The Swarm Scalable Storage System

John H. Hartman, Ian Murdock, and Tammo Spalink
Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

Abstract

Swarm is a storage system that provides scalable, reliable, and cost-effective data storage. Swarm is based on storage servers, rather than file servers; the storage servers are optimized for cost-performance and aggregated to provide high-performance data access. Swarm uses a striped log *abstraction to store data on the storage servers. This abstraction simplifies storage allocation, improves file access performance, balances server loads, provides fault-tolerance through computed redundancy, and simplifies crash recovery. We have developed a Swarm prototype using a cluster of Linux-based personal computers as the storage servers and clients; the clients access the servers via the Swarm-based Sting file system. Our performance measurements show that a single Swarm client can write to two storage servers at 3.0 MB/s., while four clients can write to eight servers at 16.0 MB/s.*

1. Introduction

Decentralized storage systems have several advantages over centralized file servers. Individual storage devices communicate with each other and their clients via a network, and are aggregated to provide high-performance, scalable, flexible, and fault-tolerant storage service. Control is distributed across the servers and clients, molding the components into a single, seamless system. In contrast, a centralized file server is often a performance bottleneck, doesn't scale well, and is a single point of failure.

Decentralization often breeds complexity, however, as distributed algorithms are typically more complex than their centralized counterparts. Nonetheless, the advantages of such a decentralized storage service make it worth the effort to tackle the inherent complexities of distribution. Swarm is a network storage system designed to run on a cluster of simple network-attached storage devices. Swarm uses log-based striping [7] to simplify the distributed control of the system; data are formed into logs that are striped across the servers in a redundant fashion. This allows the system to scale with the number of servers—as more servers are added, performance and capacity are both increased, avoiding the file server bottleneck. RAID-style redundancy provides high availability despite server failures. Parity is computed and stored for groups of data blocks that span multiple servers. Should one server fail, any block it stores can be reconstructed by computing the parity of the remaining blocks in the same parity group. Computing parity for the logs, instead of individual data blocks, allows clients to manage their logs and parity individually, avoiding synchronization overhead.

Swarm does not enforce a particular storage abstraction or access protocol; rather, it provides a configurable, extensible infrastructure for building storage services that stripe across a cluster of servers, allowing the services to implement their own abstractions and access protocols. Swarm pushes much of the services' functionality to the clients, as this allows the system's performance to scale with the number of clients. High-level abstractions and functionality, such as that provided by a distributed file system, are implemented on the clients, making Swarm flexible, while avoiding the bottlenecks imposed by running such software on the storage servers or a centralized file server.

To demonstrate Swarm's usefulness, we built a prototype on a collection of personal computers, connected by a switched network. The Swarm storage servers are implemented as user-level processes on the Linux operating system. We developed a Swarm-based local file system called Sting that allows individual clients to access files as they would on a local disk, except that the file data are instead stored on Swarm, providing Swarm's benefits of scalable performance and reliable operation. Our performance measurements show that a single client can write to two storage servers at 3.0 MB/s, increasing to 5.5 MB/s with four servers. The bottleneck in this configuration is the client, and performance improves as servers are added because the parity overhead is amortized over more data fragments. Four clients can write to eight servers at 16 MB/s, demonstrating Swarm's scalability. Sting running the Modified

Andrew Benchmark [11] on a single server is nearly twice as fast as the Linux ext2fs local file system on a local disk, using the same hardware.

2. Swarm overview

Synchronization and coordination are the bane of distributed systems. A system that is more than simply a collection of independent computers requires cooperation between its components. Cooperation is a double-edged sword, however; increased cooperation may increase the usefulness of the system, but it will also increase the amount of synchronization between the computers and limit the system's overall scalability. After all, a collection of independent computers can scale to infinite size. One of Swarm's design goals is to minimize the synchronization between components, so that a Swarm client only incurs synchronization overhead for the services it uses. Swarm servers do not synchronize with each other, and a client can access the servers without synchronizing with the other clients. If a single client chooses to access a single server, it is not forced to coordinate with other clients and servers in the system.

