Zebra: A Striped Network File System

John H. Hartman
John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

This paper presents the design of Zebra, a striped network file system. Zebra applies ideas from log-structured file system (LFS) and RAID research to network file systems, resulting in a network file system that has scalable performance, uses its servers efficiently even when its applications are using small files, and provides high availability. Zebra stripes file data across multiple servers, so that the file transfer rate is not limited by the performance of a single server. High availability is achieved by maintaining parity information for the file system. If a server fails its contents can be reconstructed using the contents of the remaining servers and the parity information. Zebra differs from existing striped file systems in the way it stripes file data: Zebra does not stripe on a per-file basis; instead it stripes the stream of bytes written by each client. Clients write to the servers in units called *stripe fragments*, which are analogous to segments in an LFS. Stripe fragments contain file blocks that were written recently, without regard to which file they belong. This method of striping has numerous advantages over per-file striping, including increased server efficiency, efficient parity computation, and elimination of parity update.

This paper will appear in the proceedings of the USENIX Workshop on File Systems, May 1992.

This work was supported in part by the National Science Foundation under grant CCR-8900029, the National Aeronautics and Space Administration and the Defense Advanced Research Projects Agency under contract NAG2-591.

1 Introduction

Zebra is a network file system architecture designed to provide both high performance and high availability. This is accomplished by incorporating ideas from log-structured file systems, such as Sprite LFS [Rosenblum91], and redundant arrays of inexpensive disks (RAID) [Patterson88] into a network file system. From log-structured file systems Zebra borrows the idea that small, independent writes to the storage subsystem can be batched together into large sequential writes, thus improving the storage subsystem's write performance. RAID research has focused on using striping and parity to obtain high performance and high availability from arrays of relatively low-performance disks. Zebra uses striping and parity as well, resulting in a network file system that stripes data across multiple storage servers, uses parity to provide high availability, and transfers file data between the clients and the storage servers in large units. The notable features of Zebra can be characterized as follows:

*Scalable performanc*e. A file in Zebra may be striped across several storage servers, allowing its contents to be transferred in parallel. Thus the aggregate file transfer bandwidth can exceed the bandwidth capabilities of a single server.

High server efficiency. Storage servers are most efficient handling large data transfers because small transfers have high overheads. Large transfers are relatively simple to achieve for large files, but small files pose a problem. Client file caches are effective at reducing server accesses for small file reads, but they aren't as effective at filtering out small file writes [Baker91]. Zebra clients use the storage servers efficiently by writing to them in large transfers, even if their applications are writing small files.

High availability[1]. Zebra can tolerate the loss of any single machine in the system, including a storage server. Zebra makes file data highly available by maintaining the parity of the file system contents. If a server crashes its contents can be reconstructed using the parity information. The use of parity allows Zebra to provide the availability of a system that maintains redundant copies of its files while requiring only a fraction of the storage overhead.

Uniform server loads. File striping causes the load incurred by a heavily used (*hot*) file to be shared by all of the storage servers that store the file. In a traditional network file system a hot file only affects the performance of the server that stores it, requiring that hot files be carefully distributed among all of the servers to balance the load.

Zebra is currently only a paper design, although a prototype is being implemented in the Sprite operating system [Ousterhout88]. This paper describes the design of Zebra, not the prototype implementation. The rest of this paper is organized as follows. Section 2 discusses striping and its application to a network file system, Section 3 discusses the use of parity to provide high availability, Section 4 gives an overview of the Zebra architecture, and Section 5 describes the Zebra design in more detail. Section 6 covers Zebra's status and future work, and Section 7 is a conclusion.

1. The distinction between availability and reliability, while it is important, is not particularly relevant to this paper. The arguments made here regarding the availability of Zebra also apply to its reliability.

2 Why Stripe?

Traditional network file systems confine each file to a single file server. Unfortunately this means that the rate at which a file can be transferred between the server and a client is limited by the performance characteristics of that one server, such as its CPU power, its memory bandwidth and the performance of its I/O controllers. This makes it difficult to improve the performance of the network file system without improving or replacing the server. Striping a file over several servers allows those servers to transfer the file in parallel, so that the aggregate transfer rate can be much higher than that of any one server. The file transfer performance of the file system can be improved simply by adding more servers.

