FreeSFS:
Peer-to-Peer Routing for SFS Read-Only

Brian Kim, Virgil King, Matthew Malchano
{kimb, vking, malchano}@mit.edu

December 15, 2000

Abstract

Heavy traffic on the Internet makes distributed mirror sites highly desirable due to their reliability, lower loads, faster response times, and lower costs. Administering, creating, locating, and authenticating mirrors cause difficulities for both providers and users. FreeSFS builds on the SFS Read-Only system, providing an independent routing layer that allows for decentralized lookups of data on SFS replicas. Using FreeSFS, a server may elect to become an authenticated mirror of any content provider, without being required to register with a centralized server. Users can then automatically search for such replica sites.

1 Introduction

The Internet is increasingly becoming the preferred means of digital content distribution. Many tools exist that facilitate publishing information on the web which range from simple http and ftp servers that can run on any machine architecture to high performace distributed databases. Users trying to download popular content, however, often face the problem of swamped servers or network congestion.

A common, but unwieldy, approach to this problem is the use of mirror sites for replicating large quantities of data. Organizations and users often volunteer to donate their bandwidth, storage, and CPU cycles in order to help out their favorite content providers. Mirror sites provide many benefits. They decentralize machine administration. They allow for lower costing bandwidth and servers. They offer better performance to users due to their lighter server loads and lower network traffic. Their network distribution both lowers transfer times and offers improved reliability.

While mirroring may be a cost-effective solution in terms of bandwidth and servers, mirrors can be an administrative nightmare. It is difficult for websites to maintain accurate listings of servers that act as their mirrors. Many of the servers may not be under the control of the original site's adminstrator, making mirrors difficult to oversee. Content providers can not ensure that mirrors provide up-to-date or authentic data. In addition, mirror sites may be unreliable, difficult to locate, and insecure, which discourages users from taking advantage of them.

The FreeSFS system addresses many of these problems. First, by using the SFS Read-Only filesystem as our underlying method of replicating data, the authenticity of the data served can be determined without either trusting the machine serving it or requiring extra offline validation by the users. The use of a digitally signed databases in SFS Read-Only allows read-only data to served from untrusted machines. In addition, strict signature checking in the client software makes the data cryptographically tamper proof[1].

FreeSFS provides a replica site lookup solution which allows end users to locate a SFS read-only replica given a hashed public key and location unique to a content provider, called a HostID. Due to the decentralized nature of FreeSFS, anyone can setup a SFS read-only replica for a site and have traffic routed to them without coordinating with any centralized administration. Utilizing a limited recursive search of the nodes of FreeSFS, a mirror location can be located efficiently by a user. Partial routing tables maintained by nodes help shortcut the inescapable costs of broadcast searches, contributing further to rapid lookup.

2 Related Work

There are several system that provide some but not all of the following features:

- trusted data
- no reliance on trusted servers
- flexible peer-to-peer network architecture
- publisher anonymity
- scalable access to servers

2.0.1 Gnutella

Gnutella is a popular decentralized file-sharing protocol. While it is completely decentralized and allows for nodes entering and leaving, it is inefficient in that it makes use of broadcast searches. A recent article on ZDnet cited that up to 50 percent of the traffic on the Gnutella network, including file transfer, is from searching[2]. Clearly, a more elegant search method is needed for Gnutella to scale to a large user base. Gnutella also makes no effort to guarantee data integrity[3].

2.0.2 Publius

Publius is a system designed for publishing content on the web while avoiding censorship and modification of content by adversaries. The publishing algorithm takes a file and distrubutes it among n webservers, only k of which are required to reconstruct the file. The URL is cryptographically tied to the data such that tampering can by easily detected. In addition, authors can delete and modify their published content using their private keys[4].

Publius provides data integrity but as a centralized system is vulnerable to organized attack. It is also hampered by a 100k maximum file size.

2.0.3 Free Haven

Free Haven is another anonymous storage solution that seeks to achieve:

- anonymity for all parties (including actual machines serving data)
- guaranteed document lifetimes set by publisher and not by server administrator
- flexible network architecture that allows for machines being added and removed
- reputation system for limiting damage done by misbehaving servers

While developers of this system have a basic proof-of-concept implementation, this project is still in its design stages[5].