The design goal of minimal coordination makes Swarm notable for what it is not—it is not a distributed file system, nor is it a distributed virtual disk. Both of these storage abstractions require extensive cooperation and synchronization between the clients and the servers, to map files and blocks to servers, allocate and deallocate space, and maintain consistency. These high-level abstractions are useful to those applications and users that need them, but impose an undue tax on those that don't, and limit the overall scalability of the system.

The basic storage abstraction in Swarm is that of a *striped log*. Each client creates its own log, formed from the data it writes, in much the same fashion that a log-structured file system stores its data in a log. The log not only batches small writes together, improving their performance, but also simplifies the parity mechanism used to recover the log upon a server failure. Each client computes the parity for its own log, and writes the parity as the log is written. This parity can then be used to reconstruct lost portions of the log.

The use of a separate log for each client allows the clients to act independently; each can store data in its log, compute parity, and stripe the log across the servers without coordination with other clients. Clients can also independently reconstruct log data lost in a server failure. In addition, since the clients control the log creation and storage, no coordination is required among the servers.

High-level abstractions, such as a distributed file system or a distributed virtual disk, can be constructed on top of the striped logs. Swarm clients cooperate to implement these abstractions. This necessarily requires synchronization between the clients, but only between those that wish to access the shared abstraction. It is our belief that such distributed storage services are best implemented at a high level, as high-level synchronization is needed to access storage anyway. Two applications running on different clients must synchronize their accesses to shared data to ensure consistency, even if the storage system enforces consistency. For example, the reader of a file must synchronize with the writer, even if the underlying file system provides strong consistency on file contents. This reduces the importance of low-level synchronization primitives and shared storage abstractions.

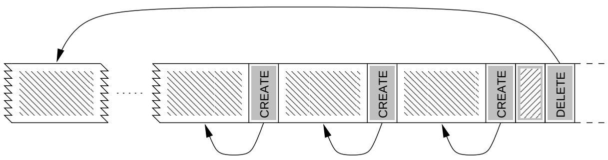

Figure 1. Log Format. The light objects are blocks, and the dark objects are records. Each CREATE record indicates the creation of a block, and each DELETE record indicates a deletion; the arrows show which block is affected by each record and represent references visible to the log layer. Note that the contents of the blocks themselves are uninterpreted by the log layer, as are the contents of those records not created by the log layer itself.

2.1. Log layer

2.1.1. Log format

The log is a conceptually infinite, ordered stream of *blocks* and *records*. It is append-only: blocks and records are written to the end of the log and cannot be modified subsequently. The log layer does not interpret block contents, allowing services to store anything they wish inside a block. For example, a file system might 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.

Records are used by storage services to recover from client crashes. A crash causes the log to end abruptly, potentially leaving the service's data structures in the log inconsistent. A service repairs these inconsistencies by storing state information in records during normal operation, and re-applying the effect of the records after a crash. This is a standard database recovery technique, and its use in Swarm allows services to reconstruct the operations they were performing at the time of the crash and update their data structures accordingly. To enable replay, the log layer guarantees the atomicity of writing individual records; it also preserves the order of records in the log, so the order in which they are replayed reflects the sequence of operations and states they represent.

To assist in crash recovery, the log layer automatically creates records that track log operations such as block creation and deletion. Services may also log service-specific operations in their own records; as with blocks, the log layer does not interpret the contents of records it does not create, so a service may create many types of records, and store whatever it wishes inside them. 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.

New data are always appended to the end of the log, allowing the log layer to batch together small writes and store the log efficiently in fixed-sized pieces called *fragments*. Each fragment is stored on a single storage server, and is identified by a 64-bit integer called a *fragment identifier* (*FID*). Blocks within a fragment are addressed by an FID and an offset within the fragment. Given the address of a block and its length, the log layer contacts the appropriate storage server, retrieves the data requested, and returns the data to the calling service. When a service stores a block in the log, the log layer responds with the FID and offset of the block so that the service may update its metadata appropriately.

2.1.2. Striping

As fragments are filled, the log layer stripes them across the storage servers. A *stripe* is a set of two or more fragments, one of which contains *parity* of the other fragments in the set. Each fragment in a stripe is stored on a different server, and the collection of servers that stores a stripe is called its *stripe group*. The parity fragment allows a client to reconstruct the contents of any fragment in the stripe given the contents of the rest of the fragments; thus, all fragments of a stripe are accessible even if one of the servers in the stripe group is unavailable. The parity fragment of successive stripes is rotated across the servers, balancing server loads during reconstruction.

