Design and Implementation of the Swarm Storage Server

Rajesh Sundaram

TR 98-02

March 11, 1998

Abstract

The Swarm storage system uses log-based striping to achieve high performance. Clients collect application writes in a log and stripe the log across multiple storage servers to aggregate server bandwidth. The Swarm storage server has been designed to meet various requirements of the Swarm storage system. These include high performance for data-intensive operations, rapid crash recovery, security support and atomic interface routines. The design of the Swarm storage server is simple enough to allow its implementation as a network appliance.

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

1 Introduction

Today's network file systems suffer from performance problems that limit their scalability. File servers often become a performance bottleneck when a system has to scale beyond a few dozens of nodes. Network file systems like NFS [Sandberg85] try to deal with scalability by having multiple servers handling different parts of the directory hierarchy. This mapping of files to servers is static and suffers from the problem of balancing load among the servers. For example, system directories often reside on the same server, making that server a hotspot.

Another problem with current network file systems is availability. The centralized nature of the file server leads to a potential single point of failure. A file server crash makes the files it stores inaccessible. Even worse, a hardware failure could cause loss of data.

Swarm is a network storage system that aims to solve these problems using three techniques. First, it uses log-based data striping to provide high throughput by aggregating bandwidth of multiple storage servers. Second, computed redundancy is used for increased reliability. Third, by pushing most of the functionality to the clients and keeping the servers simple, overall system scalability is increased.

Swarm is a storage system, as opposed to a file system. Different storage abstractions can be configured above the storage system, making Swarm more flexible. For example, Swarm can simultaneously support Sun's NFS, HTTP and its own specialized interface. The design goal is to achieve this flexibility without sacrificing performance.

The Swarm storage server is the component of Swarm that serves file data. Being a performance-critical component of Swarm, the Swarm server has been designed to achieve high data throughput by using two approaches. First, a log-structured design is used to improve the performance of small writes. Second, disk reorganization is used to improve the performance of reads. The Swarm server also has other features which help clients in implementing the additional functionality Swarm expects from them. These include an atomic interface, security facilities, logging support, etc.

The Swarm server has support for mobile code in the form of a built-in Tcl shell. Clients can download programs and execute them on the server. This, in addition to making the server interface more flexible, allows applications to be designed to run partly on the storage server, reducing the need for data transfers across the network.

2 Swarm

Swarm is a network storage system that is designed to achieve high performance that scales with the number of nodes in the system. The scalability applies not only to aggregate performance, but also to the performance of individual I/O operations.

Swarm achieves high performance by using data striping, similar to RAID. Unlike RAID, the striping is done across the network. Application writes are broken up and written to multiple storage servers simultaneously to aggregate bandwidth of the servers. While some systems stripe individual files to achieve high performance [Long94], Swarm takes a different approach. To avoid problems with handling small files and modifications to existing file blocks, Swarm instead uses log-based striping.

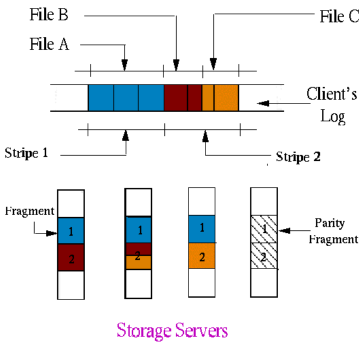

Figure 1: **Log-based striping.** Files A, B and C are collected in a log and the log is striped across the servers. File A is large enough to fill an entire stripe, while files B and C together fill a stripe. Parity information is stored for each stripe.

Figure 1 illustrates log-based striping used in Swarm. Clients collect application writes in append-only logs. Each client maintains its own log. Large segments of the log are then striped across storage servers. Similar to RAID terminology, the term *stripe* refers to the collection of data that spans the storage servers. The piece of the stripe that an individual server stores is called a *fragment*. This approach to storing data makes the I/O performance in Swarm scale as a function of the number of storage servers.

Another of Swarm's goals is high reliability. In order to achieve this, Swarm clients compute and and store a parity fragment for each stripe. If a single server crashes, the system will be able to continue operation by computing the missing data using the parity. This downside of this is slightly increased storage space usage.

