Swarm: A Log-Structured Storage System for Linux

Ian Murdock
Department of Computer Science
The University of Arizona
imurdock@cs.arizona.edu

John H. Hartman
Department of Computer Science
The University of Arizona
jhh@cs.arizona.edu

Abstract

Swarm [3] is a storage system for Linux that provides scalable, reliable, and cost-effective data storage. At its lowest level, Swarm implements a log-structured interface to a cluster of storage devices. Above the log, Swarm provides an infrastructure that allows high-level abstractions and functionality to be implemented easily and efficiently. This paper describes the design and implementation of Swarm, paying particular attention to the Swarm infrastructure and how it has been used to construct two storage systems: Sting, a log-structured file system for Linux, and ext2fs/Swarm, a Swarm-based version of the Linux ext2 file system that runs unmodified above a block device compatibility layer. The paper concludes with a discussion of our experiences using Linux as a platform for research.

1 Introduction

In Linux, file systems interface with applications through an abstraction layer called the Virtual Filesystem Switch (VFS). The VFS separates file system interface from implementation, allowing many different file systems to coexist in a single file system namespace. The VFS implements functionality common to all file systems, including the system calls that access or modify file system data and metadata (e.g., `read` and `mkdir`). To perform file-system-specific operations, the VFS vectors control to lower-level file system implementations when appropriate (e.g., when a block needs to be read from disk, or when an entry needs to be added to a directory). Below the VFS, file systems access storage indirectly through a caching subsystem. For example, disk-based file systems communicate with the underlying disks through a buffer cache, which provides a level of indirection between disks and the file systems stored on them.

In general, high-level abstractions such as those provided by file systems are tightly coupled with the low-level storage devices on which they are implemented, making it difficult to extend or configure the storage system. For example, since the ext2 file system interfaces with disks through the buffer cache, it is not possible to run ext2 above a storage device that is not block-oriented. Furthermore, the high-level abstractions themselves are often tightly coupled, providing a single large feature set that is difficult to change without directly modifying the file system. For example, ext2 implements the standard UNIX file system functionality and interface. In general, to extend the functionality of ext2 (e.g., to add support for transparent compression of file data), it is necessary to modify ext2 directly to support the desired features.

To address this inherent inflexibility, several projects have created extensible storage systems. One group of projects focuses on stacking vnodes [5, 11]. In many versions of UNIX, a vnode represents an object in the file system namespace, such as a file, directory, or named pipe. In the context of Linux, which does not support vnodes, the same effect may be achieved by layering at the VFS level, allowing new functionality to be interposed between layers. For example, to add compression to a file system, a file compression layer is interposed between the file system and the buffer cache that compresses files as they are written, uncompresses files as they are read, and manages all the details required to make this transparent (e.g., making sure cache blocks are managed properly). Thus, VFS layering allows certain functionality to be added to file systems without having to modify the file systems themselves.

Other projects attempt to provide flexibility at the bottom end. For example, Linux supports transparently-compressed block devices, providing support for compression of file system data and metadata at the block level, and software RAID, which combines several physical block devices into one virtual block device that looks and acts like a RAID. This kind of support allows high-level abstractions to escape the confines of a particular storage abstraction to a limited extent without requiring modification of the higher levels.

The problem with these approaches is that they only allow changes in implementation; they do not allow changes in interface. At the top end, VFS layering does not allow extensions to alter the file-oriented interface the VFS provides; this limits the expressibility of VFS layers to functionality that matches the file abstraction and interface. For example, file system filters, like transparent compression and encryption, fit very nicely into the VFS framework, but other kinds of extensions that diverge from the file abstraction and interface are more difficult to implement, such as integrated support for database-like functionality (e.g., transactions). At the bottom end, extensions like transparent compression of block devices and software RAID allow some file systems to provide extended functionality, but only those that support the particular storage abstraction being extended (in this case, those file systems that run above block devices).

Swarm [3] is a storage system that attempts to address the problem of inflexibility in storage systems by providing a configurable and extensible infrastructure that may be used to build high-level storage abstractions and functionality. At its lowest level, Swarm provides a log-structured interface to a cluster of storage devices that act as repositories for fixed-sized pieces of the log called *fragments*. Because the storage devices have relatively simple functionality, they are easily implemented using inexpensive commodity hardware or network-attached disks [2]. Each storage device is optimized for cost-performance and aggregated to provide the desired absolute performance.