There are several advantages to using stripe groups containing a subset of the storage servers, rather than all storage servers. First, clients can stripe across disjoint stripe groups, minimizing contention for servers and increasing scalability. Second, in the event of a server failure, fragment reconstruction involves fewer servers, lessening its impact on performance. Third, and perhaps most important, Swarm can tolerate multiple server failures, as long as two failures do not occur in the same stripe group. If all stripes were striped across all servers, multiple server failures would result in lost data.

The log layer software in the client is multi-threaded, and performs several operations concurrently to improve performance. First, fragments are written to the servers asynchronously, so that several may be written simultaneously. Second, a stripe's parity is computed as its fragments are written. Third, the log layer transfers a fragment to a server while the previous fragment is being written to disk; this keeps both the 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. This is a rudimentary form of flow control, and we are planning to investigate more sophisticated solutions.

2.1.3. Recovery

A service recovers from a crash by replaying its records from the log in a process known as *log rollforward*. In the simplest implementation, the service replays all records from the beginning of the log. This can be slow, so to speed up recovery, each service is expected to write periodically a consistent copy of its data structures followed by a special record called a *checkpoint*. A checkpoint denotes a point in the log at which the service's data structures are consistent; thus, no records older than the most recent checkpoint need to be replayed after a crash. These older records are made obsolete by the checkpoint, and implicitly deleted by the service when the checkpoint is written. The log layer tracks the most recently written checkpoint for each service and makes it available to the service on restart, enabling the service to get back to its last known consistent state quickly.

In the event of a crash, the log layer provides each service with the records the service wrote after its most recent checkpoint; these records represent changes made by the service that are not included in the checkpoint's consistent state, but that may be recoverable. By replaying these records and applying the changes they represent to the checkpoint's state, the service can reconstruct its state at the time of the crash.

It is important to note that checkpoints are merely an optimization. In the absence of checkpoints, each service simply rolls forward from the beginning of the log each time it restarts. Checkpoints enable a service to restart without having to search for the end of the log after a clean shutdown, and their frequency establishes an upper bound on recovery time after a crash. Checkpoints also simplify the cleaning process considerably, by implicitly deleting obsolete records.

2.1.4. Cleaning

The log is infinite, but physical storage is not. Fortunately, portions of the log become unused as blocks are deleted and overwritten, and records become obsolete. This free space can be reclaimed to prevent the system from running out of space in which to store new log fragments. As in other log-structured storage systems, Swarm reclaims this free space using a *cleaner* process that periodically traverses the log and moves live data out of stripes by appending them to the log, so that the space occupied by the stripe can be used to store a new stripe [3].

A block is cleaned by appending it to the log, changing its address and requiring the services that wrote it to update their 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. The creation record contains service-specific information that makes it easier for the service to find the block in its metadata and update its location. For example, the creation record for a file block might contain the inode number of the block's file, and its position within the file.

Records that are newer than the most recent checkpoint will be replayed after a crash and must not be cleaned; the cleaner therefore only cleans stripes whose records have been implicitly deleted by a more recent checkpoint. The dependence of the cleaner on the service checkpoints has the unfortunate consequence that a malicious or poorly-written service can prevent the cleaner from making progress simply by never writing checkpoints, or by writing them very infrequently. This in turn will cause the system to come to a halt when it runs out of free stripes to hold new portions of the log. We mitigate this problem by forcing services to write out checkpoints on demand; if a service ignores a request by the log layer to write out a checkpoint, it does so at its own peril, as its records will be reclaimed and not be replayed after a crash.