Swarm is a storage system as opposed to a file system. Clients are provided an interface based on logical blocks. Swarm, by providing a block level interface, requires clients to handle various file management tasks. Swarm clients are responsible for coordinating their actions when allocating and freeing storage space, for managing metadata, file naming, protection mechanisms, etc. There are two advantages to this approach. First, the system is more scalable because the file management tasks are handled in a distributed fashion by the clients themselves, eliminating the file server bottleneck. Second, clients can use file systems designed to suit their requirements, without having to use a general purpose file system that has not been optimized for any particular usage.

2.1 Swarm Architecture

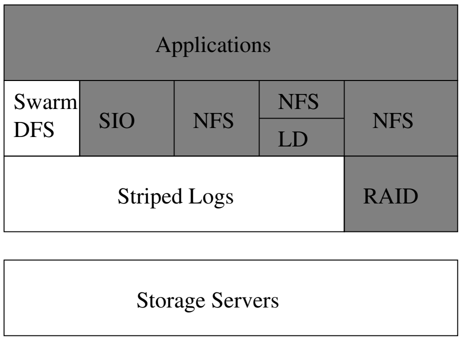

Figure 2: **Swarm layered architecture.** The unshaded portion is the part of the system that Swarm implements. Access protocols like NFS, RAID and the Scalable I/O Initiative (SIO) API may be configured above Swarm. A logical disk (LD) may be constructed as a separate layer.

Swarm has a layered architecture that can accommodate vastly differing configurations of the system. Figure 2 shows Swarm's layered architecture and how Swarm can accommodate different access protocols and storage abstractions. The unshaded portion of the figure is the part of the system that Swarm implements. This consists of the storage servers, the Striped Logs layer that implements log-based striping, and the Swarm Distributed File System (DFS). Other modules can be configured into the system. For example, the figure shows how an NFS server can be configured directly over the log layer. The figure also shows how standard NFS can be used without modifications, by constructing a logical disk between NFS and the log layer. As another example, the Scalable I/O Initiative (SIO) API [Bershad94] can be used to access the logs. Although the discussion so far assumes that clients have to go through the log layer to access the storage servers, the last example shows how it is possible for clients to bypass the log layers and implement RAID directly on the servers.

3 Server Requirements

Swarm requires certain features from the storage servers. These features are important to Swarm, in terms of how they affect overall system performance and client complexity. The following are the Swarm requirements that influenced the design of the storage server.

> **High performance for whole-fragment writes.** One of the most important storage server requirements is the need for high performance for large writes. This stems from the fact that writing whole fragments is a very common operation Swarm clients perform on the server. The size of these fragments must be large enough to

minimize network and disk overheads associated with the data transfer. Currently Swarm uses fragments of size 512 KB.

Good performance for small writes. There are two reasons why small writes can happen. One is that Swarm allows clients to access the server using protocols that are not log-based. These protocols, such as RAID, may not perform large writes. The other reason for small writes is application flushes. While flushes do not happen frequently with typical non-transaction processing workloads [Baker92], Swarm needs to take special care to handle them efficiently. This is because Swarm supports other storage abstractions such as parallel file systems, which may exhibit different behavior. A flush performed by an application running on the client causes the client's logging layer to write out an incomplete stripe to the servers. This may cause a write to one of the servers that is smaller than a full fragment (the last incomplete fragment). But more significantly, the client may append to the stripe at a later point and this would require the parity fragment to be updated - likely a small write.

Atomic interface. Parity updates during appends also pose another important requirement. A storage server crash during a parity update should not leave the old parity corrupted. To recover by discarding data appended to the stripe, the server needs the old parity. While the client can provide the functionality to perform updates in a non-overwrite fashion, it would simplify client design if the storage servers provided an efficient atomic modify operation.

Rapid crash recovery. A configuration of Swarm can have a large number of servers, and this increases the probability of server crashes. A single storage server being down causes degraded performance. Two are more servers being down causes the system to become unavailable. Hence rapid recovery from crashes is an important requirement in meeting Swarm's performance and availability goals.

Security support. A Swarm configuration may consist of multiple clients that do not trust one another. In such an environment, clients require server support to protect their data and to share data with other clients in a controlled manner. For example, a client implementing a local file system over Swarm may wish to prevent all other clients from accessing it. A group of clients implementing a distributed file system may wish to share the data amongst themselves without allowing other clients access to the data.

4 Swarm Server Interface

The interface to the Swarm server can be separated into an fragment interface and an administrative interface. The fragment interface to the Swarm server consists of operations on fragments, the units used for striping. Fragments are addressed using fragment IDs. The fragment interface has commands that allow clients to create, read, modify and delete fragments. Two other commands in the fragment interface, *prealloc* and *marked*, provide logging support. The commands are listed in Table 1.