2.0.4 Intermemory

The Intermemory project seeks to provide highly survivable digital data storage that has a high level of fault tolerance, availability, self-repair (damaged or lost data is replaced through interaction with other servers). One strong selling point of this system is that to the low-level programmer, the system can be treated like a typical filesystem. While this system provides resistance to frequent internet occurrences such as network outages and server failures, it does not address issues such as data integrity and scalability[6].

2.0.5 FreeNet

FreeNet is the system that is most similar to ours in terms of its routing algorithm. FreeNet provides user and publisher anonymity, deniability for storers of information, resistance to attempts to deny access to data, dynamic storage of routing information, and decentralization of network functions. FreeNet makes several usability concessions in exchange for these features: it has no provision for ensuring data integrity, resources cannot be updated (names are assigned to static resources for eternity on a first-come, first-serve basis), and node operators have no control over the content be served from their node[7].

2.0.6 SFS read-only filesystem

SFS read-only file system is a content distribution system which provides secure, scalable access to read-only data. It is unique in that the system security is independent from key distribution infrastructure. In addition, its servers are able to achieve high performance because the cryptographic costs of providing signed data are performed offline and the checks are performed by the clients. This allows for highly scalable access. Data management of content on a SFS read-only filesystem can be achieved through efficient incremental updates. Lastly, all signature checks are performed automatically by the client software. The main drawback to this system, which FreeSFS addresses, is the lack of a decentralized solution for locating SFS read-only replicas by signature key [1].

3 Architecture

The FreeSFS system consists of a collection of nodes connected across a network layer. Each node has knowledge of some set of "neighbors" with whom it can communicate, reminiscent of peer-to-peer systems like Gnutella and FreeNet[3],[7]. These communications allow the nodes to share routing data and pass on requests, traversing the graph of nodes recursively to find a specific server or provider.

Individually, a FreeSFS node maintains zero or more SFS Read-Only databases. These databases mirror some SFS content provider (for example, a software archive, website, or ftp server). Due to SFS's unique self-certifying pathnames, which contain a hashed public key, these mirrors offer automatic authentication of the downloaded data. Given a self-certifying pathname, a user is assured that the owner of the private key corresponding to the public key in the path name has "signed" the files. [1]

The FreeSFS node listens on a port for incoming messages and requests from other nodes and clients. When a message arrives, the system determines if it can satisfy the request locally (for example, if the node contains a replica for the desired HOSTID in a replica location request). If it is able to respond affirmatively, a response is sent back over the socket. If unable to respond usefully, the system sends the message on to one or more of its neighbors (depending on how the routing algorithm is configured) who repeat this process, and forwards the first affirmative response back to the message originator.

The main class of handled messages request the location of a replica for a particular SFS HOSTID. To aid in these lookups the FreeSFS node maintains a table of associations between HOSTIDs and hosts other than itself that maintain the HOSTIDs repository. Since this table is so compact, large amounts of supplemental routing information can accumulate in the node. Experienced nodes may offer their entire routing tables to freshly inserted nodes, rapidly informing new members of the network about resource distribution.

Communication between nodes is via TCP, in contrast to systems like Gnutella which use UDP style broadcast connections [3] . The traffic consists of request-response pairs over kept-alive sockets as in HTTP/1.1. After a request-response pair has complete, both nodes may elect to maintain the socket, reducing the network overhead incurred by re-opening sockets between hosts. Multiple requests may pend over a given socket before the first is resolved. The responses may return in arbitrary order; they can be associated with their originating request by comparing nonces if necessary. Nodes send requests and responses in ASCII text, much like HTTP requests, allowing for easy implementation of client software and extension of the messages that a system allows. Text mode communication avoids some of the difficulties associated with binary messaging (data types differ in size over different systems, binary formats clients require low level marshalling code, the behavior of the system may not be easily debuggable). Since the message sizes normally contain less than a 1K of data, text mode performance can be expected to be comparable to binary modes.

The routing of requests is modularized, making modified search algorithms and heuristics easy to retrofit into the reference implementation. The current system supports both depth-first (a client passes a given request to only one neighbor at a time) and limited-subset broadcast (a client broadcasts the message to some subset of its neighbor nodes), corresponding to beam search across the network. This allows fine tuning for scaling, since broadcast represents a higher performance, parallel search when the network connectivity does not represent the critical bottleneck in search performance. When the system scales, broadcast search rapidly fills the network with redundant requests (since a node must be contacted to determine if it has the data or has already received the request). Depth-first search minimizes network traffic, but touches nodes in the network more slowly. Gnutella's network, for example, experiences scaling problems due to its overly-anxious broadcast search algorithm. [2].