2.2. Services

The log provides rather limited functionality, and is probably not useful to application programs directly. Blocks and records can only be appended to the log, causing the log to eventually exhaust physical storage. Swarm provides additional functionality for application programs by layering *services* on top of the log. Each service can extend and/or hide the functionality of the services on which it is stacked. An example is the log cleaner. The cleaner is a service that is layered on top of the log, reclaiming unused portions of the log by moving blocks. Implementing the cleaner as a service, rather than integrating it into the log, allows the cleaner to run as a user-level process and store its

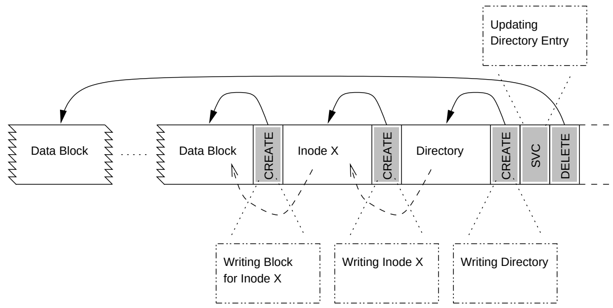

Figure 2. The log in Figure 1 as seen by a file system.

data structures in the log. This is similar to running the LFS cleaner at user-level, and enjoys the same advantages [15].

Other examples of possible services are an atomic recovery unit (ARU) [6] service that provides atomicity across multiple log writes; a logical disk [4] service that provides a disk abstraction that hides the append-only log, allowing higher-level services and applications to overwrite the blocks they store; a caching service that caches log data in main memory; an encryption service; a compression service; etc. Distributed services, such as distributed file systems and distributed cooperative caching [14], can also be layered on the base Swarm functionality.

A service modifies the functionality of the services below it by intercepting communication between those services and the services above. For example, the ARU service operates by intercepting records written by higher-level services. The records are tagged with the ARU to which they belong, and passed to the service below. During recovery, the replayed records are passed up from the lower service; the ARU service only relays upwards those records that belong to ARUs that completed before the crash. The ARU also extends the lower services' functionality by providing routines to manipulate ARUs (e.g., starting and ending ARUs).

2.3. Storage server

A Swarm storage server is specialized software that runs on commodity hardware. Swarm servers are designed to optimize cost-performance, rather than absolute performance: the desired absolute performance of the system is obtained by aggregating Swarm servers.

A Swarm storage server is merely a repository for log fragments; clients store fragments on a server, read from them, and delete them when they are no longer needed. As a result, a storage server is little more than a virtual disk that provides a sparse address space, with additional support for client crash recovery, security, and fragment reconstruction. The fragment operations supported by the server consist of storing data in a fragment, retrieving data from a fragment, deleting a fragment, preallocating space for a fragment, and querying the FID of the last marked fragment. Marked fragments are denoted as such when they are stored. The storage server's simple functionality makes it suitable for implementation on a networked-attached storage device (NASD) [5].

2.3.1. Client crash recovery

The storage server has two features that support client crash recovery. First, the client can mark a fragment when it is stored, and subsequently query the server to find the newest marked fragment. This allows a client to find its checkpoint by storing checkpoints in marked fragments. Second, all storage server operations are atomic. All records written in the same store operation will either exist or not after a crash, so that during recovery a client does not have to deal with partially-written records or checkpoints. If the write of a checkpoint fails because of a crash, the client simply recovers from the previous checkpoint and rolls forward from there.

2.3.2. Security

The storage server provides a simple security mechanisms that allows clients to protect the data they store from one another. The storage server supports access control lists (ACLs) that provide read and write protection on data in fragments. The server maintains a database of ACLs, indexed by an ACL ID (AID). The server provides routines for creating, modifying, and deleting ACLs.

When a fragment is stored each non-overlapping byte range can be be assigned an AID. Subsequent accesses to a byte range will only be permitted if the requesting client is a member of the ACL. ACLs are associated with byte ranges, instead of blocks or records, because the storage servers are not aware of these high-level abstractions. To a storage server, a log fragment is simply an opaque set of bytes.

Once written, the data's AID cannot be changed; instead, access permissions can be changed by changing the members of the ACL indicated by the AID. This makes it easy to add a client to the system with the same privileges as existing clients; once the client has been added to the appropriate ACLs, all data protected by those ACLs will be accessible.

