the same effect on reads as it does on writes, however. The disk contains a track buffer that is used to read-ahead data from the current track. Data are read into the buffer even when a transfer ends, allowing a subsequent access to the same track to be satisfied in the buffer without missing a rotation. Thus the read performance continues to improve even when accesses require several transfers. The track buffer does not prevent all missed rotations, however, because it only reads ahead data within the current track. If the next read access is not to the current track, the read-ahead buffer may not prevent a missed rotation because it will not contain the desired data. The effect of occasionally missing rotations can be seen in the larger standard deviations of the read bandwidth than write bandwidth.

Based upon these measurements, a storage server in the prototype can read 512-Kbyte fragments from its disk at the rate of 1.6 Mbytes/second, and write them at the rate of 1.1 Mbytes/second. These bandwidths represent upper bounds on the storage server performance, since the server cannot transfer data to the clients faster than the transfer rate of its disk.

6.3 File Access Performance

The benchmarks presented in this section measure the performance of file accesses on Zebra, Sprite, and NFS, as a function of the number of clients, number of servers, and the sizes of the files accessed. Also presented are the resource utilizations on Zebra and Sprite during the benchmarks. The combination of benchmark performance and resource utilizations shows that Zebra's file access performance scales with the number of servers, that batching is effective at improving server efficiency and performance, and that the Zebra architecture is successful at shifting the load from those resources that don't scale to those that do.

6.3.1 Performance vs. File Size

The first two file access benchmarks measure file system throughput and resource utilizations while reading and writing files of various sizes. In each experiment there was one client, one file manager, four data servers, and one parity server. The client ran an application that wrote or read files, and the elapsed time and resource utilizations were measured. Files were read or written sequentially and in their entirety. Start-up and end effects for small files were avoided by having the application read or write many files in each test. Each test was run ten times and the results averaged. The standard deviations were computed, but they are not shown in the graphs since they were too small to be discernible.

Figure 6-3 shows read and write bandwidth as a function of file size, revealing that read and write throughput increase dramatically as file size increases. For large files, reading is faster than writing; this is because the client CPU is saturated when accessing large files and writing has the additional overhead of computing parity. For small files, writing is faster than reading; this is because Zebra's log-based striping batches the data from many small writes together and writes them to the servers in large transfers. Reads of small files, on the other head, are not batched. Each file must be read from the servers individually, causing small file reads to have higher overhead, and therefore lower performance, than small file writes.

The shapes of the curves in Figure 6-3 are also of interest. The read performance improves as the size of the file increases, as would be expected by the amortization of the

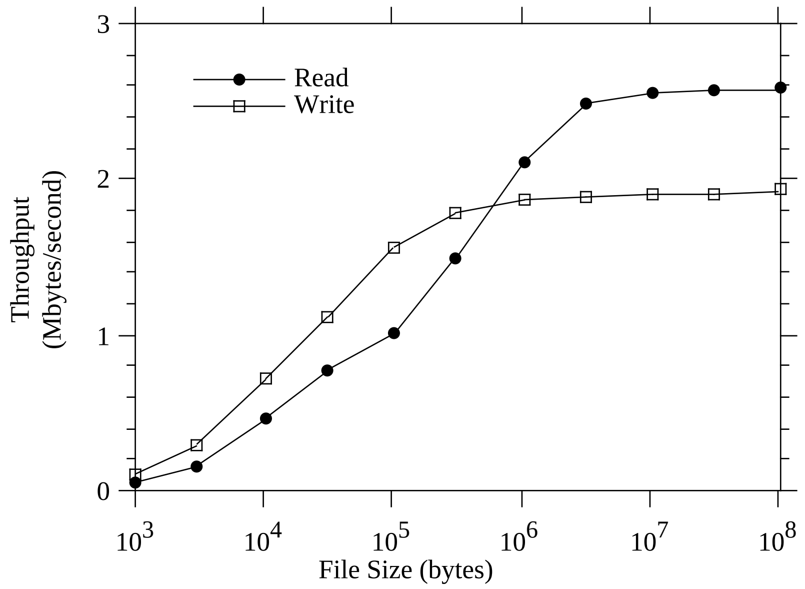

Figure 6-3. Throughput vs. file size.

A single client reads or writes files of varying size to five storage servers. Writing small files is faster than reading due to write-behind and batching; writing large files is slower than reading due to the parity computation.

per-file overheads over more file bytes transferred. The write performance shows a similar performance increase, but this behavior is not as easily explained. Small files are batched together, so that there are no-per file overheads on the servers. This implies that the write performance should not be influenced by the file size, causing the performance curve to be flat. This clearly is not the case, so there must be per-file overheads on the client that causes it to be more expensive to write small files than large.

The resource utilizations while writing files, shown in Figure 6-4, illustrate the effect of these per-file overheads. As the file size decreases, the utilizations of the storage server CPU and disk decrease, as would be expected because the write bandwidth decreases. The file manager CPU utilization is very high for small files, however. The reason for the high utilization is the processing of open and close messages from the client. Each time the client opens or closes a file it must send a message to the file manager. When writing 1-Kbyte files, most of the time is spent in opening and closing them. At the other extreme, when writing a 100-Mbyte file the open and close costs are negligible, as shown by a file manager CPU utilization of less than 2%. The bottleneck when writing large files is the client CPU, which is more than 97% utilized. The client is spending all of its time computing parity, and copying data between the application, the kernel, and the network interface.

The resource utilizations when reading files, shown in Figure 6-5, behave similarly to those when writing, but the knees of the curves occur at larger file sizes. This is because Zebra batches many small writes together, but it cannot do the same for reads. Thus larger file sizes are required to use the servers efficiently.

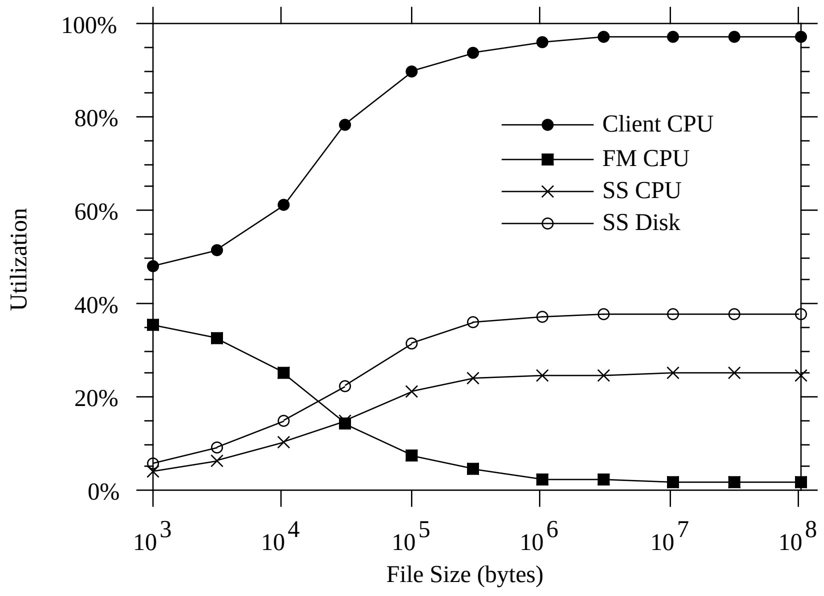

Figure 6-4. Write resource utilizations.

For small files most of the time is spent opening and closing the files; for large files the client CPU saturates. The storage server utilizations were measured on one of the five servers in the system.

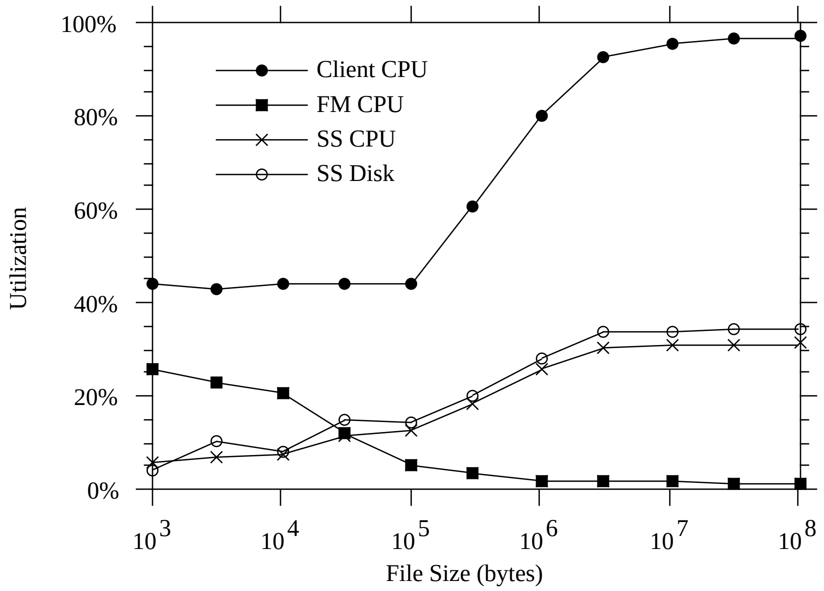

Figure 6-5. Read resource utilizations.

The curves are similar in shape to those for writing, except that the knees occur at larger file sizes. The storage server utilizations were measured on one of the five servers in the system. For small files the loads were not equal on all of the servers, causing the fluctuations in the curves.