A striped network file system has several economic advantages over a traditional network file system. First, the storage servers do not need to be high-performance, nor do they need to be constructed out of special-purpose hardware. Servers in a traditional network file system are often among the more expensive and high-performance machines in the system. In contrast, storage servers in a striped network file system can be relatively modest machines, thereby improving their cost/performance and reducing the fraction of the total system cost that they represent. Second, a striped network file system allows clients to be upgraded without requiring server upgrades as well. The increased client performance can be matched by increasing the number of servers, rather than replacing them. Both of these considerations make a striped network file system an economically attractive alternative to a traditional network file system.

The idea of using striping to improve data transfer bandwidth is not a new one. It's often used to improve the performance of disk subsystems by striping data across multiple disks attached to the same computer. Mainframes and supercomputers have used striped disks for quite a while [Johnson84]. The term *disk striping* was first defined by Salem and Garcia-Molina in 1986 [Salem86]. More recently there has been lots of interest in arrays of many small disks, originating with the paper by Patterson et al. in 1988 [Patterson88]. All of this work focuses on aggregating several relatively slow disks to create a single logical disk with a much higher data rate.

In recent years striping has been applied to file systems as a whole. In these file systems the blocks of each file are striped across multiple storage devices. These storage devices may be disks, as in HPFS [Poston88], I/O nodes in a parallel computer, as in CFS [Pierce89] and Bridge [Dibble90], or they may be network file servers as in Swift [Cabrera91]. It is important to note that these systems stripe on a per-file basis, therefore they work best with large files. Small files are a kind of Catch-22: if they span all the storage devices then the amount stored on each device will be small, causing the devices to be used inefficiently, but if small files aren't striped then the system performance when writing small files will be no better than that of a non-striped system. Applications that write many small files will not see a performance improvement.

Striping also serves as a load-balancing mechanism for the storage devices. Ideally the storage devices would have identical loads, so that no one device saturates and becomes a bottleneck. If files are constrained to a single storage device then care must be taken to ensure that hot files are evenly distributed across the devices. Striping eliminates the need for careful file placement by distributing files over multiple devices. The load caused by a heavily used file is shared by all the devices that store it, thus reducing the variance in device loads.

3 Availability

Striping can potentially reduce the availability of a network file system, since each file in the system is distributed over several storage servers. The loss of any one of these servers will cause the file to be unavailable. If a striped file system with multiple servers replaces a file system with a single server, then the availability of the system will be reduced (assuming the servers in both systems have the same failure rate).

Network file systems often improve availability by maintaining redundant copies of each file. Redundant copies are advantageous because they allow the system to withstand server failures (provided at least one copy remains available), and they can allow the system to tolerate network partitions. If each section of the partitioned network contains a copy of each file then files can continue to be accessed without interruption. Redundant copies do have disadvantages, however. Additional storage is needed for the extra copies, and there are complexities and overheads involved in keeping the copies up-to-date.

An alternative approach is to maintain error-correcting information that allows missing data to be reconstructed. RAID systems favor this scheme. For example, a simple parity scheme will allow the system to tolerate the loss of a single server. One block of data from all but one server is XOR'ed together to produce a parity block which is stored on the remaining server. Should one of the blocks of data become unavailable it can easily be recomputed from the other blocks of data and the parity block. The advantage of this approach is that the storage required for the parity blocks is much less than is needed for redundant copies. Swift proposes to use parity to tolerate server failures for just this reason.

It is easier to implement a parity mechanism for a RAID storage system than for a network file system, however. A RAID is usually connected to a host computer, through which all data transfers to and from the array must pass. This makes the host a convenient location to compute parity. A network file system doesn't have a comparable centralized location, however. This makes it more difficult to compute parity across physical storage blocks or file blocks. If files are written randomly, or written by multiple clients simultaneously, then no single location may contain all of the data needed to perform the parity calculation. File updates are also problematic, since they require updating the parity of the modified blocks as well. The new parity must be computed from the old and new contents of the block, potentially causing several server accesses per update. In addition, the update of a block and its parity must be an atomic operation, since data could be lost if the two get out of sync. Ensuring that two writes to two different servers are atomic is likely to be complex and expensive.