2.3.3. Fragment reconstruction

One of Swarm's most important features is its ability to tolerate server failures by reconstructing unavailable data. Swarm uses the same error-correcting code mechanism as RAIDs, but adapted to a distributed storage system instead of a locally-attached device. Although reconstructing a fragment is relatively simple, consisting of fetching all remaining fragments in the same stripe and XOR'ing them together to produce the missing fragment, merely finding the remaining fragments is difficult in a distributed system. Reconstruction of a fragment requires knowing which other fragments are in the same stripe, and on which servers they are stored.

In Swarm, the clients reconstruct the fragments they need. Servers do not participate in reconstruction directly; reconstruction is transparent to the servers, not the clients. Reconstruction on the client is made possible 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. A client finds fragment N-1 and N+1 by broadcasting to all storage servers. Once the client locates a fragment in the same stripe as the fragment to be reconstructed, it uses the stripe group information in that fragment to access the other fragments in the stripe and perform the reconstruction. 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.

3. Prototype

We have built a prototype of the Swarm system complete with log layer, cleaner, storage server, and a local file system called Sting. The prototype uses personal computers running the Linux operating system, connected via a switched Ethernet.

3.1. Sting

To demonstrate that file systems can be built efficiently using Swarm, we implemented a local file system called Sting. Sting is local in that each instance is confined to a single client; Sting does not support file sharing between clients. Instead, it provides the standard UNIX file system interface as if the file system were stored on a local disk. The file system data are actually stored in Swarm, providing the client with Swarm's scalable performance and reliable operation.

Sting borrows heavily from Sprite LFS [13], although it is smaller and simpler than Sprite LFS because it doesn't have to deal with log management and storage, cleaning, or reconstruction, all of which are handled by lower-level Swarm services.

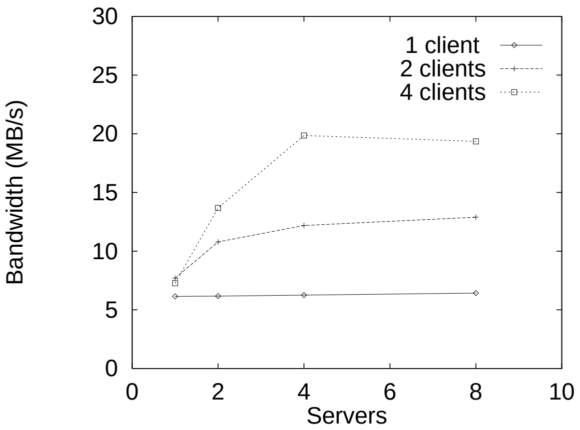

Figure 3. Raw write bandwidth. This graph shows the aggregate bandwidth of writing 10,000 4KB blocks to the log, including the overhead of writing the log metadata and the parity fragments. Each data point is the average of three experiments.

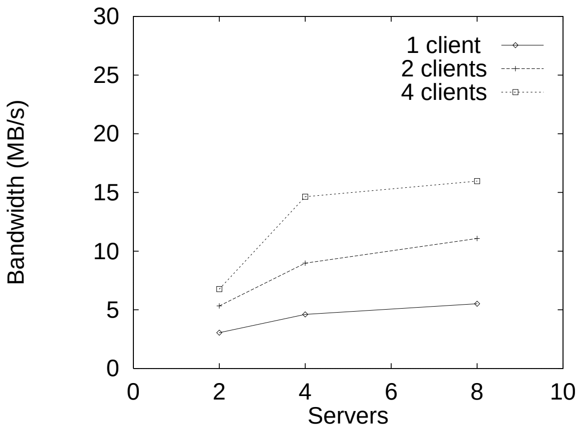

Figure 4. Useful write throughput. This graph shows the aggregate bandwidth of writing 10,000 4KB blocks to the log, as seen by the application. Each data point is the average of three experiments.

3.2. Storage server

The prototype Swarm storage server is implemented as a multi-threaded, user-level process on the Linux operating system. The server divides its disk(s) into fragment-sized slots, one for each fragment. A mapping from FID to slot is maintained in an on-disk fragment map. This architecture is based on the Zebra storage server's [7] and implements many of the same optimizations. ACLs are not currently supported.

