Striping the SFSRO file system for performance and flexibility

Frank Dabek and Eric Peterson

December, 2000

Abstract

The SFSRO file system can be extended to a striped configuration to achieve aggregate bandwidth which scales well with the number of machines dedicated to the system. We demonstrate the design and implementation of prototype system which achieves an estimated aggregate throughput of 45.4 MB/s using 64 machines.

1 Introduction

SFSRO[FKM00] provides integrity-guaranteed delivery of read-only content. Read-only data is plentiful and often in high demand (e.g. RedHat distributions, popular static webpages). While SFSRO's design allows as single server to service a large number of requests the total peak throughput of the current design is fundamentally limited by that machine's network link. The goal of this project is to eliminate that bottleneck by striping data across a number of SFSRO servers. The desired result of this collection of cooperating servers is an aggregate throughput from n servers to a large number of clients that scales well with n.

SSFSRO (Striped SFS Read-Only file system) is intended to be used in situations where such a large number of clients wish to access an SFSRO file system that the server's connection to the Internet becomes a bottleneck ([FKM00] demonstrated that the SFSRO server is capable of saturating a 100Mb/s link). By distributing load among a number of cooperating servers, SSFSRO distributes bandwidth load across a number of links, eliminating the bandwidth bottleneck near the server.

The flexible design of this project also suits it for deployment as a way for a large number of servers with limited disk and/or bandwidth resources to serve a large volume of content at high data rates.

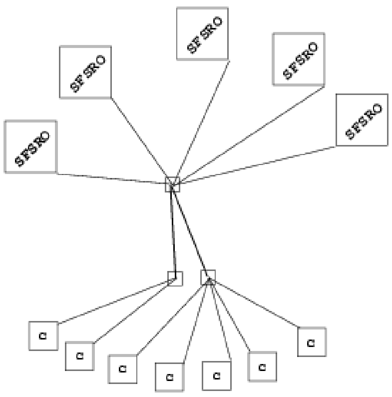

Figure 1: **Overview:** By striping data across multiple servers, the single link that a server is connected to is no longer a bottleneck.

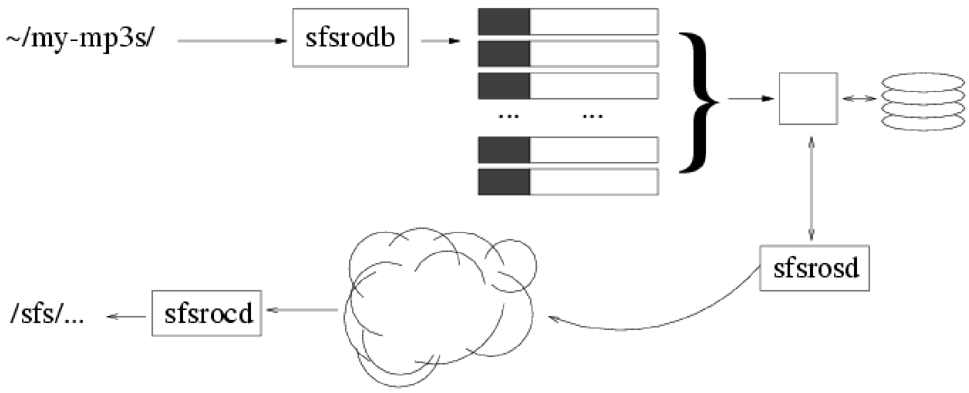

Figure 2: **SFSRO Operation:** An SFSRO file system is created by the program sfsrodb which converts a directory into a database. A server (sfsrosd) answers requests for items from the client program (sfsrocd) which contacts the server via a network interface. sfsrocd runs on the users machine and mounts the remote file system under `/sfs`

2 Related Work

This project builds on the SFS file system discussed in [MKKW99]. SFS provides a secure file system which isolates key management through the use of self-certifying pathnames. A self-certifying pathname might look like `/sfs/eb.lcs.mit.edu:yftnuuy...qqbdgwuh/RedHat/i386/`. This pathname contains the host serving the file system (eb.lcs.mit.edu) and a string of characters which is sufficient to verify that a public key presented by the file server matches the public key intended by this path. The remainder of the path follows standard UNIX semantics for file naming. The interesting property of the SFS naming scheme is that if we laboriously type in this pathname and the server returns a public key that the pathname verifies, then we have no doubt that we are accessing the file system we intended: server authentication is explicit in an SFS pathname.

While SFS provided read-write access and client authentication, SFSRO[FKM00] is optimized for read-only access. SFSRO allows data (whole file systems) to be replicated on untrusted mirrors and provides a guarantee of integrity by appending a cryptographic hash to each block. In addition, each file contains a hash that encompasses all blocks in the file and each directory a hash which can be used to verify the entire directory. Thus, by examining the hash of the root directory a user can quickly verify that the file system has not been modified.