4 Zebra Stripes

Zebra differs from existing striped network file systems in that it does not stripe on a per-file basis. Instead it stripes on a per-client basis: all of the new data from a single client is formed into a single logical stream, regardless of which files the data belongs to. This

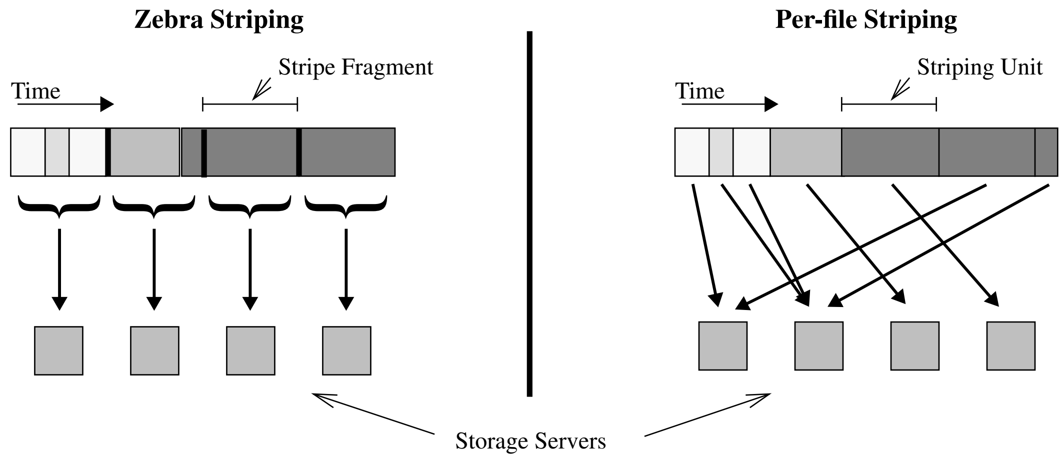

Figure 1: **Zebra striping vs. per-file striping.** This figure shows the same sequence of file data being written in both Zebra and in per-file striping. Each shaded region represents a piece of data written to a single file. Regions with the same shading belong to the same file. Zebra clients pack together file blocks and write them to the storage servers in large transfers called *stripe fragments.* Per-file striping requires more transfers, because small writes aren't batched. In this example the striping unit (the maximum amount of data written to a single server) in the per-file system is the same size as a stripe fragment in Zebra. Parity writes are not shown.

stream is then striped across the storage servers. The data written to the servers in a single pass by a single client is called a *stripe*. The portion of a stripe that is written to each server is called a *stripe fragment.* Clients compute the parity of the stripe fragments as they are written. At the end of a stripe the client writes out the resulting parity fragment and begins a new parity computation. Note that each stripe is written by a single client. Multiple clients may be writing to the storage servers simultaneously, but they are all writing to distinct stripes. Figure 1 illustrates the Zebra striping mechanism.

Zebra, like all file systems, must maintain *metadata* that records the physical storage location for each file block. When a file block is accessed the file's metadata is used to determine which storage location to use. In Zebra the *file manager* is responsible for managing the metadata (See Section 5.2 for details). Once a client has written a stripe it sends a summary of the stripe's contents to the file manager, so that the metadata can be updated accordingly.

Clients are responsible for reconstructing files from their constituent stripe fragments. Upon opening a file for reading the client obtains the metadata for the file from the file manager. To read the file the client uses the metadata to determine which stripe fragments to access, then retrieves the desired portions of those fragments from the storage servers and reassembles them into the file.

Stripes in Zebra are analogous to segments in a log-structured file system. They are large conglomerations of file data that can be efficiently transferred. The data stored in a stripe exhibits temporal locality, rather than spatial locality, i.e. a stripe contains blocks that were written during the same period of time rather than blocks from the same file.

Like segments, stripes are immutable objects. Once they are written they cannot be modified. Free space is reclaimed from stripes by *cleaning* them (see Section 5.3).

Zebra's method of striping has several advantages over per-file striping. First, the striping algorithm is relatively simple. No special mechanisms are needed for handling small files, randomly written files, or file updates; their bytes are simply merged into the client's stream of bytes. Large files that are sequentially written will be striped in a manner similar to per-file striping, however. Each file will be divided into stripe fragments and striped across the servers. Second, parity computation and management is simplified. Parity is easily computed because each client computes the parity of the stripes it produces. The parity computation doesn't require any coordination between clients, or additional data transfers between the servers and the clients. Since stripes are immutable and parity is never updated, Zebra also avoids having to atomically update a file and its parity. A simple timestamp mechanism is sufficient for ensuring that a stripe and its parity are consistent. If a client should crash while in the process of writing a stripe the timestamps are used to determine how many of the stripe fragments were actually written prior to the crash, so that the stripe's parity can be updated accordingly.