6.3.2 Large File Performance

The previous benchmarks showed how file access performance varied with the size of the files being accessed; in the benchmarks presented in this section the size of the files is fixed, but the numbers of clients and servers are varied. These benchmarks provide insight as to how the file access performance scales with the number of servers. The first benchmark consists of an application that writes a single very large file (12 Mbytes) and then invokes `fsync` to force the file to disk. Each client ran an instance of this application (each writing a different file) with varying numbers of servers, and the total throughput of the system (total number of bytes written by all clients divided by elapsed time) was computed. The benchmark was also run on Sprite and NFS, for comparison. The results of the benchmark are shown in Figure 6-6. Note that the horizontal axis of the graph is the number of *data servers* in the system, where a data server is a storage server that stores data fragments, as opposed to the storage server that stores parity fragments. Each Zebra configuration contained a parity server in addition to the data servers. Thus the minimal Zebra configuration shown in Figure 6-6 consists of a single data server and a single parity server.

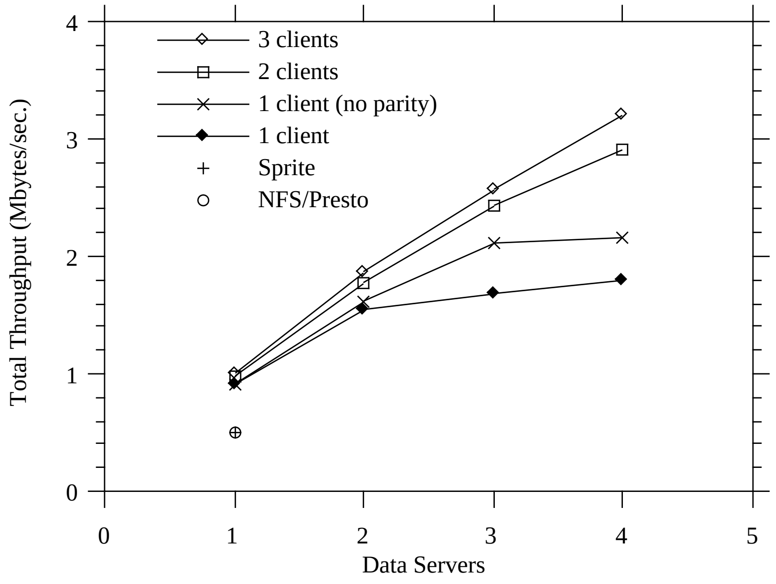

Figure 6-6. Total system throughput for large file writes.

Each client ran a single application that wrote a 12-Mbyte file and then flushed the file to disk. In multi-server configurations data were striped across all the servers with a fragment size of 512 Kbytes. Each Zebra configuration also included a parity server in addition to the data servers.

The first conclusion to be drawn is that with two or more clients the servers are a bottleneck, since performance increases steadily with each additional server. Note, however, that the performance curves are not linear. The reason for this is that adding servers only improves the rate at which data can be written from the client to the servers; it does not improve the rate at which the application program creates the data and writes it to the file cache. Thus in the limit, the write bandwidth of the entire benchmark is limited by the application performance, causing a sub-linear speedup when servers are added. This is a positive result, however, because it means that the client performance is the limiting factor in Zebra, rather than server performance as in current network file systems.

The second observation is that a single client can drive at most two data servers before it saturates. Bandwidth increases when the number of data servers is increased from one to two, but it does not increase as additional servers are added. The client is saturated at this point, so adding servers does not improve performance.

The figure also shows the overhead of computing and storing parity. The line labelled “1 client (no parity)” represents the write performance of a client that does not compute and store parity. With a single data server, parity has no effect on write bandwidth. This is because the server is the bottleneck, so that the client has plenty of resources remaining to compute and store parity. With three data servers, in contrast, the client is the bottleneck, and the overhead of handling parity reduces the write bandwidth by about 20%. Notice that the difference between the parity and no-parity lines decreases as data servers are added. As the number of data servers increases, the relative cost of parity decreases because each parity fragment protects a larger number of data fragments. As a result, the parity costs drop to zero as the number of data servers goes to infinity.

The last conclusion to be drawn from the data is that Zebra uses its servers more efficiently than either Sprite or NFS. The single-server performance for Zebra is nearly twice that of the more conventional network file systems, achieving 0.9 Mbytes/second vs. 0.5 Mbytes/second. The reason for this is that Zebra batches file data together and writes them to the server in large transfers, whereas both Sprite and NFS transfer each file block individually. Zebra also pipelines its data transfers, so that while one stripe fragment is being written to disk another is being transferred to the server. The net result is a near doubling of the single-server write performance, allowing Zebra to use nearly 82% of the disk’s raw bandwidth, as compared to 45% for the other file systems.

The second set of benchmarks, shown in Figure 6-7, measure Zebra’s performance when reading large files. As in the write benchmarks, the servers are the bottleneck when there are three clients in the system. Zebra’s performance when reading is better than when writing, however, because the servers can read data from their disks at the full SCSI bandwidth of 1.6 Mbytes/second. Thus a single client reads a file at 1.5 Mbytes/second from a single server, and three clients achieve a total bandwidth of 5.8 Mbytes/second with four data servers.

The single-client read performance also behaves similarly to the write performance. The client cannot read data fast enough to keep up with more than two servers, so that with fewer than two servers the servers are the bottleneck, and with more than two servers the client is the bottleneck.

Figure 6-7 also shows the read performance when a server is unavailable and the fragments it stores must be reconstructed. This scenario is represented by the line labeled “1 client (recon)”. With only one data server, the throughput during reconstruction is only

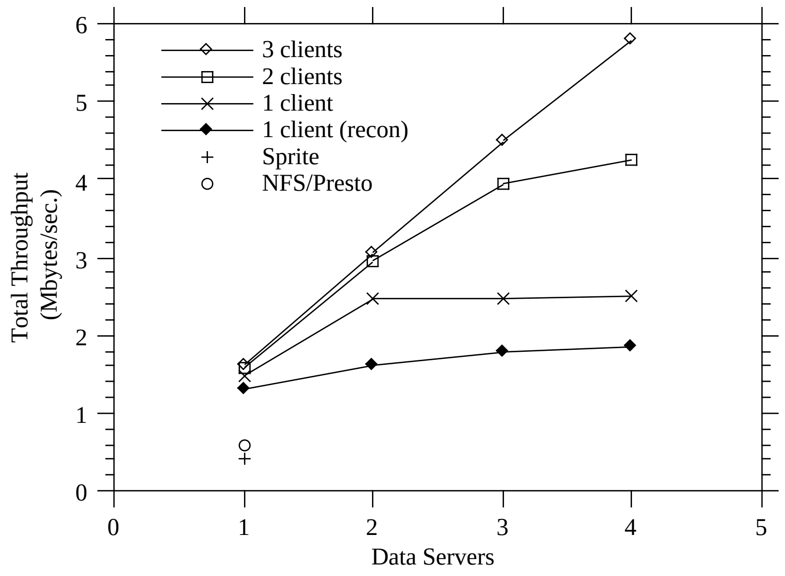

Figure 6-7. Total system throughput for large file reads.

Each client ran a single application that read a 12-Mbyte file. In multi-server configurations data were striped across all the servers with a fragment size of 512 Kbytes. The line labeled “1 client (recon)” shows reconstruction performance: one server was unavailable and the client had to reconstruct the missing stripe fragments. In addition to the servers storing file data, each Zebra configuration had a server storing parity.

slightly less than in normal operation; this is because each parity block in a system with only one data server is a mirror image of its data block and therefore reconstruction doesn’t require any additional computation by the client. The throughput doesn’t increase much with additional servers because the client CPU has saturated due to additional copying and exclusive-or operations to reconstruct the missing data.

6.3.3 Small File Performance

It is apparent from the measurements of bandwidth as a function of file size that there is a substantial amount of overhead associated with accessing a file. Both the read and write performance drop off dramatically as the file size decreases, coupled with a rise in the file manager utilization and a decrease in the client utilization. The decrease in write performance is particularly troublesome, because the batching provided by log-based striping is supposed to provide the same write bandwidth independent of the file size. This section examines in more detail the costs of writing a file, to determine the cause of the high overhead.

The overheads of writing a file were obtained by measuring the elapsed time for a single client to write 2 Mbytes worth of 1-Kbyte files. The time was broken down into different components, and the results are shown in Figure 6-8. The benchmark was run on Zebra, Sprite, and NFS to provide insight into how Zebra improves small file write performance over conventional file systems, and how it does not.

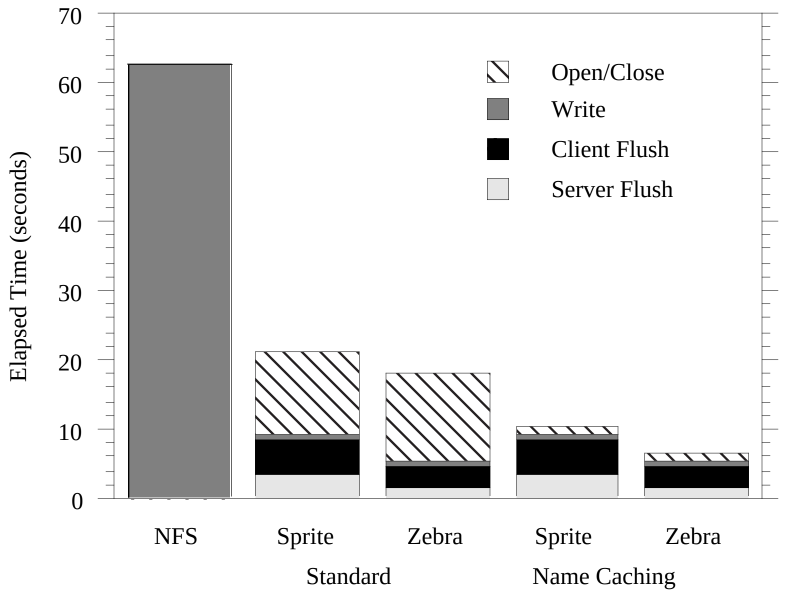

Figure 6-8. Performance for small writes.