Swarm clients use a *striped log* abstraction [4] to store data on the storage devices. This abstraction simplifies storage allocation, improves file access performance, balances server loads, provides fault-tolerance through computed redundancy, and simplifies crash recovery. Each Swarm client creates its own log, appending new data to the log and forming the log into fragments that are striped across the storage devices. The parity of log fragments is computed and stored along with them, allowing missing portions of the log to be reconstructed when a storage device fails. Since each client maintains its own log and parity, the clients may act independently, resulting in improved scalability, reliability, and performance over centralized file servers.

Swarm is a storage system, not a file system, because it can be configured to support a variety of storage abstractions and access protocols. For example, a Swarm cluster could simultaneously support Sun's Network File System (NFS) [10], HTTP, a parallel file system, and a specialized database interface. Swarm accomplishes this by decoupling high-level abstractions and functionality from low-level storage. Rather than providing these abstractions directly, Swarm provides an infrastructure that allows high-level functionality to be implemented above the underlying log abstraction easily and efficiently. This infrastructure is based on layered modules that can be combined together to implement the desired functionality. Each layer can augment, extend, or hide the functionality of the layers below it. For example, an atomicity service can layer above the log, providing atomicity across multiple log operations. In turn, a logical disk service can layer above this extended log abstraction, providing a disk-like interface to the log and hiding its append-only nature. This is in contrast to VFS or vnode layering, in which there is a uniform interface across all layers.

Swarm has been under development at the University of Arizona for the past two years. We have implemented the Swarm infrastructure in both a user-level library and the Linux kernel (versions 2.0 and 2.2), and we have used this infrastructure to implement two storage systems, Sting, a log-structured file system for Linux, and ext2fs/Swarm, a Swarm-based version of the Linux ext2 file system that runs unmodified above a block device compatibility layer.

2 Swarm infrastructure

Swarm provides an infrastructure for building storage services, allowing applications to tailor the storage system to their exact needs. Although this means that many different storage abstractions and communication protocols are possible, Swarm-based storage systems typically store data in a *striped log* storage abstraction and use a storage-optimized protocol for transferring data between client and server.

In the striped log abstraction, each client forms data into an append-only log, much like a log-structured file system [9]. The log is then divided into fixed-sized pieces called *fragments* that are striped across the storage devices. Each fragment is identified by a 64-bit integer called a *fragment identifier (FID)*. Fragments may be given arbitrary FIDs, allowing higher levels to construct multi-device fragment address spaces. As the log is written, the parity of the fragments is computed and stored, allowing missing fragments to be reconstructed should a storage device fail. A collection of fragments and its associated parity fragment is called a *stripe*, and the collection of devices they span is called a *stripe group*.

The striped log abstraction is central to Swarm's high-performance and relatively simple implementation. The log batches together many small writes by applications into large, fragment-sized writes to the storage devices, and stripes across the devices to improve concurrency. Computing parity across log fragments, rather than file blocks, decouples the parity overhead from file sizes, and eliminates the need for updating parity when files are modified, since log fragments are immutable. Finally, since each client writes its own log, clients can store and access data on the servers without coordination between the clients or the servers.

Swarm is implemented as a collection of modules that may be layered to build storage systems in much the same way that protocols may be layered to build network communications subsystems [6]. Each module in Swarm implements a storage service that communicates with the lower levels of the storage system through a well-defined interface, and exports its own well-defined interface to higher levels. Storage systems are constructed by layering the appropriate modules such that all interfaces between modules are compatible. This section describes the basic modules that are used to construct storage systems in Swarm.

2.1 Disk

As in most storage systems, the disk is the primary storage device in Swarm. Swarm accesses disks through the *disk layer*. The disk layer exports a simple, fragment-oriented interface that allows the layers above to read, write, and delete fragments. Fragment writes are atomic; if the system crashes before the write completes, the disk state is "rolled back" to the state it was in prior to the write.

The disk layer operates by dividing the disk into fragment-sized pieces and translating fragment requests into disk requests. The mappings from fragment identifiers to disk addresses are stored in an on-disk *fragment map* that is stored in the middle of the disk to reduce access time. The disk contains two copies of the fragment map, each with a trailing timestamp, to permit recovery when the system crashes while a fragment map write is in progress. As an optimization, the disk layer only writes out the fragment map periodically, saving two additional seeks and a disk write on most fragment operations.