A client accesses a server through TCL [10] scripts. Thus, reading and writing fragments are effected by sending the server an ASCII TCL script that reads or writes the appropriate data. The use of TCL made it easy to debug and extend the server interface, and because every fragment operation involves a disk access, the overhead of using TCL is inconsequential. In addition, using TCL effectively turns the storage server into an Active Disk [1][12], although we have made little use of this functionality other than for debugging purposes.

3.3. Experimental setup

The prototype's clients and storage servers are 200 MHz Pentium Pro-based machines with 128 MB of memory, connected via 100 Mb/s switched Ethernet. The operating system is Linux 2.2.2, modified to support a write-back page cache. Each storage server contains a Quantum Viking II SCSI disk dedicated to holding log fragments. The size of a log fragment is 1 MB. The storage server can write fragment-sized blocks to the disk at 10.3 MB/s, providing an upper bound on the server performance.

3.4. Experimental results

We measured the write performance of both the raw log layer and Sting. Read performance was not measured extensively, as we expect most reads to be handled by the client cache; the prototype servers do not cache log fragments in memory, and the clients do not prefetch blocks from the servers. Both of these optimizations would greatly improve the performance of reads that miss in the client cache. As a result, a Swarm client can read 4KB blocks from the servers at only 1.7 MB/s.

Log layer write performance was measured using a simple microbenchmark that wrote 10,000 4KB blocks into the log, then flushed the log to the storage servers. We measured both the raw aggregate throughput of the system (Figure 3) as well as the useful aggregate throughput (Figure 4). The former measures the rate at which the clients write to the servers, including not only the actual data written by the benchmark, but also all log metadata and the log parity fragments. The latter measures the bandwidth of writing only the data written by the application, and hence represents the bandwidth that is useful to an application.

The raw write bandwidth of a single client is 6.1 MB/s. This nearly saturates the client, so that performance improves only slightly as servers are added, reaching 6.4 MB/s with eight servers. A single server is capable of sustaining 7.7 MB/s, a rate that is achieved when more than one client writes to the same server. Configurations with more

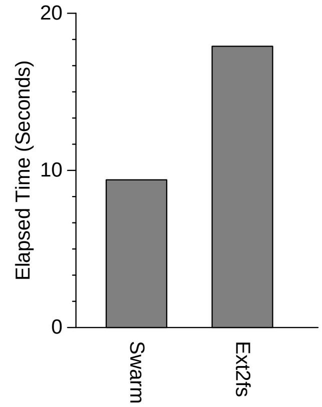

Figure 5. Modified Andrew Benchmark. This shows the elapsed time to complete the Modified Andrew Benchmark. Sting accesses a single storage server via the network; ext2fs accesses a local disk. Each data point is the average of three experiments.

than one client can benefit from additional servers: two clients achieve 12.9 MB/s with eight servers and four clients achieve 19.3 MB/s.

The useful aggregate write bandwidth reflects the bandwidth seen by services using the log. When measuring useful bandwidth, the minimum system configuration consisted of a single client and two servers, one to store data and the other parity. The parity and log metadata overhead results in a useful bandwidth of 3.0 MB/s for a single client and server. Performance improves as servers are added because as the width of the stripe increases, the cost of computing and writing the parity fragment is amortized over more data fragments. Performance approaches, but will not achieve, raw bandwidth; with an infinite number of servers, the parity overhead drops to zero, leaving only the log metadata overhead as the difference between the raw and useful bandwidths.

Configurations with two and four clients exhibit this same trend—performance increases as servers are added and parity is amortized over more fragments. With four clients and eight servers, the aggregate bandwidth is 16 MB/s, only 17% less than the raw bandwidth.

The Modified Andrew Benchmark [11] measures the performance of typical file operations, such as copying files, traversing a directory hierarchy, compilation, etc. We unmount the file systems as part of the benchmark to ensure that the data written are eventually stored to disk. Ext2fs is the local file system for Linux, providing a baseline comparison for Sting. Sting was configured with a single client and a single server, so that the data written by Sting were stored in the client's log and written across the network to the server.