Striping the logical stream of bytes from each client, rather than files, improves the performance of writing small files. Clients write stripe fragments to the storage servers, not files or file blocks. This decouples the size of the files used by the client applications from the size of the accesses made to the storage servers. Applications that write many small files will see a performance improvement over traditional network file systems because the files will be batched together and written to the server in large transfers.

5 Zebra Design

Figure 2 illustrates the components of Zebra. The storage servers store file data, in the form of stripe fragments. The file manager manages the file name space and keeps track of where files are located. The stripe manager handles storage management by reclaiming the space occupied by data that is no longer used. The rest of this section describes these components in more detail.

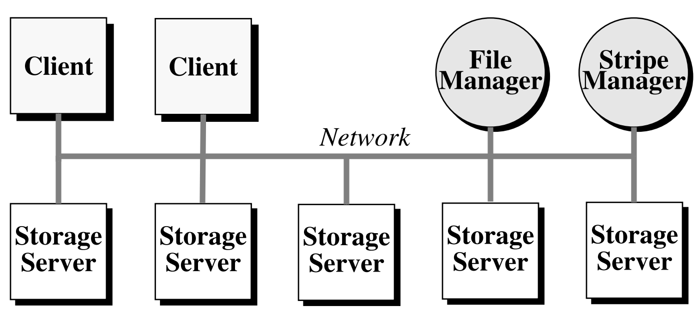

Figure 2: Zebra schematic. Squares represent individual machines; circles represent logical entities. The file manager and the stripe manager can run on any machine in the system, although it is likely that one machine will be designated to run both of them.

5.1 Zebra Storage Servers

The Zebra storage servers are merely repositories for stripe fragments. They create new fragments in response to client requests and retain the fragments until they are deleted by the stripe manager. The fragments are opaque to the storage servers -- the servers know nothing about the files or blocks that the fragments contain. This simple functionality makes it possible to implement the storage server in a variety of different ways. One option is to build a special-purpose storage server that has been optimized for storing stripe fragments. Another is to store stripe fragments as files in a local file system. This approach is not only easy to implement but it also allows the storage servers to be traditional network file servers, some of whose files happen to be Zebra stripe fragments.

The simple functionality of the storage servers is well-suited to machines that emphasize I/O capabilities rather than CPU speed. For example, the RAID-II project at Berkeley [Lee92] is building a storage system that has a high-bandwidth connection between its disk array and the network. Unfortunately, it is relatively expensive for the host CPU to access the data that passes over that connection. Traditional network file systems are likely to run slowly on such an architecture since the host must process each file block. A Zebra storage server, on the other hand, only performs per-fragment processing and is better able to take advantage of this type of architecture.

5.2 Zebra File Manager

The Zebra file manager manages the file system metadata, i.e. the file system name space and the mapping of logical file blocks into stripe fragments. Clients send all name space modifications, such as file creation and deletion, as well as file open and close events to the file manager. This allows the file manager to ensure the consistency of the metadata even if clients are modifying it concurrently. The file manager is a critical resource; the file system cannot be accessed if its metadata is unavailable. Zebra employs a backup file manager that can take the place of the primary file manager should the primary fail. During normal operation the primary file manager logs all changes in the metadata to the backup. If the primary should fail the backup uses this information to reconstruct the current state of the metadata.

The modification rate of the metadata in Zebra is likely to be higher than in traditional file systems. This is because the mapping of a file block to a stripe fragment changes when the file block is modified. Clients determine the storage location for a file block by packing it into a stripe fragment and writing the fragment to the next server in its rotation. If a client overwrites an existing file block then the new version of the block will probably be stored in a different fragment. When a client writes a stripe it must tell the file manager which file blocks it contains so the file manager can update its metadata.

The centralization of name service and file mapping information on the file manager is a potential performance bottleneck. One technique for eliminating this bottleneck is to have the clients cache both name and mapping information. Client name caching has been shown to be effective at reducing the load on the name server in a network file system [Shirriff92]. By caching whole directories of file names and their mapping information the Zebra clients can eliminate the need for contacting the file manager each time they modify the name space or access a file.