The fragment map is not written to disk each time it is updated, so the disk layer must be able to make it consistent again after crashes, or fragments written after the last fragment map write will be lost. To address this problem, the disk layer borrows a trick from the Zebra storage server [4]. It includes a header with each fragment that contains enough information for the fragment map to be updated appropriately at recovery time. To allow crash recovery to proceed without having to scan the entire disk, the disk layer preallocates the next set of fragments to write before it writes the fragment map and stores their locations in the fragment map. At recovery time, the disk layer need only examine the fragments preallocated in the fragment map.

2.2 Network-attached storage

Swarm also supports the use of network-attached storage devices. Swarm provides a network transparency layer that has the same interface as the disk layer, allowing locally-attached and network-attached storage devices to be used interchangeably. The network transparency layer accepts fragment operations from the layers above it, sends them across the network to the appropriate device, and returns the result of the operation to the caller transparently.

In Swarm, network-attached storage devices are called *storage servers*. The storage servers are essentially enhanced disk appliances running on commodity hardware, and provide the same fragment-oriented interface as disks, with added support for security. The storage server security mechanism consists of access control lists that may be applied to arbitrary byte ranges within a fragment. The lists give a client fine-grained control over which clients can access its data, without limiting the ways in which fragments may be shared.

2.3 Striper

The *striper* layer is responsible for striping fragments across the storage devices. It exports a fragment-oriented interface to the higher layers. As fragments are written, the striper forms the fragments into stripes and writes them to the appropriate storage devices in parallel. To provide flow control between the striper and the storage devices, the striper maintains a queue of fragments to be written for each device; the striper puts fragments into these queues, and the storage layers take them out and store them on the appropriate device. Fragments are written to disk one-at-a-time, but with a storage server, the network layer transfers the next fragment to the server while the previous one is being written to disk; this keeps both the storage server's disk and the network busy, so that as soon as one fragment has been written to disk, the server can immediately begin writing another fragment.

2.4 Parity

The *parity layer* implements the standard parity mechanism used by RAID [8]. The parity layer sits above the striper and provides a compatible interface, allowing the striper and parity layer to be used interchangeably. One difficulty that arises from Swarm's support for stripe groups is determining which fragments constitute the remaining fragments of a stripe during reconstruction, and on which devices they are stored. Swarm solves this problem by storing stripe group information in each fragment of a stripe, and numbering the fragments in the same stripe consecutively. If fragment N needs to be reconstructed, then either fragment N-1 or fragment N+1 is in the same stripe. The client queries all the storage devices until it finds either fragment N-1 or N+1. For locally-attached disks, the client uses configuration information to find all the disks; for storage servers, the client simply broadcasts to find the desired fragments. Broadcast is used because it is simple and makes Swarm self-hosting—no additional mechanism is needed to distribute stripe group and storage server information reliably to all clients.

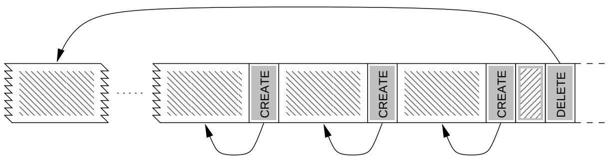

Figure 1: The light objects are blocks, and the dark objects are records. Each CREATE record indicates the creation of a block, and each DELETE record indicates a deletion; the arrows show which block is affected by each record and represent references visible to the log layer. Note that the contents of the blocks themselves are uninterpreted by the log layer.

2.5 Striped log

Above the fragment-oriented interfaces provided by the storage, striper, and parity layers is the *log layer*. The log layer implements the striped log storage abstraction and corresponding interface, forming data written by higher levels into an append-only log and striping the log across the underlying storage devices. The layers above the log are called *storage services* (*services* for short) and are responsible for implementing high-level storage abstractions and functionality. The log layer's main function is to multiplex the underlying storage devices among multiple services, allowing storage system resources to be shared easily and efficiently.

The log is a conceptually infinite, ordered stream of *blocks* and *records* (Figure 1). It is append-only: blocks and records are written to the end of the log and are immutable. Block contents are service-defined and are not interpreted by the log layer. For example, a file system would use blocks not only to store file data, but also inodes, directories, and other file system metadata. Once written, blocks persist until explicitly deleted, though their physical locations in the log may change as a result of cleaning or other reorganization.