A single client created 2048 files, each 1 Kbyte in length, then flushed all the files to a single server. The elapsed time is divided into four components: the time to open and close the files, the time for the application to write the data, the time for the client to flush its cache, and the time for the server to flush its cache to disk. For NFS, each file was flushed as it was closed. The two rightmost bars are estimates for Sprite and Zebra if name caching were implemented.

To allow a more fair comparison between the different file systems, the system configurations were made as similar as possible. In all of the tests there was a single client that wrote the files. In both the NFS and Sprite tests there was a single file server on which the files were stored, while in the Zebra test there were two storage servers (one data and one parity) and one file manager. The use of comparable configurations allows any performance advantage by Zebra to be attributed to batching, rather than its ability to use more servers.

The three left-most bars in the figure show the measured performance of NFS, Sprite, and Zebra when running the benchmark. As can be seen, both Zebra and Sprite complete the benchmark nearly three times as fast as NFS. The reason for this performance improvement is that both Zebra and Sprite batch together small writes into large transfers to the disk; Zebra does so as a result of its log-based striping mechanism, and Sprite as a result of its underlying LFS file system. NFS, on the other hand, must write each file block to the disk in a separate access.

Although Zebra completes the benchmark more than three times as fast as NFS, it is only 15% faster than Sprite. The relatively modest performance improvement over Sprite is less than might be expected from Zebra's ability to batch small files. To determine the cause of this limited performance improvement I broke down the elapsed time into four components: opening and closing files, writing file data into the client file cache, flushing the client's cache to the server, and flushing the server's cache to disk. I was unable to instrument the computers used in the NFS tests, so this breakdown of elapsed time was not possible. In the Zebra, the meaning of the last two components are slightly different than in Sprite. In Zebra the third component is the time it takes the client to flush its cache to the servers, which includes writing the data to disk. The fourth component is the time it takes the file manager to process the deltas and write the newly modified metadata to the storage servers.

The categorization of Sprite and Zebra elapsed times show that Zebra provides higher performance when writing file data, but that the benchmark is dominated by the time to open and close the files, so that the overall performance improvement is relatively small. For the components of the benchmark in which batching helps, i.e. flushing the client cache and flushing the server cache, Zebra is almost twice as fast as Sprite. This agrees with the performance numbers in Section 6.3.2, which show that Zebra's single-server bandwidth is nearly twice that of Sprite. Zebra does not reduce the cost of opening and closing files, however, and since the time required to do so accounts for more than half of the time to complete the benchmark, Zebra's overall performance improvement over Sprite is only 15%.

Clients in Zebra and Sprite must send a message when opening or closing a file to ensure that the client cache's remain consistent. This is not the only solution to the cache consistency problem, however, and it is possible that other solutions will have less of an effect on the benchmark's performance. For this reason, I estimated the performance of Zebra and Sprite if both implemented name caching, so that the clients do not need to contact the file server each time they open or close a file. These estimates are shown in the two right-most bars of Figure 6-8. The time to open and close the files was measured on a Sprite machine with a local disk, so that file opens and closes were handled locally, and the file system metadata could be cached in local memory. The bars show that name caching significantly reduces the costs of opening and closing files, so that Zebra is nearly 40% faster than Sprite. This makes intuitive sense, since the elapsed time is now dominated by the cache flushes, whose performance can be improved by Zebra's batching. Furthermore, Zebra is able to take advantage of additional servers to further reduce the cache flush time. Thus Zebra's advantages over conventional file systems are even more pronounced if name caching is used, as I would expect a production version of Zebra to do.

6.3.4 Resource Utilizations

The previous benchmarks showed that Zebra has higher read and write bandwidths than Sprite, even with only a single server in each system. This section looks at the resource utilizations in order to explain the performance differences, and provide insight into how both Zebra and Sprite will scale to larger systems and faster components. The utilizations for three benchmarks are shown in Figure 6-9: reading 12-Mbyte files, writing 12-Mbyte files, and writing 1-Kbyte files. The Zebra and Sprite utilizations are shown for each benchmark. In Zebra there is one client, one data server, one parity server, and one file manager. In Sprite there is one client and one server. The utilizations are shown for the file manager (file server) CPU, the client CPU, the storage server CPU (for Zebra only), and the disk (in Zebra the disk is attached to the storage server, while in Sprite it is attached to the file server).

Figure 6-9. Resource utilizations.

Utilizations of the file manager (FM) CPU, client CPU, storage server (SS) CPU, and the disk during the previous three benchmarks. The Zebra system consisted of a single client, a single file manager, and two storage servers, one of which stored parity; the Sprite system consisted of a single client and a single file server, which served as both file manager and storage server.

The figure shows that Zebra and Sprite use their resources very differently when reading and writing large files. First, Zebra has a higher disk utilization, which translates into less time to complete the benchmark. Second, in Sprite the file server CPU has a higher utilization than the client. This means that most of the work is being done by the file server. In Zebra, however, the file manager CPU utilization is less than 2%, with most of the work being done by the clients and storage servers. Thus Zebra shifts work from the centralized file server to the clients and storage servers, both of which are scalable resources. This allows Zebra to scale to accommodate larger numbers of clients and servers before a centralized resource saturates.

The advantages of Zebra over Sprite are not as apparent in the small file write benchmark, however. As mentioned previously, this benchmark is dominated by the time to open and close the files, so there is little performance difference between Zebra and Sprite. As can be seen in the figure, the Sprite and Zebra resource utilizations are comparable when writing small files, because both systems are spending most of their time opening and closing files.

6.3.5 File Access Summary

The file access benchmarks illustrate some of the advantages of Zebra over traditional network file systems like NFS and Sprite, including performance that scales with the number of servers in the system, and higher server efficiency. Zebra's performance improves as servers are added to the system, provided there are enough clients. Zebra's ability to scale the servers makes the client the bottleneck, so that a single client can only drive two or three servers before it saturates. This makes it possible to improve the file transfer bandwidth simply by improving the client performance. In contrast, the server is the bottleneck in traditional network file systems, necessitating a server upgrade to achieve higher performance.

The benchmarks also revealed the necessity of name caching to reduce the overhead associated with opening and closing files. Zebra is only 15% faster than Sprite when writing small files despite its ability to batch small files together; the modest performance improvement is due to the cost of opening and closing each file written. With name caching, I estimate Zebra to be 40% better than Sprite on the same benchmark.

6.4 Scalability

Although the file access benchmarks are useful for determining how access bandwidth scales with the number of clients and servers, and with file size, they do not provide much insight into how large the system can scale before a bottleneck limits its performance. The bandwidth when accessing large files, for example, continued to improve as clients and servers were added to the system. Clearly this improvement cannot go on forever, but there weren't enough machines available for use in the prototype to determine where the limit lies.

This section focuses on bottlenecks that can limit the overall scalability of the system, and estimates at what system size the bottlenecks will saturate. The estimates are made by looking at the bottleneck utilizations while running benchmarks on the relatively small system used in the prototype, then extrapolating to the system size at which the utilizations reach 100%.

6.4.1 File Manager

The file manager performs three functions that are potential performance bottlenecks: managing the file system name space, managing the consistency of client caches, and managing the file block maps. Managing the name space presents a bottleneck because all name space modifications, such as the creation of files and directories, are handled by the file manager. Managing the consistency of client caches is a potential bottleneck because the file manager must process an open and close request each time a client uses a file. These two functions are not unique problems in Zebra: every network file system must manage the file system name space and ensure that the client caches remain consistent, and doing so may limit the scalability of the system. I have not, however, measured the load these two activities induce on the Zebra file manager, partially because the difficulty in determining their cost under “real” workloads, and partially because I expect a production version of Zebra to use name caching which would significantly reduce the load anyway. Shirriff [Shirriff92] has shown that name caching can be very effective at reducing the load on the file server (a 40-Kbyte name cache on a client produces a hit rate of 97%), making the load on the file manager in the prototype very different from its load in a real system.

Management of the block maps is a problem that is both unique to Zebra and limits its overall scalability. There are two overheads caused by managing the block maps. First, the file manager must process client requests for block pointers. An upper bound on the system bandwidth caused by this processing can be determined from Figure 6-3 and Figure 6-5. When reading a 100-Mbyte file, the file manager load can be attributed almost entirely to handling block pointer requests. As can be seen from the figures, a client achieves 2.6 Mbytes/second when reading a 100-Mbyte file, with a corresponding file manager utilization of 1.3%. Thus the file manager will saturate at a client data rate of 2.6 ÷ 1.3% = 223 Mbytes/second. This translates into approximately 90 clients, each reading data at the rate of 2.6 Mbytes/second.