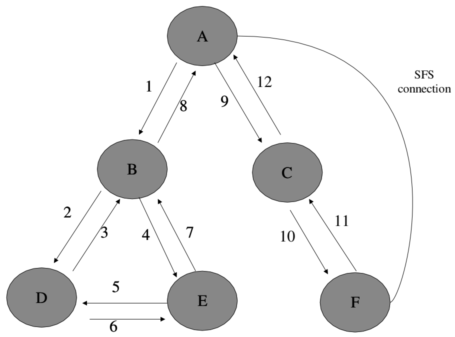

Figure 1: Depth first search

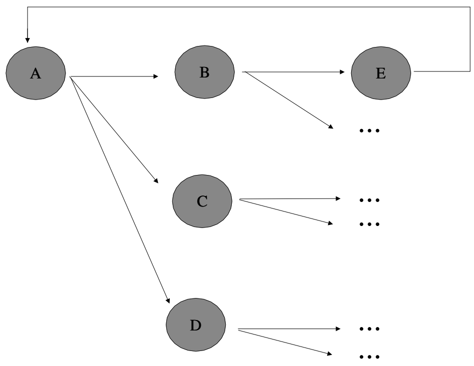

Figure 2: Beam search

4 Protocol

FreeSFS relies on a simple text-marshalled communication protocol to communicate between servers. Communication focuses on lookup of repositories by SFS HOSTID and on sharing routing information between hosts. Effort was made to implement minimal policy in the nodes of the FreeSFS network, allowing us to treat it as a baseline implementation of a distributed router and then consider extensions to the system separately.

The basic communication message format for FreeSFS is of the form: `(message type, nonce, args*)` where:

message type For requests, a type of request such as `get_file` or `get_table`. For responses, an equivalent type such as `found_file`, or `failure`.

nonce In the case of requests, this is an arbitrary random string to differentiate requests and prevent cycles. Responses use the nonce from the cooresponding request. All equivalent requests that originated from the same source contain the same nonce.

args* The semantics of these arguments depends on the message type. For example, `get_file` requests include the HOSTID of the mirrored content searched for.

Nodes can initiate and respond to three main message types in FreeSFS: `get_file`, `get_table`, and `announce_self`.

In the case of the `get_file` request the node receives, for example, `(get_file, 12345, HOSTID)` over a connection from a peer node. After parsing the request, the node will then check to determine if HOSTID exists in the table of SFSRODB repositories it itself serves. If the node does serve the HOSTID repository, it replies with `(heres_file, 12345, HOSTID, location)` where location is node's network address. If the node does not serve HOSTID, the node checks its routing table to determine if it knows of a machine that does serve HOSTID. If so it connects to that server and forwards the `get_file` request, tunneling the response back to the client if successful. Should the response indicate a failure, the host may remove the obsolete association from its routing table and continue as though no entry existed in the table.

In the case that neither of the options result in an affirmative result, the message is repeated to some subset of the other neighbors of the node. The node may attempt to use kept-alive network connections first, since all neighbor nodes can be considered to have an equal probabilty of satisifying the request. The neighbors will repeat the process above for the request. Should a neighbor contain a replica for HOSTID, it will return a response as above to the node. The node will forward this response to the client, and may elect to store the result in its own routing table.

The `get_table` request allows a node to request the routing table of other nodes. The target node responds with `(heres_table, nonce, <serialized table>)` if it sufficiently unloaded. This request allows for fast startup of a node, kick-starting it with the mature routing tables of more seasoned nodes. The client node may augment its own routing table with the additional associations or associate all HOSTIDs in the new table with the server that provided them. The reference implementation implements fast startup using `get_table` and performs a union of the results with its original table. The routing table is stored in a text file on disk, so it can also be easily augmented by hand.

The `announce_self` message informs a peer node of a new node's insertion into the network and reports the list of HOSTIDs available on the new node. The receiving node may optionally import the arguments into its routing table and forward the message to others of its peers.

Our trivial FreeSFS client uses this text protocol to query a daemon running on a specified host and port for a replica for a desired HOSTID. The request eventually returns either the network address of an active replica or failure after the search has traversed some subset of the network. The replica can then be mounted to appear as a local read-only filesystem. A more sophisticated client might intercept file system reads within the /sfs directory and transparently seek out and mount appropriate replicas using FreeSFS. The routing daemon and clients cannot compromise SFS's data authentication mechanisms since all file transfers take place exclusively with SFS.