New blocks are always appended to the end of the log, allowing the log layer to batch together small writes into fragments that may be efficiently written to the storage devices. As fragments are filled, the log layer passes them down to the striper for storage. Once written, a block may be accessed given its *log address*, which consists of the FID of the fragment in which it is stored and its offset within the containing fragment. Given a block's log address and length, the log layer retrieves the block from the appropriate storage device and returns it to the calling service. When a service stores a block in the log, the log layer responds with its log address so that the service may update its metadata appropriately.

Records are used to recover from client crashes. A crash causes the log to end abruptly, potentially leaving a service's data structures in the log inconsistent. The service repairs these inconsistencies by storing state information in records during normal operation, and re-applying the effect of the records after a crash. For example, a file system might append records to the log as it performs high-level operations that involve changing several data structures (e.g., as happens during file creation and deletion). During replay, these records allow the file system to easily redo (or undo) the high-level operations. Records are implicitly deleted by *checkpoints*, special records that denote consistent states. The log layer guarantees atomicity of record writes and preserves the order of records in the log, so that services are guaranteed to replay them in the correct order.

2.6 Stripe cleaner

As in LFS, Swarm uses a *cleaner* to periodically compress free space in the log to make room for new stripes [9]. In Swarm, the cleaner is implemented as a layer above the log, hiding the log's finite capacity from higher-level services. The cleaner service monitors the blocks and records written to the log, allowing it to track which portions of the log are unused. A block is cleaned by re-appending it to the log, which changes its address and requires the service that wrote it to update its metadata accordingly. When a block is cleaned, the cleaner notifies the service that created it that the block has moved. The notification contains the old and new addresses of the block, as well as the block's creation record.

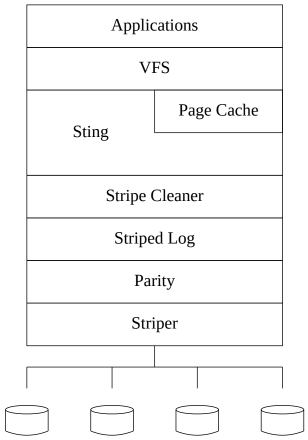

Figure 2: This figure shows the Swarm module configuration for Sting. The storage devices at the bottom layer may be either locally-attached or network-attached disks, and the number of devices is configurable.

The creation record contains service-specific information that makes it easier for the service to update its metadata. For example, the creation record for a file block might contain the inode number of the block's file and the block's offset. The cleaner is also responsible for free space management, enforcing quotas on higher-level services, reserving the appropriate number of stripes so that cleaning is always able to proceed, and initiating cleaning to make room for new stripes in the event there are no free stripes.

3 Swarm storage systems

We have implemented the Swarm infrastructure described in the previous section in both a user-level library and the Linux kernel, and we have used this infrastructure to implement two storage systems. This section describes our two demonstration storage systems, Sting, a log-structured file system for Linux, and ext2fs/Swarm, a Swarm-based version of the Linux ext2 file system that runs unmodified above a block device compatibility layer.

3.1 Sting

Sting is a log-structured file system for Linux that is based on Swarm. It is similar to Sprite LFS [9], although it is smaller and simpler because the underlying Swarm infrastructure deals transparently with log management and storage, cleaning, and other LFS tasks. As with all other Linux-based file systems, Sting interacts with applications through the Linux Virtual Filesystem Switch (VFS). Thus, Linux applications may run above Sting as they would any other Linux file system. However, unlike other Linux-based file systems, Sting accesses storage through the striped log abstraction rather than a block device (see Figure 2).

Sting stores all of its data and metadata in blocks, and uses many of the same data structures as LFS. Files are indexed using the standard UNIX inode mechanism. Directories are simply files mapping names to inode numbers. Inodes are located via the *inode map*, or *imap*, which is a special file that stores the current location of each inode. A similar data structure is not needed in most file systems because inodes are stored at fixed locations on the disk and modified in place. In Sting, when an inode is modified or cleaned, a new version of the inode is written to the end of the log, causing the inode's address to change periodically.

Sting uses records to recover its state after a crash. For example, when a file is created, Sting writes a record to this effect to the log, so that it may easily recreate the file after a crash. This mechanism is similar to that used by journaling file systems. Without Swarm's record mechanism, Sting would be forced to write out the affected metadata when a file is created, and write it out in a particular order so that a file system integrity checker (e.g., `fsck`) can rectify an inconsistency after a crash. In Swarm, services may summarize changes in records without actually making them, and they are freed from having to do their own ordering because Swarm guarantees that records are properly ordered by time of creation during recovery.