The second overhead of managing the file block maps is that the file manager must process the deltas from the client logs and apply them to the maps. This overhead has two components: the file manager must receive the deltas from the clients and store them in the delta buffer, and it must fetch deltas from the delta buffer and apply them to the block maps. I did not measure the file manager utilization caused by putting deltas into the delta buffer, but it can be estimated from the file manager’s I/O bandwidth and memory bandwidth. To put a delta into the delta buffer it must first be copied from the network interface board into a kernel RPC buffer, and then from the RPC buffer to the delta buffer. The first copy is done by the DMA engine on the network interface at a rate of 8 Mbytes/second, or 6 μs/delta. The second copy is done by the file manager CPU at a rate of 12 Mbytes/second, or 4 μs/delta. Thus the network interface will saturate at a rate of 175,000 deltas/second and the file manager CPU at 200,000 deltas/second, which correspond to data rates on the clients of 684 Mbytes/second and 781 Mbytes/second, respectively.

The second component of the block map overhead is the cost of fetching the deltas from the delta buffer and applying them to the block maps. Table 6-1 shows the results of an experiment in which the file manager processed 1000 deltas stored in nine stripes in its own client log. The deltas were read from the delta buffer, and the block maps (inodes) to be updated were cached on the file manager so that no disk operations were required. As can be seen, the cost of processing the deltas dominates the cost of putting them into the delta buffer. The marginal cost of fetching a delta from the buffer is 8 μs, and the marginal cost of processing it is 36 μs, for a total of 44 μs/delta in processing. When added to the time to put them into the delta buffer, this results in 48 μs/delta of overhead on the file manager caused by processing the deltas.

Activity	Per-Delta (µs)	Per-Stripe (ms)	Total (ms)
Get deltas	8	0.9 0.1	7.8 0.2
Process deltas	36	4.0 0.8	36.1 0.2
Stripe overhead		0.0 0.0	0.06 0.0
Overhead			0.2 0.0
Total	**44**	**4.9 1.0**	**44.2 0.2**

Table 6-1. File manager delta processing.

The elapsed time for the file manager to process 1000 deltas stored in nine stripes within a single client log (the same client on which the file manager was running). The time is broken down into four components: retrieve the deltas from the delta buffer, process the deltas, overhead for processing a stripe, and overhead associated with invoking the processing.

The per-delta processing time can be used to compute the maximum write bandwidth that the system can support before the file manager CPU saturates. In the steady state, two deltas are generated for each block written, one to create the block and one to delete it. The overhead for each delta is approximately 50 µs, for a total of 100 µs per block. For 4-Kbyte blocks this translates into a bandwidth of approximately 40 Mbytes/second. The maximum write bandwidth of a client is almost 2 Mbytes/second, meaning that the file manager can support up to 20 clients writing at their maximum bandwidth before the delta processing causes its CPU to saturate.

6.4.2 Stripe Cleaner

Like the file manager, the stripe cleaner limits scalability because it is a centralized service. There are three ways in which the cleaner limits system performance: the maximum bandwidth of the cleaner limits the rate at which stripes can be cleaned, and therefore overall system bandwidth; the overhead of controlling the cleaning operation limits the rate at which blocks can be cleaned, and therefore the cleaning bandwidth; and the overhead of processing deltas from the client logs limits the rate at which clients can generate deltas.

6.4.2.1 Cleaning Bandwidth

The first way in which the cleaner limits the overall system scalability is that the clients cannot write new stripes any faster than the cleaner can clean old stripes, since old stripes must be cleaned to create storage space for new. The rate at which the cleaner can clean old stripes is in turn limited by the rate at which it can copy the live data out of the stripes and to the end of its client log, and the amount of live data that must be copied to clean a stripe. Unfortunately, the latter is dependent of the workload, which makes it difficult to estimate at what point the cleaning bandwidth becomes a bottleneck. For example, consider a workload in which very large files are created and deleted. When each file is deleted, the stripes it occupied are left completely empty, so that the bandwidth required to clean them is zero. Under this workload the cleaner bandwidth can support an infinite system bandwidth.

The answer to the more relevant question of what cleaning bandwidth is required to support a real workload can be estimated from the LFS studies. Rosenblum [Rosenblum91] found that only 2-7% of the bytes in stripes that were cleaned were live and needed to be copied. If we use 5% as the number of live bytes in an average stripe that is cleaned, this translates into a 10% cleaning overhead (each byte must be both read and written by the cleaner). While this may not seem excessive, it means that the cleaner can support a maximum of 10 clients before it saturates. Clearly, the impact of cleaning on system performance bears more investigation, including a study of cleaning costs under real workloads, and the development of methods to distribute the cleaner, but I have yet to do either of these tasks.

6.4.2.2 Cleaning Overhead

The computation required to control the cleaning operation is also a performance bottleneck. Cleaning requires processing to determine which stripes should be cleaned, and to initiate the cleaning of each live block within those stripes. Table 6-2 itemizes the overheads of cleaning 100 stripes, each containing 126 live blocks. This is the maximum number of 4-Kbyte blocks that can be stored in a 512-Kbyte fragment, making the stripes unrealistically full, but it provides an upper limit of the overhead caused by cleaning a stripe, and allows the marginal cost per block to be determined.

Activity	**Per-Block (µs)**	**Per-Stripe (ms)**	**Total (ms)**
Sort stripes		0.1 0	9 4
Process status files	175	22.1 35.9	2204 711
Sort blocks	38	4.8 1.1	475 7
Initiate syscall	35	4.4 3.8	442 73
Miscellaneous			2 1
Total	**248**	**31.2 7.6**	**3131 761**

Table 6-2. Cleaning overhead.

The overhead of cleaning 100 stripes containing 126 blocks each is shown. The benchmark was run 10 times and the average values shown. The costs are broken down per block and per stripe, when appropriate. The per-block and per-stripe costs for sorting are somewhat misleading because the cost of sorting is not linear in the number of items sorted.

Table 6-2 breaks down the processing overhead of cleaning into five categories, based upon the operations the cleaner must perform to clean a stripe. First, the cleaner must compute the priority of all existing stripes, then sort them based upon their priorities. The highest priority stripes are chosen to be cleaned, and their status files processed to determine which blocks are live. The blocks are then sorted so that they are brought into the cache sequentially, and the system call is initiated to clean the blocks. The actual costs of reading and writing the blocks are not shown in the table because these costs scale with the number of servers in the system, and have already been quantified in the previous section.

As the table illustrates, it requires 248 µs of processing to clean a file block, most of which is spent processing the contents of the stripe status file. For 4-Kbyte file blocks this allows live data to be cleaned at the rate of 16 Mbytes/second before the cleaner's CPU saturates due the overhead of cleaning. If the stripes that are cleaned are assumed to contain 5% live data, this means that the cleaner can generate free stripes at the rate 320 Mbytes/second before its processor saturates.

6.4.2.3 Delta Processing

The rate at which the cleaner processes deltas also limits the overall system bandwidth, since the clients cannot generate deltas faster than they can be processed by the cleaner. Table 6-3 shows the overhead of this processing, as determined by measuring the time it takes the cleaner to process 50000 deltas. As can be seen, the marginal cost of processing a delta is 96 µs. In the steady state, the cleaner processes two deltas per block written to the client logs: one to create the block and one to delete it. Thus the overhead for each block is 192 µs. This translates into a data rate of 20 Mbytes/second for 4-Kbyte blocks, or about 10 clients writing at their maximum bandwidth.

Activity	**Per-Delta (µs)**	**Per-Stripe (ms)**	**Total (ms)**
Get deltas	20	2.5 1.2	1001 30
Process deltas	74	9.4 1.5	3726 31
Stripe overhead	2	0.3 0.0	124 2
Miscellaneous			16 1
Total	**96**	**12.2 2.0**	**4868 48**

Table 6-3. Stripe cleaner delta processing.

This benchmark measured the time for the stripe cleaner to process deltas. The results are the average of 10 runs of the benchmark in which the cleaner processed 50000 deltas stored in 397 stripes in a single client log.

6.4.3 Scalability Summary

The scalability results suffer from the lack of a real workload to measure, hence the need to perform bottleneck analysis to determine how large the system can scale. Table 6-4 summarizes the results of this analysis. As can be seen, delta processing on both the stripe cleaner and the file manager are concerns, as is the bandwidth required to clean blocks. The delta processing overhead can be reduced in several ways, including modifying the delta format so that a single delta refers to a range of file blocks, reducing the per-block cost of processing the deltas. Also, the use of stripe status files by the stripe cleaner results in overheads associated with opening and closing status files and writing the deltas to them. Simply storing deltas from multiple stripes in the same file would help reduce the processing overhead.

Activity	**Limiting Resource**	**Maximum System Bandwidth (Mbytes/second)**
Delta processing by stripe cleaner	Stripe cleaner CPU	20
Reading and writing file blocks during cleaning.	Stripe cleaner CPU	20*
Delta processing by file manager	File manager CPU	40
Responding to lock pointer requests	File manager CPU	220r
Controlling the cleaning process	Stripe cleaner CPU	320*
Receiving deltas from clients	File manager network interface	680
Putting deltas into delta buffer	File manager CPU	780

Table 6-4. Scalability limits.

This table summarizes those system activities that are performed on a centralized resource and therefore limit the overall scalability of the system. For each activity the resource that saturates is identified, as well as the overall system bandwidth that results. The activities are sorted based upon the resulting bandwidths, so that the biggest bottlenecks appear first. Bandwidths tagged with an ‘r’ are read bandwidths, all others are write bandwidths. Bandwidths tagged with ‘*’ are based on an average of 5% live data in stripes that are cleaned.