Although SFSRO was designed to allow replication, it is not, by default, well suited for cooperative mirroring. SFSRO provides only for manual distribution of complete databases. It does not provide an easy path for a client to learn the identity of other mirrors, or for sites who wish to become mirrors to do so.

The idea of striping a file system across multiple servers is presented in xFS [TEA95] and Zebra [HO93]. These file systems structure data as a transaction log and stripe the log across multiple servers. The systems achieved aggregate throughput performance which scaled with the number of machines acting as servers and exceeded the performance of a single traditional server. From this approach we take the idea that a single server imposes its computational, bandwidth, and memory throughput limits on the maximum throughput of a network file system.

When designing SSFSRO we also had in mind the proliferation of peer-to-peer networking systems such as Freenet [Cla99], the Free Haven project [RD00], Publius [MWC00] and gnutella [gnu]. In these projects a large number of participants cooperate to serve content (sometimes with strong guarantees on anonymity). Although we have not deployed SSFSRO in this context, we hope that the design choices we have made (support for server partial file systems, flexible server selection) will make it possible to utilize this project as part of a peer-to-peer networked

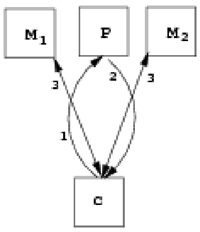

Figure 3: **Messages in SSFSRO:** Clients first contact the publisher who returns a list of mirrors. Future requests are sent to those mirrors rather than the publisher.

file system.

3 Design

The design goals for SSFSRO were flexibility and performance. We evaluated these goals in terms of several main issues: fragmentation granularity, server selection, redundancy, and database creating.

The overall structure of SSFSRO is a primary-mirror organization. One server is designated as the publisher and maintains a complete copy of the file system to be served. Other servers, known as mirrors, obtain copies of the file system (complete or partial) from the publisher. Aside from distributing file system data, the publisher is responsible for accepting the first request from every client and referring them to a list of mirrors.

The issue of granularity arises when we decide how to divide a file system among cooperating mirrors. The original SFSRO design divides file systems at the level of a whole file system: a replica contains either nothing or all data necessary to service any request. Our initial design for this project used a block size inherent in SFSRO as a division. When SFSRO stores a file system, it does so by dividing the file system into blocks and storing those blocks in a database keyed by a hash of that block. These blocks seemed like a logical division of the file system among mirrors. In this scheme each mirror would be responsible for the blocks corresponding to keys in a given partition of the range of the hash function. Dividing the database in this way offers good fragmentation (each data block in a file is likely to map to a different region of the hash space and thus to a different mirror) and was easy to implement. However, if certain blocks become highly popular (for instance, blocks containing directory information for top-level directories) this scheme would distribute load in an unbalanced manner. Replicating popular blocks among multiple servers is one solution to the problem of unbalanced load.

To alleviate the problems of unbalanced load due to popular blocks, our final design divides each block among a number of mirrors. Every item stored by SSFSRO in its database is considered as partitioned into 64 sub-blocks which are in turn distributed to any number of mirrors. A client that wishes to obtain a block must obtain (at least) one copy of each sub-block and assemble them. This scheme is slightly more complicated (it required the implementation of a new RPC) but it offers ideal load balancing. An SFSRO client has no way of requesting an amount of data smaller than a block; all mirrors are therefore involved in the handling of a request for any block. This approach also accords us the opportunity to dynamically vary the request size for individual mirrors: a slower mirror could be asked for a smaller percentage of a block than a faster mirror in proportion to its available bandwidth.

Before considering the question of server selection we provide a path for clients to obtain a list of mirrors. In the unmodified SFSRO system, when a client initially contacts a server a `fsinfo` structure is returned containing information about the file system such as expiration time, public key, etc. We augment this structure with information about the number of mirrors and the fraction of the database they are capable of serving. Thus, when a client connects to the publisher it receives a list of potential mirrors. In keeping with the SFSRO philosophy of

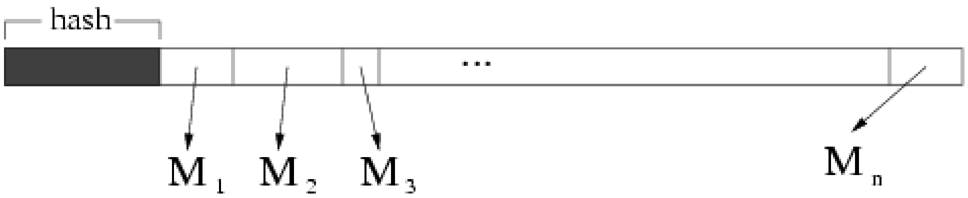

Figure 4: **block division:** Each block is obtained by requesting slices from one or more mirrors.