Sting outperforms ext2fs by nearly a factor of two, completing the benchmark in 9.4 seconds as compared to ext2fs's 17.9 seconds (Figure 5). Sting makes much better use of the disk by writing data sequentially to the log and writing the log to the disk in 1 MB fragments. Swarm's poor read performance is masked by the client-side cache. As a result, Sting achieves 93% CPU utilization, while ext2fs is more disk-bound and achieves only 57% utilization. Sting's better use of the disk more than offsets any network overhead, and allows Sting's performance to scale better with improvements in processor speed.

4. Related work

There are many projects whose work is similar to Swarm's. Zebra [7] and xFS [2] are both distributed file systems that use striped logging to store file data on a collection of file servers. xFS supports many features that Zebra does not, including stripe groups, distributed file management, and server-less systems in which clients cooperate to provide file service. Both of these systems differ from Swarm in that they are complete file systems, rather than low-level storage systems. Swarm also retains a division of responsibility between the clients and servers unlike xFS, but like xFS pushes most file system functionality to the clients.

Petal [9] is a distributed storage system that provides virtual disk abstractions. The Petal servers cooperate to provide a consistent view of a virtual disk to the clients, so that clients accessing the same disk block see the same contents. Petal also provides a distributed locking service for concurrency control.

Frangipani [16] is a distributed file system built on Petal. Frangipani clients run local file systems modified to access Petal for disk storage and for locking the file system metadata. A file system such as Frangipani could be implemented as a Swarm service, although we have yet to do so.

The NASD [5] project is developing network-attached disks. These disks will coordinate to provide file service to the clients. The disks provide an object-oriented interface, so that files consist of collections of objects stored on different disks. This is in contrast to Swarm, in which the storage servers provide a very low-level fragment-oriented interface.

Stackable file systems [8] provide extensible file system functionality by allowing file systems to be stacked. Each file system in the stack provides a vnode interface to the file system above, and expects a vnode interface from the file system below. Swarm places no restrictions on the interfaces between the services layered on the log, enabling interfaces that are not based on vnodes.

5. Conclusion

Swarm is a scalable network storage system that provides scalable, reliable, and extensible storage service. Swarm servers are commodity machines, aggregated to provide high-performance storage service. The Swarm infrastructure on the clients consists of layered services, allowing applications to pick and chose the exact services needed. Our measurements show that performance does scale with the number of servers until the point where the clients are the bottleneck. The useful write bandwidth of a single client is 3.0 MB/s with two servers and 5.5 MB/s with four; for four clients, the corresponding numbers are 6.7 MB/s and 16.0 MB/s. Sting also outperforms ext2fs on standard file system operations, completing the Modified Andrew Benchmark almost twice as quickly.

6. Acknowledgements

This work was supported in part by DARPA contracts DABT63-95-C-0075 and N66001-96-8518.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms, and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-98)*, pages 81–91. ACM Press, Nov. 1998.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, and R. W. D. Roselli. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP-95)*, pages 41–79. ACM Press, Dec. 1995.

[3] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the 1995 USENIX Technical Conference*, pages 277–288. USENIX, Jan. 1995.

[4] W. De Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: a new approach to improving file systems. *Operating Systems Review*, 27(5):15–28, Dec. 1993.

[5] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS-97)*, pages 272–284. ACM Press, June 1997.

[6] R. Grimm, W. C. Hsieh, M. F. Kaashoek, and W. de Jonge. Atomic recovery units: Failure atomicity for logical disks. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-96)*, pages 26–37. IEEE, May 1996.

[7] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, Aug. 1995.

[8] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, Feb. 1994.

[9] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-96)*, pages 84–92. ACM Press, Oct. 1996.

[10] J. K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the Winter 1990 USENIX Conference*, pages 133–146. USENIX, Jan. 1990.

[11] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256. USENIX, June 1990.

[12] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proc. of the 24th International Conference on Very Large Databases (VLDB-98)*, pages 62–73, Aug. 1998.

[13] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th Symposium on Operating Systems Principles (SOSP-91)*, pages 1–15. ACM Press, Oct. 1991.

[14] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI-96)*, pages 35–46. USENIX, Oct. 1996.

[15] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the Winter 1993 USENIX Conference*, pages 307–326. USENIX, Jan. 1993.

[16] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, pages 224–237. ACM Press, Oct. 1997.