6.5 Availability

Zebra provides highly-available file service, allowing any single machine failure to be masked. This ability does not come without a price, however. The availability mechanisms incur overheads both in processing and in storage space. This section quantifies the overheads associated with storing parity and the overhead of the checkpoint/recovery mechanisms used to make the file manager and stripe cleaner highly available.

6.5.1 Parity

There is overhead in the parity mechanism both in the space required to store the parity, and in the cost of computing the parity and writing it to the storage servers. One of the advantages of log-based striping is that these overheads are smaller than they are in systems that used file-based striping. In Zebra, the storage overhead required for parity is proportional to the number of servers spanned by each stripe, independent of the sizes of the files stored. If there are N data fragments in a stripe, the parity overhead is 1/N since each parity fragment protects N data fragments. For example, if there are ten data servers the overhead of storing the parity is 10%. The parity storage overhead can thus be made arbitrarily small by striping across more servers.

The cost of computing parity fragments and writing them to the storage servers is also proportional to the number of servers in a stripe. As the stripe width grows, the parity overhead decreases because each parity fragment protects more data fragments. In a system with N data servers, the performance is reduced by a factor N/(N+1) when parity is written, since a total N+1 fragments must be written for every N data fragments. If the client is not saturated this overhead may simply increase the client's utilization, as was seen in Figure 6-6, in which the bandwidth with a single data server was the same whether or not parity was computed. If the client is saturated, however, the parity overhead will cause a reduction in the client's write bandwidth, since some of the raw bandwidth is consumed by writing parity. This effect can also be seen in Figure 6-6, in which a system with 4 data servers and a single client achieves a data rate of 2.2 Mbytes/second without parity, and 1.8 Mbytes/second with parity, the ratio of which is 1.8/2.2 or 82%, which agrees with a predicted overhead of 4/5 or 80%.

6.5.2 File Manager Checkpoint and Recovery

The checkpoint mechanism used to make the file manager highly available has several overheads. First, the checkpoints consume space on the storage servers. A file manager checkpoint contains three pieces of information: a list of log offsets that represent the file manager's current progress in processing deltas from the client logs, a list of current version numbers for each file in the system, and the block map for the virtual disk. The progress list is the smallest of the three, consuming only 4 bytes per client. Thus, for most systems the progress list will only be a few Kbytes in size. In the prototype, for example, the maximum number of clients is 200, so that the size of the progress list is 800 bytes. The file version list requires 4 bytes per file, thus the overhead is function of the average size of the files stored. If the average file size is 20 Kbytes, for example, the overhead of the version numbers is 0.02% of the total size of the file data stored. Finally, the overhead of the virtual disk block map is approximately 0.0004%, as described in Section 5.4.3. The net result is that a file manager checkpoint consumes a negligible percentage of the overall storage space.

A checkpoint may be small relative to the amount of storage space, but its size is still important because it determines the time required to create a checkpoint. To checkpoint its state the file manager must force any dirty blocks for the virtual disk file to its client log, then write a checkpoint region to the log. The time required to complete the former operation is difficult to determine, since the number of dirty blocks in the cache is a function of the workload, the locality of writes to the virtual disk, the effectiveness of the file cache, and the time since the last checkpoint. I have not run any real workloads on the prototype, therefore I have not measured the time required to flush the cache during a checkpoint.

I have measured, however, the time it takes to write out the checkpoint region itself. Table 6-5 shows the time required to create and write out a checkpoint in a system with a single data server. The checkpoint contains the delta progress for 200 clients (800 bytes), the version numbers for 100 files (400 bytes), and the block map for an 80-Mbyte virtual disk file (160 Kbytes). These numbers are simply the default parameters I chose for the prototype, and are not meant to represent the configuration of a real system. They do, however, allow the marginal costs of each component of the benchmark to be computed, which in turn allows the checkpoint costs for larger systems to be estimated. For example, consider a system with 1000 clients, 1 Tbyte of storage space, and an average file size of 20 Kbytes. A checkpoint in this system would require 4 Kbytes for the progress information, 216 Mbytes for the file version numbers, and 4 Mbytes for the block map. Based upon the numbers in the table, it would take 3 ms to put the progress information into the checkpoint, 54 seconds for the version numbers, and 4 seconds for the block map. The checkpoint would require approximately 4 minutes to write if there was a single storage server, and proportionally less time with more servers.

Activity	Elapsed Time (ms)	
Put progress into checkpoint	0.7	0.0
Put file versions into checkpoint	0.1	0.0
Put disk metadata into checkpoint	89.4	0.1
Store checkpoint in log	1.6	1.1
Flush log to server	240.7	9.4
Miscellaneous	0.01	0.0
Total	**332.6**	**8.5**

Table 6-5. File manager checkpoint.

The elapsed time for the file manager to create and write a checkpoint. The system measured had 200 clients, 100 files, an 80-Mbyte virtual disk, and a single data server. The average and standard deviations were computed from nine runs of the benchmark.

This calculation of the checkpoint cost in a large system reveals several problems with the checkpoint implementation in the prototype. The file manager checkpoint implementation in the prototype favored simplicity over performance, making it inefficient to use in a real system. First, there is no reason for the version numbers to be in the checkpoint. In a real system I would expect them to be stored in the file inodes, but in the prototype I stored them in the checkpoint because I didn’t want to modify the existing LFS disk layout. Second, it is probably unwise to store the entire virtual disk block map in the checkpoint. A better solution, discussed in Section 5.4.4, is to add another level of indirection and store the virtual disk block map in a special file, and store that file’s block map in the checkpoint. This would reduce the size of the block map in the checkpoint by two orders of magnitude. With these changes, I would expect the size of the file manager to be less than one Mbyte, even for systems with one Tbyte of storage space.

Another overhead related to file manager availability is the time required for the file manager to recover from a crash. While the file manager is recovering, the clients cannot open and close files, leaving the file system unusable. The file manager recovery time indirectly affects normal system operation, since the recovery time is proportional to the checkpoint interval. Frequent checkpointing reduces the recovery time, but it slows down the normal processing. The choice of the checkpoint interval is a trade-off between the recovery time and the overhead of checkpointing.

The file manager recovery consists of two phases: initializing the file manager state from the most recent checkpoint, and processing the deltas in the client logs to bring its state up-to-date. Table 6-6 shows the time for the file manager to initialize its state from the same checkpoint used to create Table 6-5. The checkpoint is stored in the last fragment of the log, so that the deltas of only one fragment need to be examined to find the checkpoint file. Once again, the time to process the checkpoint is excessively long because the prototype stores the file version numbers and the virtual disk block map in the checkpoint. I would expect a production version of Zebra to store this information elsewhere, reducing the size of a checkpoint to less than one Mbyte, and thus the time to process it during recovery.

Activity	Elapsed Time (ms)	
Find checkpoint deltas	332.2	10.1
Read checkpoint blocks	358.5	77.0
Initialize delta progress	12	0.1
Initialize file versions	0.2	0.0
Initialize disk block map	73	0.1
Miscellaneous	1.2	0.0
Total	**781.1**	**11.0**

Table 6-6. File manager recovery

The elapsed time for the file manager to find and read the most recent checkpoint, and initialize its state from the contents.

The breakdown of the recovery time uncovers another inefficiency in the prototype. Even though the checkpoint is only 161 Kbytes in length, it takes a total of 690 ms to find the checkpoint in the log and read it. There are two reasons for this anomaly. First, the file manager reads deltas from the log by reading whole fragments and extracting the deltas. Thus to find the deltas for the checkpoint the file manager reads the last fragment in the log, requiring 332 ms. Then, once the file manager has found the deltas for the checkpoint, it reads the checkpoint blocks from the log one at a time, even though the entire fragment has already been read. At the very least, it makes sense to read the checkpoint in a single large transfer, rather than many small ones to read individual blocks. With these corrections, I would expect the file manager to find and read a 161-Kbyte checkpoint from a single data server in less than 200 ms, instead of the 690 ms now required. This alone would reduce the time to initialize the file manager's state from the checkpoint from 781 ms to less than 300 ms.

The second part of the recovery overhead is the time for the file manager to process the deltas in the client logs to bring itself up-to-date. I have not measured this time directly, because it is dependent on the workload. It can be estimated, however, from other measurements. First, if the file manager delta processing is the system's performance bottleneck, then the time to replay the deltas will take at least as long as the time between the last checkpoint and the crash. For example, if ten seconds elapse between the most recent checkpoint and the crash, and the file manager has been saturated during that time processing deltas, then it will take at least ten seconds for the file manager to process the deltas during recovery. Another way to look at the problem is to determine recovery time as a function of the client data rate, and the checkpoint interval. For a client to write data at the rate of 1 MByte/second, the file manager must process deltas at the rate of 512/second (256 blocks/second * 2 deltas/block). The marginal cost of processing a delta was computed to be about 50 µs, so it takes the file manager about 26 ms to process the deltas generated by a data rate of 1 Mbyte/second. Thus the recovery time in seconds is given by the function 0.026 * W * I/2, where W is the system write bandwidth in Mbytes/second, and I is the interval between checkpoints (on average a crash will occur halfway through a checkpoint interval). For example, if the write bandwidth is 10 Mbytes/second, and the checkpoint interval is 60 seconds, the expected recovery time is 7.8 seconds.