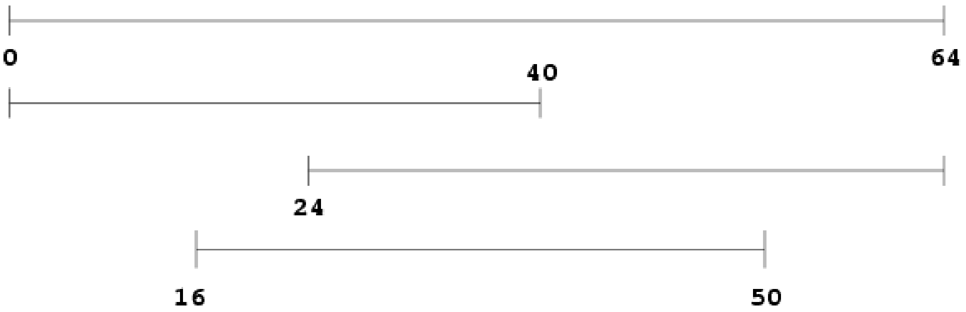

Figure 5: **server selection:** The problem of server selection can be thought of as choosing a mirror for each of 64 slices from the regions each mirror serves.

placing as little intelligence in the server as possible, the server makes no attempt to verify the status, load or even legitimacy of the mirrors it returns to clients. This feature seems to offer the opportunity for a denial of service attack: a malicious user could populate a server's mirror list with incorrect data. In future versions we may force mirrors who wish to register with the publisher to solve a client puzzle.

The choice of dividing each block among mirrors also provides for flexible server selection. The client is provided with a list or ranges offered by each mirror. It is free to choose how it distributes the load of its requests based on any number of parameters: server load, locality or availability.

The client is also solely responsible for recovering from servers which are or become unavailable. Because this system is being used to serve read-only data, complicated schemes for maintaining consistency (such as in [BL99] [HC00]) are not necessary. If a client detects that a mirror is down, it simply removes that mirror from its list and recomputes a tiling of the block space using the other mirrors available. Such a tiling is always possible since, the publisher, as an invariant, holds the entire file system.

4 Implementation

The prototype implementation described here required the modification of the RPC interface exported by the read-only server sfsrosd, the server itself, as well as the SFSRO client, sfsrocd. Additionally, small changes were made to the programs used to generate databases and to pull them from machine to machine.

These changes were made with compatibility with the single server model in mind but do not maintain backwards compatibility with existing binaries. Existing clients will not be able to communicate with our server (mainly because they will not be able to decode the modified fsinfo record.) We do not see this as a problem, however, since, to the best of our knowledge, there are no deployed SFSRO servers.

To implement block striping we add a new RPC SFSPROC_GETPARTIALDATA which is analogous to the existing SFSPROC_GETDATA except that it takes additional arguments specifying an offset and a fraction within the block. The other protocol change we make is to the fsinfo record. To this record we add an unsigned list of mirrors and the portion of the database which they are mirroring. We added a new RPC rather than replacing the existing GET_DATA RPC so that clients who wish to interact under the single server model may do so without modification.

On the client side, we modified sfsrocd to examine the list of mirrors returned in the fsinfo structure and map a region of the database range to each mirror. The algorithm that does this mapping is currently quite simple and is one area which can be improved substantially. The modified client then requests a region of the database from each mirror, assembles them and hands off the assembled block for further processing.

Modifications to sfsrodb were minimal: after a database is created the additional fields in its fsinfo structure are modified to include information about the range of blocks stored. pulldb was also modified to allow it to create a partial database and to pull a database that does not exist locally from a remote machine.

5 Performance

Measuring the performance of a distributed system such as this one is difficult given the limited experimental setup available. Under some reasonable assumptions, however, we may estimate performance of the entire system using only a few machines. The goal of these tests is to determine aggregate throughput from a group of servers. Client performance issues are not addressed: in conducting these tests we assume that a large number of clients attempt to access the servers and that their computational or bandwidth limitations do not form a bottleneck for server performance. In addition, we assume the independence of each mirror's network resources.

These tests were designed to test the hypothesis that n mirrors could achieve an aggregate throughput that scaled with n. The experiment was conducted using two machines: each was powered by a 550 Mhz Pentium III CPU and contained 256 MB of RAM and a SCSI hard disk. The machines were connected via 100Mb/s switched Ethernet.

The file system used for the test consisted of 44 files ranging from 30 to 330 kilobytes in size. The trace for each client was generated by first listing all the files and then sequentially reading them. In each experiment a trace was collected for single mirror serving a varying fraction of each block. A testing program simulated 100 concurrent clients and recorded bandwidth over a 25 second period. To estimate total aggregate bandwidth, this bandwidth was scaled by the inverse of the fraction of the database the mirror was serving. For example, if a mirror serving $\frac{1}{2}$ of the database achieved a bandwidth of 10 MB/s by our independence assumption we estimate that two such mirrors could achieve a combined bandwidth of 20 MB/s.