5.3 Stripe Manager

The Zebra stripe manager is responsible for managing the storage space on the storage servers by reclaiming free space from existing stripes. Its function is similar to the *segment cleaner* in a log-structured file system. The manager keeps a list of all stripes in the system, and which file blocks they contain. As blocks are deleted or overwritten the list is updated accordingly. When free space is needed a stripe is *cleaned*[2], a process in which its live data is read and then written to a new stripe. The storage servers are then notified that the space currently allocated to the cleaned stripe can be reused. The stripe manager represents a centralized location in which the live data from stripes that are cleaned can be formed into stripe fragments and their parity computed. Zebra uses a backup stripe manager to ensure that the stripe manager is always available.

Cleaning's impact on system performance is proportional to the amount of live data in the stripes that are chosen to be cleaned. Ideally the stripes would not contain any live data, so that cleaning them would not cause any data transfers. Measurements of Sprite LFS show that the write traffic to the disk due to cleaning is relatively low (for non-swap file systems it is between 2% and 7% of the overall write traffic to the disk) [Rosenblum91]. Further research is required to determine if Zebra exhibits the same behavior.

6 Status and Future Work

Zebra is currently a paper design. Implementation of a prototype began in the spring of 1992, and should be completed by late 1992. Once the prototype is completed it will be measured and compared to existing network file systems in the following ways:

- Performance under a variety of workloads, each of which has a different distribution of file sizes and read/write ratios. The emphasis will be on Zebra's performance running workloads with lots of large files (supercomputer workload), and workloads with lots of small files (UNIX workload). The workloads will probably be synthetic.
- Parity's cost, in terms of CPU cycles, network bandwidth, and storage server resources.
- Stripe cleaning's impact on performance.
- Tolerance of failures of the storage servers, the file manager, the stripe manager, and the clients.

7 Conclusion

Zebra applies ideas from log-structured file system research and RAID research to network file systems, resulting in a system that has the following advantages over existing network file systems:

Scalable performance. The transfer rate for a single file is proportional to the number

2. The algorithm for choosing which stripe to clean is beyond the scope of this paper.

of servers in the system.

Cost effective servers. Zebra servers do not need to be high-performance machines or have special-purpose hardware. The performance of the file system can be increased by adding more servers, rather than replacing the existing ones.

High server efficiency. Server overhead is reduced because clients write to the storage servers in large transfers, and the servers do not interpret the contents of the stripe fragments they store. There are no per-file or per-block server overheads associated with writing a stripe fragment to a storage server.

Simple parity mechanism. Parity is computed by the clients as they write out stripe fragments. Parity is never updated, so expensive parity update computations and atomic operations are not needed.

Uniform server loads. Striping reduces the variance of the server loads by distributing hot files across multiple storage servers.

The Zebra architecture promises to provide high-performance file access to both large and small files. Large files are striped to improve their transfer rate; small file writes are batched together to reduce server overheads. The result is a cost-effective, scalable, highly-available network file system that can provide file service to a supercomputer as easily as to a workstation.

8 Acknowledgments

Tom Anderson provided helpful comments on drafts of this paper. Alan Smith suggested the name “Zebra”.

9 References

[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.

[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.

[Dibble90] Peter C. Dibble, “A Parallel Interleaved File System”, Ph.D. Thesis, University of Rochester, 1990.

[Johnson84] O. G. Johnson, “Three-dimensional wave equation computations on vector computers”, *Proceedings of the IEEE 72* (January 1984).

[Lee92] Edward K. Lee, Peter M. Chen, John H. Hartman, Ann L. Chervenak Drapeau, Ethan L. Miller, Randy H. Katz, Garth A. Gibson and David A. Patterson, “RAID-II: A Scalable Storage Architecture for High Bandwidth Network File Service”, Technical Report UCB/CSD 92/672, Computer Science Division, Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, February 1992.

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

[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.

[Poston88] Alan Poston "A High Performance File System for UNIX", NASA NAS document, June 1988.

[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.

[Salem86] Kenneth Salem and Hector Garcia-Molina, "Disk Striping", *Proceedings of the 2nd International Conference on Data Engineering*, February 1986, 336-342.

[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*, San Francisco, CA, January 1992, 315- 331.