6.5.3 Stripe Cleaner Checkpoint and Recovery

Like the file manager, the stripe cleaner uses a checkpoint/roll-forward mechanism to provide high availability. The striper cleaner creates a checkpoint by writing out its progress in processing deltas and the stripe statistics database to a regular Zebra file. The file is then forced through to the log, allowing the cleaner to read its initial state from this file following a crash. The cleaner alternates between two checkpoint files so that a crash during a checkpoint does not prevent the cleaner from being able to recover.

The first overhead associated with the stripe cleaner checkpoints is the amount of storage space consumed by a checkpoint file. A checkpoint file contains two pieces of information: a list of log offsets for each client that indicates how far the cleaner has gotten in processing the deltas, and a list of statistics for each existing stripe. The size of the delta progress information is 8 bytes per client, so that in the prototype this information requires a total of 1600 bytes in the file. The size of the stripe statistics information is much larger, since it is proportional to the number of stripes in the system. Each stripe requires a total of 12 bytes to record its stripe ID, the number of bytes that are alive, and the age of those bytes. For example, the stripe cleaner checkpoint in a system with 200,000 stripes is about 2.5 Mbytes in size. While this is a sizeable amount of data to write during a checkpoint, its overhead in terms of storage space consumed is negligible.

The second overhead caused by the stripe cleaner checkpoints is the amount of time required to write a checkpoint. Table 6-7 shows the results of an experiment in which the stripe cleaner wrote out a checkpoint file for a system with 200 clients and 1000 stripes. The total size of the checkpoint file is about 14 Kbytes, and it takes the cleaner 557 ms to create and write the file. Most of the time is spent in opening the file and in flushing the log to the storage server, taking 247 ms and 204 ms, respectively. The time to open the file is excessively long, probably because the file cache was cold each time the benchmark was run. This forces the file manager to fetch the inodes and directories from the storage server during the name lookup. The length of the open indicates that about ten accesses were required, but I have not instrumented the benchmark to determine if this is indeed correct. The excessive length of time to flush the checkpoint file to the storage server was probably caused by the underlying LFS file system on which the Zebra file system is implemented. The current LFS implementation lacks the roll-forward mechanism that allows it to fully recover from a crash. Instead, it simply reinitializes its state to that found in the most recent checkpoint. Thus, to ensure that a file that is forced to disk is safely written, LFS issues a checkpoint after writing the file. This means that LFS writes a checkpoint when the stripe cleaner forces its checkpoint file through to the servers, causing the abnormally large time to write the checkpoint file. This is another problem that I would expect to be fixed in a production version of Zebra.

Activity	**Elapsed Time (ms)**	
Open checkpoint file	247.2	50.1
Write delta progress to file	1.2	0.4
Write stripe statistics to file	98.2	8.6
Flush log to storage server	204.6	26.8
Miscellaneous	6.0	0.8
Total	**557.2**	**57.4**

Table 6-7. Stripe cleaner checkpoint.

This table shows the time it takes the stripe cleaner to checkpoint its state. The system had 1000 stripes, 200 clients, and a single storage server. The benchmark was run nine times and the averages and standard deviations computed.

The time required for the stripe cleaner to process a checkpoint file during recovery is shown in Table 6-8. The checkpoint file used in the experiments was the same one used to collect the checkpoint measurements, containing the delta progress for 200 clients and statistics for 1000 stripes. More than two-thirds of the recovery time is spent in opening the checkpoint file. Like the checkpoint benchmarks, the file cache was cold when this benchmark was run, forcing the file manager to fetch inodes and directories from the storage server to perform the name-lookup during the open. I have not verified the number of accesses that are actually done. The remainder of the recovery time is spent initializing the delta progress and the stripe statistics database. Processing the deltas requires 34 ms, or 170 µs/client, and processing the stripe statistics requires 32 ms, or 32 µs/stripe. Thus for a hypothetical system with 1000 clients and 200,000 stripes the total time to initialize the delta progress and stripe statistics database would be 170 ms + 6.4 seconds = 6.6 seconds.

Activity	**Elapsed Time (ms)**
Open checkpoint file	228.4 37.8
Read delta progress	34.2 4.6
Read stripe statistics	32.3 1.0
Miscellaneous	13.9 8.2
Total	**308.8 43.4**

Table 6-8. Stripe cleaner recovery.

This table shows the average time for the stripe cleaner to recover from a crash by opening the checkpoint file, read the progress for 200 clients, and read the statistics for 1000 stripes. The benchmark was run ten times.

The measurements of the stripe cleaner checkpoint and recovery times illustrate some obvious performance problems in the prototype. Not only are the checkpoint files large, but this causes the checkpointing and recovery times to be excessively long. Like the file manager, the checkpoint mechanism for the stripe cleaner was implemented primarily for simplicity, rather than performance. In a production system I would expect the stripe statistics database to be stored in a separate file, rather than in the checkpoint. This would significantly reduce the time to create and process a checkpoint file. Also, I would expect the time to open the checkpoint file to be smaller than in the prototype. I have not instrumented the kernel to determine where the time is going and make changes to reduce it. If my hypothesis about the name-lookup is correct, the open time could be reduced simply by moving the checkpoint file to the root of the file system, or by providing a means of opening a file based upon its file number rather than its name, so that name-lookup is avoided altogether.

The remaining issue in the stripe cleaner recovery is the time it takes to bring its state up-to-date by processing the deltas in the client logs that were created after the most recent checkpoint, but before the crash. This time is a function of the checkpoint interval and the workload, since a small interval and light workload means that there are relatively few deltas to be processed. The expected recovery time can be expressed, however, as a function of the system's write bandwidth and the checkpoint interval, in a manner similar to that done for the file manager. The marginal time required for the stripe cleaner to process a delta was computed to be 96 µs, or about 49 ms to process the deltas produced to write 1 Mbyte of data in the steady state. Thus the time to processes the deltas, in seconds, is given by the equation 0.049 * W * I/2, where W is the average write-bandwidth of the system in Mbytes/second, and I is the interval between stripe cleaner checkpoints in seconds. Thus a system that has an average write bandwidth of 10 Mbytes/second and a 60-second stripe cleaner checkpoint interval will require about 15 seconds to recover from a stripe cleaner crash.

6.5.4 Availability Summary

While the storage and computation overheads associated with the parity mechanism are in line with what is predicted by the Zebra architecture, the overheads of the checkpoint and recovery mechanisms are excessive. The underlying problem is that the checkpoint mechanisms used in the prototype were designed primarily for simplicity, rather than speed. If the system size is scaled up to 1 Tbyte of storage, the file manager and stripe cleaner checkpoint sizes are 224 Mbytes and 2.5 Mbytes, respectively. Both checkpoints contain information that is not strictly needed to be in the checkpoint itself, and was stored there only for simplicity. The prototype implementation is good enough to show that the checkpoint and recovery mechanisms are feasible, but they would need to be reworked in a production system.

7 Conclusion

The great promise of distributed computing is that a collection of computers connected by a network can provide higher performance, higher availability, and better scalability than a single main-frame or supercomputer, and do it for less money. This promise may have been achieved in some realms of computing, but as of yet network file systems is not one of them. Current network file systems represent very loosely-coupled distributed systems: file service is provided by a collection of separate file servers, rather than seamless pool of servers working together. Each file is stored in its entirety on a single file server. This means that the performance obtained when accessing a file is limited by the file server that stores it. Additional servers may improve the aggregate performance of the file system, but do not necessarily improve the performance of accessing a particular file. This configuration also leads to hotspots, as server loads become unbalanced due to non-uniform accesses within the file system. Servers that store popular files will be more heavily loaded that those that don't. Furthermore, the restriction of a file to a single server leaves it vulnerable to server failures.

Many attempts have been made to rectify these problems with network file systems. Most of these previous efforts tackle only one of the related problems of availability and scalability. There are many examples of network file systems that replicate files to improve availability. If the primary server for a file is unavailable the file can be accessed via a backup copy. Replication allows the system to survive an arbitrary number of server failures and network partitions (provided there are enough replicas of each file), but it has high overheads in terms of storage space and in keeping the replicas up-to-date.

File systems have also been designed that allow files to be striped across multiple storage devices, thus decoupling the file access performance from the performance of an individual storage device. Most of these file systems stripe across disks connected to a single host computer, but there are examples of file systems for parallel computers in which file data are striped across I/O nodes of the computer. These systems are all similar in that they are designed to operate in a tightly-coupled environment, such as a parallel computer. No effort is made to provide highly-available service.

There is one existing network file system, however, that provides both highly available and scalable file service. Swift [Cabrera91][Montague93] stripes files across multiple file servers and uses parity to provide highly available service even in the face of server failures. This design allows the file access performance of each file in the system to scale with the number of servers, without compromising availability. High availability does not come cheaply, however. The way in which Swift stripes files across the servers and maintains their parity leads to a performance overhead of 47% when compared to striping without parity. Furthermore, the Swift architecture leaves unanswered some issues related to storage allocation and name space management.

This dissertation has presented a new network file system called Zebra. Zebra borrows ideas from RAID and LFS and applies them to network file systems, resulting in a file system that provides scalable performance by striping files across multiple servers, and highly-available service by maintaining the parity of the file system contents. Zebra's key technology is the use of log-based striping to store files on the file servers. Each client forms file data into a log and stripes the log across the servers, rather than striping individual files. As the log is created the client computes and stores its parity. The use of log-based striping allows Zebra to batch together many small writes into large transfers to the servers, improving small write performance and server efficiency. Log-based striping also simplifies the parity mechanism, because it avoids partial stripe writes and the associated overhead for updating parity.