Sting runs above Swarm's striped log, not a block device, so it is unable to use the buffer cache used by disk-based file systems for buffering writes and caching metadata. Rather than adding our own specialized buffering mechanism to Sting, we modified the page cache to support tracking and writing back dirty pages so that Sting could use it as its primary file cache for both reads and writes. With our changes, pages may be modified and marked "dirty" for later writing. When a page is marked dirty, it is added to a list of dirty pages. Then, during each run of `update`, the list of dirty pages is traversed. If a page is older than a certain threshold age (currently twenty-five seconds), it is written via a VFS method called `writepage`. In addition, Sting uses the page cache to cache indirect blocks, directory blocks, symbolic links, and the inode map. Our modifications to the page cache were small but allowed us to more easily implement a file system that does not run above a block device.[1]

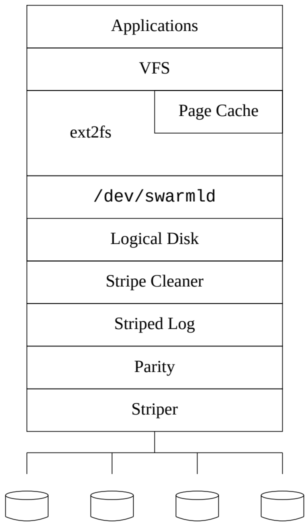

Figure 3: This figure shows the Swarm module configuration for ext2fs/Swarm. The stack below the file system is identical to that of Figure 2, with the introduction of two compatibility layers. As with Sting, the storage devices at the bottom layer may be either locally-attached or network-attached disks, and the number of devices is configurable.

Sting accesses storage through Swarm, allowing it run on either locally-attached disks or on a Swarm cluster as a network file system (or a combination of the two). In addition, it may also take full advantage of Swarm's striping capabilities transparently. It does not yet support file sharing between clients. We are in the process of implementing a locking service that will allow clients to synchronize file accesses, allowing us to easily modify Sting to be used as a distributed file system.

3.2 Ext2fs/Swarm

Ext2fs/Swarm is a version of the Linux ext2 file system that runs unmodified above Swarm. This is possible through the use of a special logical disk service that provides a disk-like interface above Swarm's striped log, and a Linux block device driver that translates Linux block device requests into Swarm logical disk requests (see Figure 3). The block device driver exports the usual block device interface via the `/dev/swarmld` device file, which may be read, written, or mounted like any other block device. Thus, via the logical disk service and `/dev/swarmld`, Swarm appears to Linux to be just an ordinary block device.

The Swarm logical disk is similar to the MIT Logical Disk [1], but it is much simpler because it is implemented as a service above Swarm. It hides the append-only nature of the striped log, providing the illusion that higher-level services are running above a disk. As with real disks, blocks in a logical disk are written to fixed addresses that do not change over time, and that can be overwritten in place. This provides a much more convenient abstraction for some services than the append-only log, in which services must always append blocks to the end of the log and block addresses change as the result of being cleaned or overwritten.

The Swarm logical disk service's primary function is to maintain a mapping from a logical disk's finite address space to the log's infinite address space. This mapping is maintained in main memory, and is periodically written to the log by the logical disk service. In addition to updating the mapping table during normal operation, the logical disk service intercepts change-of-address notifications from the stripe cleaner as blocks are cleaned and updates the mapping table transparently.

Although the logical disk is a general-purpose storage service that may be used to implement any storage system, it serves as an excellent compatibility layer that allows existing file systems that run above a disk to run unmodified above Swarm. Using `/dev/swarmld`, ext2fs (and all other disk-based Linux file systems) may run above Swarm unmodified. As read requests and write requests are generated by applications, they are passed to the device driver; in turn, the requests are passed to the logical disk, which performs the appropriate operation on the striped log. In addition to striping, improved performance on small writes, and other benefits provided by Swarm, ext2fs/Swarm can be configured to run on a Swarm cluster as a network file system. Note, however, that because we run ext2fs unmodified, its concurrency aspects are unchanged, so only one client at a time may have write access to any given instance of it.

[1] Many of these shortcomings have been addressed in the latest development version of Linux, 2.3. Our development efforts were under Linux 2.0 and Linux 2.2.