The administrative interface to the Swarm server has commands to set various security parameters, a command for gathering statistics, and a command to shutdown the server. The security provisions in the Swarm server and the security interface are discussed in the next Section.

Command	**Description**
Store(FID, flags, offset, length, securityInfo, data)	Write a range of bytes given by offset and length
Retrieve(FID, offset, length, data)	Read byte range from fragment specified by offset and length
Delete(FID)	Delete fragment and free up storage space
Preallocate(FID)	Reserve one fragments worth of space
Marked()	Return ID of last fragment marked by a client

Table 1: **Swarm server fragment interface.** *FID* stands for the fragment ID. The fragment ID is a 64 bit quantity. The Swarm server fragment interface provides three commands - *store*, *retrieve* and *delete* - to create, access, modify and destroy fragments. Two other commands, *prealloc* and *marked* provide logging support.

Store, *retrieve* and *delete* are the three main commands in the Swarm server's fragment interface. The *store* command is used by clients to write data to the servers. The client specifies as parameters to the command the fragment ID, the offset into the fragment, and the length of the write. There are two other parameters, *securityInfo* and *flags*. The *securityInfo* parameter contains permission settings for the byte range defined by the offset and the length. The *flags* parameter is currently used only for one flag, the *mark* flag that tags a fragment for recovery. The *store* command allows clients to write a full or a partial fragment. It also allows modification to existing data.

The *retrieve* command is used by clients to read back stored data. It takes an offset and a length as parameters and returns the range of bytes that corresponds to the offset and length.

The *delete* command deletes the fragment specified by the fragment ID in the request. It frees up all the space used for storing the fragment data and metadata. After a fragment is deleted its ID may be reused. The *delete* command is used during *stripe cleaning*. Stripe cleaning is the process by which free space is reclaimed from the logs. During stripe cleaning, live data in a stripe is identified and copied to a new stripe by client cleaner processes. After the data has been copied, the fragments belonging to the stripe are deleted.

The *preallocate* command reserves a fragment's worth of storage space on the server. It is primarily used by the stripe cleaner to ensure that a *store* command will not fail because of lack of space. The server maintains a list of preallocated fragments for every client. If a client crashes, the server frees all fragments preallocated by the client. A server crash causes all fragments preallocated on the server to be freed. The *preallocate* command can also be used to support client file system semantics. Many file systems do not tolerate the failure of a write operation due to lack of space. The *preallocate* command can be used to protect against such a failure.

The *marked* command simplifies client crash recovery by providing clients a convenient and efficient way of finding checkpoints. Clients mark fragments containing checkpoint information by setting a flag when invoking the *store* command. The *marked* command returns the ID of the last fragment marked by the client making the call. The client can use the fragment ID to read the checkpoint information or to find the end of the log.

4.1 Security Provisions

The Swarm server has provisions to provide security based on Access Control Lists (ACLs). An ACL consists of a list of client IDs and is associated with a range of bytes in a fragment. This association is set during a *store* command when the data is first written. A read, modify or delete operation on a range of bytes is allowed only if the client performing the operation is on the ACL associated with the range. A modify operation cannot change the ACL ID associated with a byte range.

A client ID is associated with one or more IP addresses. Multiple IP addresses may be used, for example, to accommodate all the IP addresses of a multi-homed host or to form a client group.

Clients can use the ACL based security mechanism to protect data from other clients and from external hosts. This is a flexible mechanism that can accommodate different security policies. For example, a client could create a ACL consisting of itself and a a trusted client running a cleaner process. By associating this ACL with the data it writes, the client can ensure that no other host will be allowed access to this data.

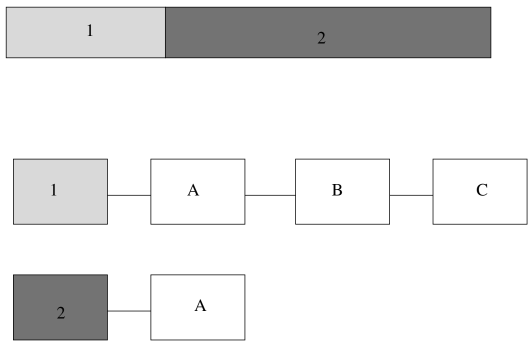

Figure 3: **ACLs example.** The portion of the fragment marked with ACL ID 1 and can be accessed by clients A, B and C. The portion marked with ACL ID 2 can be accessed only by client A.