The use of logs to store data has many advantages in the design of a network file system. The logs can be used as reliable communication channels, by appending messages to the end of the logs. The recipient is guaranteed to receive the messages in the order in which they were sent, and furthermore, there is no danger of the messages being lost since the log is reliable. Thus the various Zebra components can use the logs to communicate changes in the system's distributed state. Deltas describe changes in block locations and are used to communicate these changes between the clients, file manager and stripe cleaner. The file manager uses the deltas to keep the file system metadata up-to-date, and the stripe cleaner uses them to manage the system's free space. Deltas are also used by the stripe cleaner and the file manager to coordinate the garbage collection of free space.

The use of deltas to communicate state changes leads to simple and well-defined interfaces between the components, and makes it easy for the components to recover from failures. For example, the file manager updates the files' block maps based on the deltas stored in the client logs. Crash recovery is handled by having it occasionally checkpoint its state to the log. After a crash it simply reads this state from the log, then begins processing deltas where it left off. No special effort needs to be made to ensure that the checkpoint or the deltas will be available after the crash, since they are stored in the logs. Furthermore, the file manager software has no dependencies on the physical machine on which it is running. It can easily be restarted on another machine should its current host fail. Stripe cleaner availability is handled in a similar manner. The net result is that the logs and deltas make it easy to maintain the distributed state of the file system, even in the face of machine failures. Without the highly-available logs, distributed state management would be much more complicated, particularly if it is to be fault tolerant.

Finally, the use of a log abstraction at the interface between the clients and the servers makes it possible to distribute the tasks of managing the system’s storage space. Clients implicitly allocate storage through the act of storing file blocks in their logs. The file manager uses the deltas in the log to keep track of where file blocks are located, so that a client that wishes to access a particular file block can easily determine where it is stored. The stripe cleaner uses the deltas to keep track of which data in the stripes are still in-use, and which are not. This information is used to clean stripes, allowing the system to reuse the unused space that they contain. Thus the tasks associated with storage allocation are easily distributed among the components of the system, instead of concentrating them in a single file server as is currently done.

7.1 Future Research in Zebra

There are several ways in which the current Zebra architecture and prototype can be improved. In most cases these improvements are needed because simplifications were made in the prototype to make it easier to implement, which in some cases led to poor performance. A production version of Zebra would need to fix these to make Zebra truly usable. The first improvement is the addition of name-caching. In the prototype, clients must contact the file manager each time a file is opened or closed. For small file accesses these open and close costs dominate the time to access the file. Name caching is a well-understood mechanism; it was not implemented in the prototype simply because of the effort required to add it to the underlying Sprite operating system.

The Zebra prototype also has a very inefficient metadata implementation. The block pointers for each Zebra file are currently stored in an associated Sprite file. This organization made it easier to add Zebra to Sprite, but it leads to inefficiencies in the way in which metadata is stored on the disk and transferred between the file manager and clients. A production version of Zebra would want to store the block pointers in the inodes directly.

The checkpoint and recovery mechanisms used in the prototype are similarly inefficient, both in the space required to store a checkpoint and the times required to create a checkpoint and recover from a crash. The current implementation was chosen because it was simple, not because it was efficient. It would be an easy task to move information out of the checkpoint that is not strictly needed to be there, greatly reducing the overheads of the checkpoints.

Flow control is also a weak point in the Zebra prototype. The data path via which data flow between the clients and the storage server disks can be thought of as a pipeline that includes the client CPU, client memory, client network interface, network, server network interface, server memory, server CPU, and server disk subsystem. Maximum performance is achieved when each one of these components is kept busy, and flow control should be provided across the entire channel. For example, if the server disk is busy then fragments should back up onto the client, at which point the application writing the data should be blocked. In the current prototype flow control is provided via careful coding of the clients and servers and by discarding data when a resource is overrun. The clients and servers implicitly understand each other's capabilities and are careful not to exceed them. This solution isn't even entirely correct because it does not handle multiple clients accessing the same server. Discarding data is a simple way of dealing with flow control, but it has a large impact on system performance. When data is discarded it will need to be resent at a later time, after an appropriate time-out period. This not only increases the amount of data transferred, but leads to long delays as the time-out period expires. A much better solution is to provide flow control between the client and the servers so that a transfer is not started unless there are sufficient resources to complete it.

The Zebra architecture itself has room for further improvement. One area is in support for transaction processing workloads. I expect Zebra to work well on the same workloads as LFS, which include most workstation applications. These workloads are characterized by short file lifetimes, sequential access to files, and very little modification of existing files. In a transaction processing environment, on the other hand, there is typically a single large long-lived file (the database) that is overwritten and accessed randomly. There is little experience with LFS in such an environment, and Seltzer's measurements [Seltzer93] suggest that there may be performance problems. More work is needed to understand the problems and see if there are simple solutions.

Zebra may also suffer performance problems under workloads characterized by reading many small files. Zebra's log-based striping allows small file writes to be batched, thereby improving their performance, but Zebra currently does not do anything to improve the performance of small file reads. Doing so requires a prefetch mechanism that allows for read-ahead among files as well as within a file. Such a mechanism would allow files that will be read in the future to be brought into the client's cache while the application that will read them is processing the current files. One interesting possibility is that the locality in reading files is similar to the locality in writing them, so that prefetching whole stripes of files will provide a substantial performance improvement.

7.2 Related Research

Zebra provides a basis for a network file system that provides scalable and highly-available file service. Zebra has its limitations, however, and there is plenty of interesting research that could be done to eliminate them and examine the feasibility of the Zebra ideas in different computing environments. The most enticing of these is to further improve Zebra's scalability by eliminating its few centralized services. The file manager and stripe cleaner both limit the scalability of the system because they are not distributed services. The file manager provides a centralized synchronization point for metadata updates and for maintaining the consistency of client caches. Future network file systems could have the clients update the file system metadata themselves, eliminating the need for the file manager to do the job. Deltas might still be maintained as a way of communicating state changes to other clients, and as a way of doing write-ahead logging to tolerate failures. The role of the file manager would then be reduced to ensuring that the client caches remain consistent. This cache management function could then be distributed, perhaps using some of the techniques used to provide cache consistency on shared-memory multiprocessors.

Distribution is also desirable for the stripe cleaner. Section 4.5.8 touched on straightforward ways of doing this distribution. Future research might explore solutions that scale better, perhaps distributing the cleaning functionality to all of the clients.

Zebra is also lacking in support for parallel computing. The use of networks of workstations to perform parallel computations almost certainly will become a reality in the near future. Zebra is not designed to support these parallel computations; in particular it does not support concurrent write-sharing nor application-directed file layout. Concurrent write-sharing is rare in the UNIX workstation environment; it is unusual for a file to be open simultaneously open by several clients, one of whom is writing the file. This is probably not the case in a parallel computation, in which many processors might cooperate to produce the output file. These applications may also wish to specify the way in which the file data should be laid out on the storage servers, to provide better performance on later accesses. Zebra implicitly assumes that files will be read sequentially and in their entirety. If the application program has knowledge that this is not the case it should be possible to provide this information to the file system so that the file is laid out properly.

Another limitation in Zebra is that the storage servers are assumed to have the same capabilities, in terms of processor speed, I/O speed, and storage capacity. This assumption is likely to be false in large-scale distributed file systems, due to the sheer number of servers involved. The server configurations will change frequently as processors are upgraded, disks break, memory is added, etc. A distributed file system should be designed to ensure that all servers are fully utilized, even if their capabilities differ and are constantly changing.

The use of log-based striping in Zebra is also interesting because it makes it possible to use compression as a way of not only reducing storage space, but also of reducing the network bandwidth requirements. One of the problems with using compression in a traditional storage manager is that the compressed size of a block of data may change if the block is updated. This makes it difficult to allocate space for a block because its final size is not known, and the size will change as the block is modified. LFS solves this problem by appending new data to the end of the log. If a block is modified and rewritten its new compressed image is simply appended to the log, and the old copy is reclaimed by the stripe cleaner. Zebra pushes the log mechanism back to the clients, allowing the clients to compress the log before writing it to the storage servers. Not only does this distribute the compression mechanism nicely, but it also reduces the amount of network bandwidth required to transfer the log fragments.

Finally, Zebra does not provide guaranteed service when accessing a file. File blocks are transferred as quickly as possible, but there is no limit on the variance between the service time for different blocks. Applications such as real-time video may prefer to have a limited variance between file blocks, perhaps as the expense of a higher average service time. It would be interesting to see how such service guarantees could be added to a file system such as Zebra.

7.3 Closing Comments

Zebra is one step in the evolution of network file systems from centralized to distributed systems. It has shown that striping and parity can be combined in a network file system to improve its performance, scalability, and availability. The use of log-based striping makes it feasible to use parity to provide high-availability, but one of the biggest surprises of the project was that it also makes it easy to maintain the distributed state of the system, by providing a reliable communication channel between the components. Deltas are a simple means of communicating changes in block locations between the clients, file manager, and stripe cleaner. Much work is left to be done, but I believe that Zebra serves as a solid basis for the network file systems of the future.