5 Implementation

All code for FreeSFS was written in C++. The software depends on SFS for its asynchronous I/O infrastructure and uses STL extensively. We have compiled and tested it successfully on Linux and FreeBSD, although it occasionally fails mysteriously under heavy loads on FreeBSD for reasons we have been as yet unable to identify.

6 Performance

For a rudimentary test of the real-world scalability of our routing daemon we set up two nodes on separate machines, both with 550MHz Intel PIII processors, connected by a 10MBit LAN. Driver scripts written in Perl ran on a third machine with identical hardware. We tested four scenarios with four conditions each: successes and failures against one isolated node, and successes and failures against a node with one known peer, which was queried for each request-response pair. For each condition we ran ten simultaneous instantiations of the test driver, each issuing 100, 200, 300, or 400 requests in serial over separate TCP connections. The number of simultaneous connections we could manage was bounded by open file descriptor limitations. We saw a maximum of 5% cpu load on the daemon machines during these tests. Compare the resulting figures with our result of greater than 0.3 seconds for a single request for a file a few bytes in length from the local machine under FreeNet, excluding FreeNet's expensive filename hashing. These data do not validate the scalability of our network in any way, but at least demonstrate that our unoptimized reference implementation of the routing daemon can field hundreds of requests per second while consuming nominal cpu cycles. A more thorough analysis might leverage these results to draw conclusions about the scalability of the system as a function of average routing table size, bandwidth, etc.

Our second series of tests focused on the protocol's inherent performance and the value of the routing table updates. A chain of nodes was constructed, such that each node's routing table pointed only to the next node in the chain and contained no HostIDs for that node. Each node contained a unique repository HostID. Starting from the first node in the chain, the test script made a series of requests for a given HostID. The first request traversed down the chain of nodes to a predictable depth and returned. The cached response of the first request then served to satisify later requests. The test was repeated for various depths along the chain.

total # of requests	0 Hop Success	0 Hop Failure	1 Hop Success	1 Hop Failure
1000	258	384	95	369
2000	392	393	116	368
3000	389	395	106	372
4000	362	345	99	353

Table 1: For stress testing, a client script sent multiple requests to single-node and dual-node FreeSFS networks. O Hop Success and Failure denote sinlge-node network testing where the request is resolved locally in either a success or failure. 1 Hop Success and Failure denote the dual-node network testing where the request must be forwarded on to one other node to be satisfied.

7 Analysis

Determining some general search cost in a peer-to-peer system seems nearly impossible. Given some initial network topology, a probabilistic analysis of return times for a given object on the network can be accomplished through mathematical means. Since large numbers of independent variables including server performance, network connectivity, and routing table information characterize a network topology, results from such analysis are less than indicative of actual real-world peer-to-peer search performance.[7].

8 Future Work

A number of extensions could be made to FreeSFS with ease.

First, using the incremental database update feature in SFS Read-Only, a FreeNet-like anonymity could be implemented. Instead of returning the location a HOSTID is served from, the node could act as a tunnel for the two SFS Read-Only RPC calls. A given node then would be indistinguishable in all but performance from a node actually serving the repository and could act as a surrogate node. As files were requested,

Trial Number	Chain Length: 1	2	3	4	5	6	7	8	9	10
1	0.0006	0.0113	0.0239	0.0060	0.0132	0.0031	0.0123	0.0042	0.0026	0.0072
2	0.0007	0.0015	0.0093	0.0027	0.0043	0.0026	0.0026	0.0037	0.0102	0.0038
3	0.0009	0.0014	0.0015	0.0015	0.0012	0.0018	0.0013	0.0021	0.0023	0.0101
4	0.0007	0.0015	0.0014	0.0015	0.0014	0.0024	0.0015	0.0064	0.0023	0.0034
5	0.0010	0.0016	0.0015	0.0016	0.0012	0.0024	0.0022	0.0028	0.0054	0.0027
6	0.0007	0.0015	0.0021	0.0021	0.0011	0.0016	0.0031	0.0017	0.0025	0.0063
7	0.0008	0.0014	0.0014	0.0020	0.0020	0.0018	0.0012	0.0021	0.0015	0.0053
8	0.0007	0.0015	0.0103	0.0021	0.0012	0.0021	0.0010	0.0059	0.0043	0.0058
9	0.0006	0.0015	0.0017	0.0022	0.0014	0.0016	0.0012	0.0025	0.0026	0.0032
10	0.0011	0.0014	0.0014	0.0019	0.0011	0.0019	0.0028	0.0018	0.0017	0.0039