Figure 3 gives an example of how ACLs are used. The *store* command marks the first part of the fragment with ACL ID 1 and the second portion with ACL ID 2. Clients A, B or C can access and modify the first portion. Only client A will be allowed to access or modify the second portion.

Command	Description
aclSet(aclId, clientIdList)	Sets the client IDs for an ACL. If the ACL already exists, it is first cleared. If the ACL does not exist, a new one is created.
aclAdd(aclId, clientIdList)	Add client IDs to ACL. If the ACL does not exist, a new one is created.
aclRemove(aclId, clientIdList)	Remove client IDs from ACL.
aclDestroy(aclID)	Destroy ACL
clientIdSet(clientId, ipAddressList)	Set IP addresses for a client ID. A pre-existing client ID is first cleared.
clientIdAdd(clientId, ipAddressList)	Add IP addresses to client ID.
clientIdRemove(clientId, ipAddressList)	Remove IP addresses from client ID.
clientIdDestroy(clientId)	Destroy client ID.

Table 2: **Swarm server security interface.** The Swarm server security interface offers commands to set and modify the membership of an ACL. The members of an ACL are identified using client IDs. A client ID may be associated with one or more IP address.

The server security interface, listed in Table 2, offers commands for setting the members of an ACL, adding to a ACL and removing members from ACLs. It also lists commands to set the IP addresses associated with a Client ID.

5 Design Choices

While designing the swarm server we considered some straightforward designs that we might be able to use. We discuss two of them and the reasons why we decided against using them.

Network disk. A simple design is to use network attached disks as storage servers. In this scheme, clients write directly to disks attached to the network. While it is conceivable how these disks could be used to perform large writes efficiently (we could just partition the disk into fragment-sized blocks), small writes pose a problem. The overhead of seeks between each small write restricts the achievable performance to a fraction of the maximum disk bandwidth. Another problem with this approach is that all additional functionality like disk management, atomicity, etc., has to be implemented by the clients. In many cases it is more suitable to keep the functionality at the servers. Security is one example. Unless we fully trust the clients we cannot move security support to the clients.

File server. Another possible design for the Swarm server is to use a standard file system like UFS. Using this approach we could store fragments as files. This approach also has some problems. One of the problems is that most file systems don't achieve very good write performance [Rosenblum91]. We could use a write optimized file system like the Sprite LFS, but there is a loss of efficiency and added complexity due to the maintenance of file-system related metadata for file structures, permissions, directories, etc. Perhaps the biggest problem with this approach is that standard file systems are constructed as a part of general purpose OS kernels. There is a cost, is terms of complexity and performance, due to the generality of these systems that make them unsuitable for a specialized application like the Swarm server.

6 Swarm Server Prototype

We have implemented a prototype of the Swarm server. The prototype has a two-layered architecture. The upper layer implements the Swarm server functionality. The lower layer, a log-structured implementation of the Logical Disk (LD) [deJonge93] manages the disk subsystem. The Swarm server is constructed as a part of the Scout operating system [Montz94].

6.1 Logical Disk

Logical Disk (LD) is a software layer over a raw disk that provides an abstract interface to the disk. This interface is based on logical block numbers and block lists. The primary motivation for developing LD came from modern OS kernels that can support multiple file systems. LD's interface, by separating file management from disk management, allows different file systems to share a single LD implementation.

File systems use logical block numbers to address data blocks. The logical address of a block is independent of the physical location where LD chooses to place the block on disk. LD takes advantage of this by transparently reorganizing data on disk during cleaning to improve read performance.

Block lists are lists of logical block numbers and are used to provide LD with block layout hints. For example, blocks can be ordered in a block list according to the expected read pattern. During reorganization, LD will lay the blocks on a list sequentially.

LD also provides an abstraction called Atomic Recovery Units (ARUs). During recovery, all operations performed as a part of an ARU are treated as a single indivisible operation. That is, LD either recovers to a state that existed after all the operations were performed or a state that existed before any of the operations began. The latter scenario happens if some of the operations were lost due to the crash.

LD's log-structured architecture has two advantages, particularly from Swarm's perspective. First, it provides improved small write performance. Because small writes are batched together before being written, almost all seeks are eliminated. Second, it simplifies crash recovery. By check-pointing the state of the storage server periodically, we only need to scan a recent portion of the log during recovery as opposed to scanning the entire disk.