Bibliography

[Baker91]	Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, "Measurements of a Distributed File System", *Proceedings of the 13th Symposium on Operating Systems Principles (SOSP)*, Asilomar, CA, October 1991, 198-212. Published as *ACM SIGOPS Operating Systems Review 25*, 5.
[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,* June 1992, 31-43.
[Baker92b]	Mary Baker, Satoshi Asami, Etienne Deprit, and John Ousterhout, "Non-Volatile Memory for Fast, Reliable File Systems", *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* Boston, MA, October 1992, 10-22.
[Baker94]	Mary Baker, "Fast Crash Recovery in Distributed File Systems", Ph.D. Thesis, Computer Science Division, University of California, Berkeley, January 1994. Also available as Technical Report UCB/CSD 94/787.
[Bernstein81]	Philip A. Bernstein and Nathan Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys 13,* 2 (June 1981), 185-222.
[Bhide91a]	Anupam Bhide, Elmootazbellah N. Elnozahy, and Stephen P. Morgan, "A Highly Available Network File Server", *Proceedings of the Winter 1991 USENIX Conference*, Dallas, TX, January 1991, 199-205.
[Bhide91b]	Anupam Bhide, Elmootazbellah N. Elnozahy, Stephen P. Morgan, and Alex Siegel, "A Comparison of Two Approaches to Build Reliable Distributed File Servers", *International Conference on Distributed Computing Systems (ICDCS)*, 1991.
[Birman84]	Kenneth P. Birman, Amr El Abbadi, Wally Dietrich, Thomas Joseph, and Thomas Raeuchle, "An Overview of the Isis Project", Technical Report 84-642, Department of Computer Science, Cornell University, October 1984.
[Brown85]	Mark R. Brown, Karen N. Koling, and Edward A. Taft, "The Alpine File System", *ACM Transactions on Computer Systems 3,* 4 (1985), 261-293.

[Cabrera91] Luis-Felipe Cabrera and Darrell D. E. Long, “Swift: Using Distributed Disk Striping to Provide High I/O Data Rates”, *Computing Systems 4*, 4 (Fall 1991), 405-436.

[Chen90] Peter M. Chen and David A. Patterson, “Maximizing Performance in a Striped Disk Array”, *Proceedings of the 17th Annual International Symposium of Computer Architecture*, May 1990, 322-331.

[Dibble90] Peter C. Dibble, Michael L. Scott, and Carla Schlatter Ellis, “Bridge: A High-Performance File System for Parallel Processors”, *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, 1988, 154-161.

[Drapeau94] Ann L. Drapeau, Ken Shirriff, John H. Hartman, Ethan L. Miller, Srinivasan Seshan, Randy H. Katz, Ken Lutz, David A. Patterson, Edward K. Lee, Peter M. Chen, and Garth A. Gibson, “RAID-II: A High-Bandwidth Network File Server”, *Proceedings of the 21st Annual International Symposium of Computer Architecture*, April 1994.

[Guy90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier, “Implementation of the Ficus Replicated File System”, *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, June 1990, 63-71.

[Hagmann87] Robert Hagmann, “Reimplementing the Cedar File System Using Logging and Group Commit”, *Proceedings of the 131h Symposium on Operating Systems Principles (SOSP)*, November, 1987, 155-162. Published as *ACM SIGOPS Operating Systems Review 2*1, 5.

[Hartman93] John H. Hartman and John K. Ousterhout, Letter to the Editor, *ACM SIGOPS Operating Systems Review 27*, 1 (January 1993), 7-10.

[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, 49-54.

[Hitz94] Dave Hitz, James Lau, and Michael Malcolm, “File System Design for an NFS File Server Appliance”, *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, CA, January 1994, 235-246.

[Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West, “Scale and Performance in a Distributed File System”, *ACM Transactions on Computer Systems 6*, 1 (February 1988), 51-81.

[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, Edward R. Zayas, “DEcorum File System Architectural Overview”, *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, June 1990, 151-163.

[Koch87] Philip D. L. Koch, “Disk File Allocation Based on the Buddy System”, *ACM Transactions on Computer Systems 4*, 5 (1987), 32-370.

[Liskov91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams, “Replication in the Harp File System”, *Proceedings of the 13th Symposium on Operating Systems Principles (SOSP)*, Asilomar, CA, October 1991, 226-238. Published as *ACM SIGOPS Operating Systems Review 25*, 5.

[Long94] Darrell D. E. Long, Bruce R. Montague, and Luis-Felipe Cabrera, "Swift/RAID: A Distributed RAID System", *Computing Systems 7*, 3 (Summer 1994), 333-359.

[Lo Verso93] Susan J. Lo Verso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler, "*sfs*: A Parallel File System for the CM-5", *Proceedings of the Summer 1993 USENIX Conference*, Cincinnati, OH, June 1993, 291-305.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems 2*, 3 (August 1984), 181-197.

[McVoy91] Larry W. McVoy and Steve R. Kleiman, "Extent-like Performance from a UNIX File System", *Proceedings of the Winter 1991 USENIX Conference*, Dallas, TX, January 1991, 33-43.

[Moran90] J. Moran, R. Sandberg, D. Coleman, J. Kepecs, and B. Lyon, "Breaking Through the NFS Performance Barrier", Proceedings of EUUG Spring 1990, Munich, Germany, April 1990, 199-206.

[Nelson93] Bruce Nelson and Raphael Frommer, "An Overview of Functional Multiprocessing for NFS Network Servers", Technical Report 1, Seventh Edition, Auspex Systems Inc., October 1993.

[Nelson88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout, "Caching in the Sprite Network File System", *ACM Transactions on Computer Systems 6*, 1 (February 1988), 134-154.

[Ousterhout88] John Ousterhout, Andrew Cherenson, Fred Douglis, Mike Nelson, and Brent Welch, "The Sprite Network Operating System", *IEEE Computer 21*, 2 (February 1988), 23-36.

[Ousterhout90] John Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast As Hardware?", *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, June 1990, 247-256.

[Page91] Thomas W. Page, Jr., Richard G. Guy, John S. Heidemann, Gerald J. Popek, Wai Mak, and Dieter Rothmeier, "Management of Replicated Volume Location Data in the Ficus Replicated File System", *Proceedings of the Summer 1991 USENIX Conference*, Nashville, TN, June 1991, 17-29.

[Patterson88] David A. Patterson, Garth Gibson, and Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD),* Chicago, IL, June 1988, 109-116.

[Pierce89] Paul Pierce, "A Concurrent File System for a Highly Parallel Mass Storage Subsystem", *Proceedings of the Fourth Conference on Hypercubes*, Monterey CA, March 1989.

[Rosenblum91] Mendel Rosenblum and John K. Ousterhout, "The Design and Implementation of a Log-Structured File System", *Proceedings of the 13th Symposium on Operating Systems Principles (SOSP),* Asilomar, CA, October 1991, 1-15. Published as *ACM SIGOPS Operating Systems Review 25*, 5.

[Rosenblum92] Mendel Rosenblum, “The Design and Implementation of a Log-structured File System”, Ph.D. Thesis, Computer Science Division, University of California, Berkeley, June 1992. Also available as Technical Report UCB/CSD 92/696.

[Ruemmler93] Chis Ruemmler and John Wilkes, “UNIX disk access patterns”, *Proceedings of the Winter 1993 USENIX Conference*, San Diego, CA, January 1993, 405-420.

[Sandberg85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon, “Design and Implementation of the Sun Network Filesystem”, *Proceedings of the Summer 1985 USENIX Conference*, Portland, OR, June 1985, 119-130.

[Satyanarayanan90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, E. H. Siegel, and D. C. Steere, “Coda: a highly available file system for a distributed workstation environment”, *IEEE Transactions on Computers 39*, 4 (April 1990), 447-459.

[Seltzer90] Margo Seltzer, Peter Chen, and John Ousterhout, “Disk Scheduling Revisited”, *Proceedings of the Winter 1990 USENIX Conference*, January 1990, 313-324.

[Seltzer93] 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*, San Diego, CA, January 1993, 307-326.

[Shirriff92] Ken Shirriff and John Ousterhout, “A Trace-driven Analysis of Name and Attribute Caching in a Distributed File System”, *Proceedings of the Winter 1992 USENIX Conference*, January 1992, 315-331.

[Siegel90] Alex Siegel, Kenneth Birman, and Keith Marzullo, “Deceit: A Flexible Distributed File System”, *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, June 1990, 51-61.

[van Renesse88] Robbert van Renesse, Andrew S. Tanenbaum, and Annita Wilschut, “The Design of a High-Performance File Server”, IR-178, Vrije Universiteit Amsterdam, November 1988.

[Walker83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel, “The LOCUS Distributed Operating System”, *Proceedings of the 9th Symposium on Operating Systems Principles (SOSP)*, November 1983, 49-70. Published as *ACM SIGOPS Operating Systems Review 17*, 5.

[Welch86] Brent B. Welch, “The Sprite Remote Procedure Call System”, Technical Report UCB/CSD 86/302, Computer Science Division, University of California, Berkeley, June 1986.

[Wilkes89] John Wilkes, “DataMesh -- Scope and Objectives: A Commentary”, Technical Report HPL-DSD-89-44, Hewlett-Packard Company, Palo Alto, CA, July 19 1989.

[Wilkes91] John Wilkes, “DataMesh -- Parallel Storage Systems for the 1990s”, *Proceedings of the Eleventh IEEE Symposium on Mass Storage Systems*, Monterey, CA, October 1991, 131-136.