4 Experiences using Linux for research

Swarm has been under development in the Department of Computer Science at the University of Arizona for the past two years. Early on, we decided to base Swarm on Linux because Linux had a large and rapidly-growing user base, and we wanted to build a storage system that people could use for day-to-day data storage. As with most decisions, ours was not without a downside, and we have learned some important lessons about Linux over the past few years that may prove useful to fellow researchers who are considering using Linux as a platform for research. We hope these observations will be equally useful to Linux developers, and that they will help make Linux an even better platform for research and development.

The lack of documentation is one of the biggest limitations of Linux to the uninitiated. There is very little documentation on the internal workings of Linux, and what little exists tends to become outdated quickly. The code is often the best (and sometimes the only) point of reference on how a particular part of Linux works. Unfortunately, the functionality of a piece of code is not always obvious at first glance, as comments are sparse in many places, and the code is often highly optimized and thus frequently difficult to understand at first glance.

Fortunately, Linux has a large and friendly development community that is normally more than happy to answer questions (as long as the asker has done his homework first), and the Internet serves as an invaluable archival reference for finding out if someone else has asked the same questions in the past (they almost certainly have). Still, Linux would do well to improve its documentation efforts. Since the code evolves so rapidly, any successful documentation mechanism must be somehow tied to the code itself (e.g., automatically generated from comments in the code), to prevent divergence of documentation from documented functionality.

Another limitation of Linux is its rather spartan development environment. Linux (at least the x86 version) does not include a kernel debugger, which we consider to be an essential part of any operating system development environment. Rather than using an interactive debugger, Linux developers prefer to rely on `printk` to debug new code, and use textual crash ("oops") dumps of registers, assembly code, and other low-level system state when something goes wrong. Fortunately, there are patches available that allow the kernel to be debugged interactively. Linux is also light on diagnostic facilities. We often found ourselves having to manually poke around inside system data structures using the debugger or `printk` to gain insight into the source of a bug when something had gone wrong. A clean interface to displaying and analyzing system state, accessible from the debugger, is another integral part of any operating system development environment.

Finally, Linux does not always follow good software engineering practices [7]. For example, much of the code we worked with seemed optimized unnecessarily, usually at the expense of code clarity. Operating system code should only be optimized when doing so has a clear, *quantified* impact on *overall* performance. Most operating system code is not performance critical and has little or no effect on on overall performance; such code should be structured for the human reader rather than the computer, to make the code easier to understand and maintain, to make bugs less likely to be introduced when the code is modified, and to make bugs that are introduced easier to find and fix. Furthermore, the abstraction boundaries in Linux are not always obeyed or even well-defined. We had problems with both the file system and I/O subsystems in this area, where the implementation of an abstraction made certain assumptions about how it would be used that were not always readily apparent from the abstraction's interface.

Despite its shortcomings, Linux has treated us well. Ten years ago, we would have had to license the source code to a proprietary operating system or use an in-house research operating system to implement Swarm, either of which would have limited the impact of our work. Linux has allowed to us to implement Swarm in a real operating system that is used by real people to do real work. With a little more work in the areas of documentation, development environment, and software engineering practices, Linux has the potential to be an excellent platform for systems research and development.

5 Acknowledgements

We would like to thank Tammo Spalink and Scott Baker for their help in designing and implementing Swarm. We would also like to thank our shepherd, Stephen Tweedie, for his helpful comments and suggestions. This work was supported in part by DARPA contracts DABT63-95-C-0075 and N66001-96-8518, and NSF grants CCR-9624845 and CDA-9500991.

6 Availability

For more information about Swarm, please visit `http://www.cs.arizona.edu/swarm/`.

References

[1] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 14th Symposium on Operating Systems Principles*, December 1993.

[2] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 1997.

[3] John H. Hartman, Ian Murdock, and Tammo Spalink. The Swarm scalable storage system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, June 1999.

[4] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. *ACM*

Transactions on Computer Systems, 13(3):274–310, August 1995.

[5] John S. Heidemann and Gerald J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[6] Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[7] Bulter W. Lampson. Hints for computer system design. *Operating Systems Review*, 17(5):33–48, October 1983.

[8] David Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Record*, 17(3):109–116, September 1988.

[9] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[10] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *Proceedings of the Summer 1985 USENIX Conference*, June 1985.

[11] Erez Zadok, Ion Badulescu, and Alex Shender. Extending file systems using stackable templates. In *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999.