6.2 Use of the Logical Disk

As shown in Figure 4, clients address data on the Swarm server with a Fragment ID-offset pair. The Swarm server interface function maps this to a logical block number and passes it to LD. LD maps the logical block number to a physical address which it uses to access the disk.

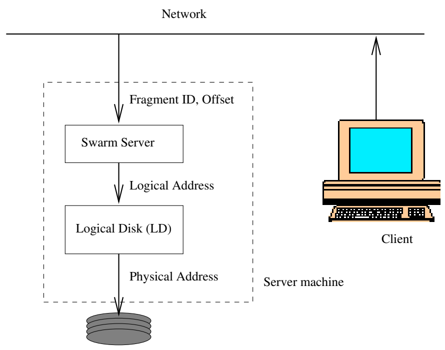

Figure 4: **Data access using the Logical Disk.** The client passes to the server interface routine the fragment ID and offset which gets mapped to a logical block number. LD maps the logical block number to a physical block number.

The features of LD fit in very well with requirements of the Swarm server. The Swarm server uses LD's ARUs to make all the interface operations atomic. All disk operations that happen during an interface operation are placed within the same ARU. LD ensures that these operations are treated as indivisible; that is, if there is a crash, either all these operations will become persistent or none will. Apart from parity updates, clients can make use of the atomic interface to simplify the implementation of file system consistency. For example, to create a new file the client could combine into one operation the write of file data with the update of metadata, eliminating the need for consistency checks during recovery.

The Swarm server makes use of LD's block lists to lay blocks effectively. All blocks belonging to a fragment are placed in the same LD block list, in order. During disk reorganization, LD will cluster fragment blocks that may have become scattered due to modify operations. This improves the performance of sequential fragment reads.

6.3 The Scout Operating System

Scout is a configurable, communication-oriented operating system (OS). It provides infrastructure to build specialized OS code targeted at specific applications. An instance of the Scout OS is built by configuring in just the modules that are required for that particular system.

Unlike the process abstraction in a conventional multi-processing OS, Scout makes *paths* its central abstraction. Paths intuitively are extensions of the physical network into the host system. Since Scout is communication oriented, its principle task is the transfer of data between I/O source and I/O sink. Paths provides the mechanisms and resources necessary for such data movement. In Scout, paths, rather than threads, are the schedulable entity.

The path abstraction has many benefits. It facilitates a design that identifies and optimizes the critical path I/O data follows. Paths also serve as a unifying mechanism for resource management. In contrast to conventional operating systems, where the CPU is the only schedulable resource, Scout allocates all resources like memory, I/O bus bandwidth, etc., on a per-path basis.

The configurable and communication oriented nature of Scout makes it a suitable platform for the Swarm server. The Swarm server has a simple design and its requirements are minimal. All that the Swarm server requires is access to the network and access to a disk. Scout provides the necessary infrastructure and architectural support to build such a stripped-down system with only the required functionality. The Swarm server also takes advantage of various path-based optimizations to improve its performance.

In Scout the modules that implement the different functionality are called *routers*, analogous to network routers. Scout is configured using a router graph that specifies the connections between the routers.

6.3.1 Router Graph for the Swarm Server

Figure 5 shows the Swarm server's router graph. The Swarm Server router implements the Swarm server functionality. It uses the abstract interface to the disk provided by the LD router. The Swarm Stub router, and a set of routers below it (TCP, IP, ARP etc.), implement networking functionality.

The Swarm Stub router has two functions. First, it acts as a messaging protocol. Being a stream oriented protocol, TCP may break up or batch together client request messages. The Swarm Stub router is responsible for forming the original client request messages from the packets it receives from TCP. Second, the Swarm Stub router performs any byte swapping that may be required on the request message before dispatching it to the Swarm server.

6.3.2 Path Architecture

As Figure 5 shows, there is one path per TCP connection between Swarm Stub and the network. Each of these paths store the state of the corresponding TCP connection. Resources are allocated on a per-connection basis.

The network paths map one-to-one to paths between Swarm Stub and LD. The reason for the one-to-one mapping is the fact that currently, all client calls are synchronous. That is, there can be at most only one outstanding client request per TCP connection between the client and server. Each path between the Swarm Stub and LD stores the state of the outstanding call on the connection.