Table 2: In order to measure the cost of searching, a client script would send a request to a FreeSFS network that had the topology of a chain (where the first node only knows the second node and the second node only knows the third node, etc.). All requests were for an item residing in the last node of the chain, forcing full recursion. Each request was repeated for ten trials serially.

they could be tunneled through the surrogate node and cached there. Implementing a number of the finer points of FreeNet's algorithm, the anonymity of FreeNet could be combined with the authentication of SFS. The result would be user and publisher anonymity with data authentication. Hashing on servers further allows a more web-like interface to the FreeNet-esque file system.

Improvements in the routing algorithms of FreeSFS also seem obvious. Smarter routing table management, self-organizing specialization of nodes, and richer information in routing entries all might reduce lookup latency and bandwidth consumption. Since the routing table entries in FreeSFS occupy little memory and the overall space of names in FreeSFS is minimized because of per-server hashing, routing tables can be significant lists of nodes on the network. Mirrors also could be selected from a route table based on round-trip time to guarentee the return of the nearest replica.

Finally, FreeSFS could be integrated into SFS's lookup to provide transparent access to directories named with only HOSTIDs. This removes the need for locations for read-only repositories, creating a useful abstraction barrier between clients and the services they access.

All of these project directions were discussed by the authors of the paper, and each seems conceptually reasonable. Due to time constraints, only a basic level of daemon functionality was implemented. These further directions for work validate some of the possibilities of SFS's self-certifying pathnames and their combination with decentralized lookup schemes.

9 Conclusions

The data integrity assurances and manageability of SFS Read-Only suggest it as a natural partial solution to the mirror problem; adding FreeSFS's decentralized replica search completes the package. FreeSFS enables efficient lookup of volunteer repository replicas and fast insertion of new mirrors in response to demand, which should encourage increased mirror utilization due to the convenience of mirror discovery, and in turn lead to lower loads on master servers and higher resource availability.

We believe FreeSFS offers a unique and useful combination of features in a peer-to-peer system while performing competitively with existing systems. With respect to Gnutella, we introduce the significant improvement of a larger unit of granularity in the namespace: the repository. This provides two major advantages. It should substantially reduce the size of the namespace since each name corresponds to a much larger body of data and many names share an explicit mapping to a single master repository, reducing the expected number of hops to locate a particular resource. Secondly, it provides an efficient format for accumulating descriptive network information at nodes to reduce the depth of future searches. Such information would be impractical to maintain in Gnutella's flat namespace. We expect these two advantages would reduce both the number of searches required by a client in our network and the expected number of nodes each search would visit. Finally, the value of the additional data integrity properties of FreeSFS is obvious.

With respect to FreeNet, we sacrifice anonymity features for data integrity and superior performance. FreeNet consumes nonfunctional bandwidth to obscure network traffic, an unnecessary feature for many potential users. It also denies administrative control over nodes to their owners, as nodes autonomously converge on specialization in ranges of the namespace which may include objectionable content. We believe that for applications where anonymity is unnecessary FreeSFS is strictly superior to FreeNet.

Our testing demonstrated the plausibility of our FreeSFS implementation on a reasonably large scale along with providing foundational data necessary for theoretical analysis of network performance.

Acknowledgments

Thanks to Kevin Fu for his help in compiling and running the SFSRoDB package[8]. Thanks also to Professors Franz Kaashoek and Robert Morris for consulting about the project idea in its early inception.

References

[1] 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.

[2] A. Pava, "Gnutella is dead," http://music.zdnet.com/features/highnote/092100_gnutella_dead.html, Sept. 2000.

[3] Gnutella, "http://gnutella.wega.com," 2000.

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

[5] Roger Dingledine, "The free haven project: Design and deploymenet of an anonymous secure data haven," M.S. thesis, Massachusetts Institute of Technology, June 2000.

[6] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos, "A prototype implementation of archival intermemory," in *Proceedings of the fourth ACM Conference on Digital libraries (DL '99)*, 1999.

[7] I. Clarke, O. Sandberg, B. Wiley, and T. Hong, "Freenet: A distributed anonymouse information storage and retrieval system," in *ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[8] Kevin Fu, "emails about compiling sfs read-only," emails, December 2000.