In the first experiment (labeled "Full Database") the mirror held a complete copy of the database (independent of what fraction it was serving). In the second experiment (labeled "Partial Database") pulldb was used to create a partial database which contained only the information the mirror was responsible for.

In both cases aggregate bandwidth increased with the number of mirrors added to the system. However, as more mirrors were added the rate of increase in performance dropped off. Beyond 16 mirrors, the reward for adding more mirrors diminishes. This diminishing return is most likely attributable to increased protocol overhead. For example, when using 64 mirrors, the largest request is only 128 bytes. Overhead due to TCP, RPC, and XDR is a significant fraction of this data size. Fixed per request processing costs are also a limiting factor: saturating a 100Mb/s ethernet at this packet size would require the server to process 86,000 requests per second. However, small requests are not an inherent weakness of the partial request striping approach. In order to compensate for the smaller size of fractional requests, the block size of the database could be increased appropriately. For example, we would expect that the efficient scaling exhibited by 4 and 8 mirror configurations could be achieved with 16 and 32 configurations by merely quadrupling the database block size. Unfortunately, we were unable to test different block sizes since it was not a parameter of the existing system.

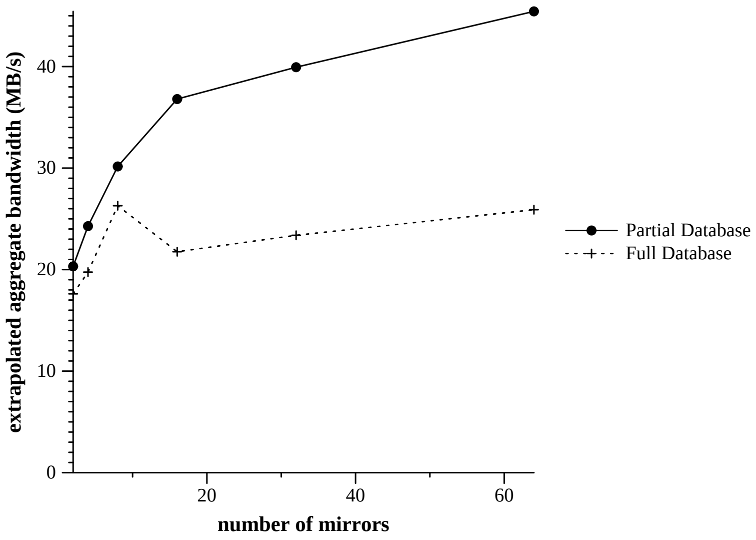

Figure 6: **Performance** of the striped SFSRO file system with a varying number of mirrors. The solid line indicates performance with a partial copy of the database, the dotted line indicates performance with a total copy of the database (including sections not served by the tested mirror).

The second experiment reveals the benefits of producing partial databases. The partial database test scaled much better than the full database test. This increased scaling is likely due to a combination of better caching performance and more efficient use of the database. In the full database test, an entire block was fetched for each request even though only a fraction of the data contained in the block was returned to the client. This additional overhead is avoided in the partial database case.

6 Conclusion and Future Work

This project demonstrates the feasibility of using striping to improve the performance of the SFSRO system. The system disappointingly produced diminishing returns as the number of mirrors was increased, however. This result signals the need to reevaluate our data division strategy. Future iterations of this project will focus on an improving performance by lowering the fixed cost of each request, dynamically updating the load placed on each mirror via a feedback system, and improving communication between mirrors and the publisher to enable mirrors to more efficiently choose which sections of the file system to replicate.

7 References

References

[BL99] Yasushi Saito Brian N. Bershad and Henry M. Levy. Manageability, availability, and performance in porcupine: a highly scalable, cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, South Carolina, December 1999.

[Cla99] Ian Clarke. *A Distributed Decentralised Information Storage and Retrieval System.* PhD thesis, University of Edinburgh, 1999.

[FKM00] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating*

Systems Design and Implementation (OSDI 2000), San Diego, California, October 2000.

[gnu] gnutella website. http://gnutella.wego.com.

[HC00] Steven D. Gribble Eric A. Brewer Joseph M. Hellerstein and David Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, October 2000.

[HO93] John H. Hartman and John K. Ousterhout. The zebra striped network file system. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, December 1993.

[MKKW99] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, South Carolina, December 1999.

[MWC00] Aviel D. Rubin Marc Waldman and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[RD00] Michael J. Freedman Roger Dingledine, David Molnar. The free haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[TEA95] Jeanna M. Neefe et al. Thomas E. Anderson, Michael D. Dahlin. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.