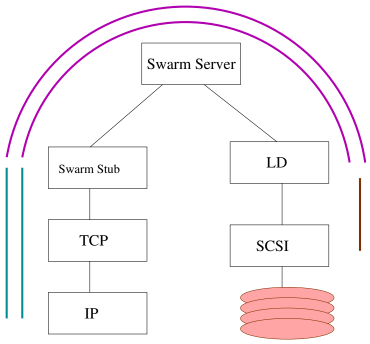

Figure 5: **Swarm server path architecture.** The Swarm server has one path per TCP connection between the network and the Swarm Stub router. These network paths have corresponding paths between the Swarm Stub router and the LD router.

Scout provides a message abstraction that provides a set of operations for manipulating data with minimal data copying. The Swarm server uses the Scout message abstraction for data manipulation and transfer between the routers. The message abstraction is also used in moving data within a router. The SCSI router interface, however, does not currently use the message abstraction. For this reason, LD copies the message data into a buffer before invoking SCSI interface routines. This is the only instance in the Swarm server where data copying is required. Upgrading the SCSI router to use the message abstraction will eliminate this requirement.

6.4 Support for Mobile Code

The Swarm server has support for mobile code in the form of a built-in Tcl shell with all standard Tcl commands. Clients can download Tcl code to be executed on the server.

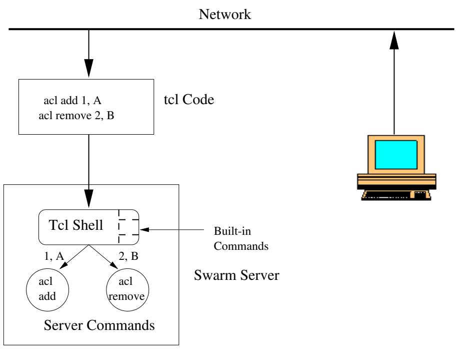

Figure 6: **Built-in Tcl Shell in Swarm server.** The Tcl shell interprets Tcl commands sent by clients and invokes the appropriate server command.

The Tcl shell is extensible and we have extended the shell with many additional commands. Currently all interface operations on the Swarm server are Tcl commands. For example, if a client wants to perform a *store* operation it sends to the server a string containing the the *store* Tcl command. The Tcl shell on the server interprets this command and invokes the appropriate routine on the server. One of the reasons we implemented all interface as Tcl commands was to demonstrate the feasibility of using the built-in Tcl shell as a platform to support mobile code. Implementing the interface as Tcl commands also minimizes source modifications when interfaces change. We have also implemented some Tcl commands for timing and gathering statistics.

Since Tcl is an interpreted language, performance is a concern. We ran tests to measure the interpretation overhead. The test results showed less than 1% overhead for interpreting the commands. The reason for this is that all the performance critical server interface routines access the disk and and the interpretation time is very small compared to disk access time.

This support for mobile code provided by the server open up new possibilities for designing file system applications. Clients can take advantage of the fact that a part of their application can run on the servers, close to the data. The savings on the amount of data transfers from servers to clients, besides leading to a more efficient design, also reduces the load on the network.

6.5 Performance Measurements

We measured the performance of data operations on the Swarm server. Figure 7 shows the results for the *store* operation and compares it to raw disk performance. The *store* operation is currently a synchronous operation and commits data to disk before completion. We were able achieve about 70% of disk bandwidth for most data sizes. The main reason for the loss of the other 30% of disk bandwidth was data copying involved. We expect to further improve performance when we upgrade the SCSI router to use the Scout message abstraction.

Figure 8 is a graph of *retrieve* results. The *retrieve* performance shows similar characteristics as *store*. Again, performance could be improved by reducing data copies. We disabled disk read caching while measuring performance to reduce variance among different measurements.

7 Related Work

This section compares the Swarm storage server with some existing storage servers.

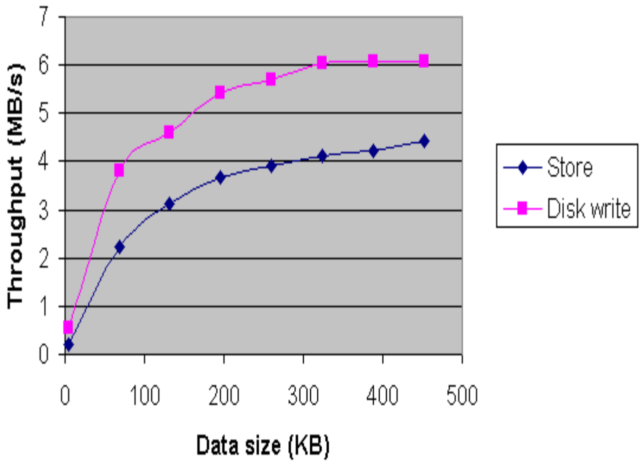

Figure 7: **Swarm server *store* performance.**

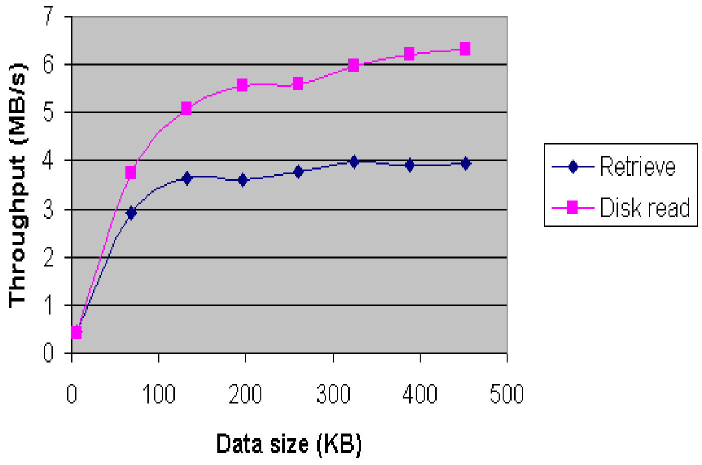

Figure 8: **Swarm server *retrieve* performance.**

Zebra [Hartman95] is a log-based, striped network file system. The storage servers in Zebra are very simple. All fragments are of the same size. Zebra stores fragments by mapping them to fixed locations on disk. Zebra allows writing of partial fragments and appending to existing fragments. Parity fragments can be modified by replacing a old copy of the fragment with a new one; partial modifies are not allowed. This simple architecture performs well for typical workstation workloads, but performance will degrade for transaction processing workloads which cause frequent writes of small partial fragments and parity updates. Zebra's storage server design makes it unsuitable for non log-based access protocols.

Petal [Lee96] is a block level storage system. Storage servers in Petal cooperatively manage a pool of physical disks, providing clients with globally visible virtual disks. A Petal server is more complex than the Swarm server. This is because Petal's servers need to maintain globally replicated data structures to implement the virtual disks. In contrast, the address space of the Swarm servers are independent of one another. In Swarm, it is the client's responsibility to locate the server that stores a block. Another difference is that Petal's servers map data blocks to fixed locations on disk. For writes we expect that the log-structured approach of the Swarm server will enable it to use the disk bandwidth more efficiently.

Network Attached Secure Disks (NASD) [Gibson96] are network attached disk drives that have built-in capability to allow clients to directly access them. The goal is to off-load much of the file manager's work onto the disk drive. Data-intensive operations like file reads and writes are sent directly to the disk. This approach has the advantage of using low-cost disks to achieve scalability. The disadvantage is that certain useful features like atomicity, disk reorganization etc., which help simplify file managers cannot be implemented on the drives because of the complexity involved. It is difficult to estimate how much computational power these drives will be able to provide. For example, it is not clear if these disks could be made to support a write optimized, log-structured layout for files.

8 Conclusion

Swarm is a flexible and scalable storage system that uses data striping for high performance. While Swarm does not expect complex functionality from the storage servers, their performance is critical to the system. Swarm performs on the storage servers data access operations with a wide range of data sizes. The Swarm storage server combines two existing ideas, log-structured file systems and virtual disks in a design that achieves high performance on these data operations. It's design is sufficiently simple to allow its implementation as a stripped down network appliance. The Swarm server also has other useful features like security support and an atomic interface. Its flexible interface and good performance on a wide variety of access patterns make it suitable choice for other systems which require high performance storage service as well.

References

[Baker92] Mary Baker, Satoshi Asami, Etienne Deprit, and John Ousterhout. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming languages and Operating Systems (ASPLOS)*, pages 10–22, October 1992.

[Bershad94] B. Bershad, D. Black, D. DeWitt, G. Gibson, K. Li, L. Peterson, and M. Snir. Operating System Support for High-Performance Parallel I/O Systems. *Scalable I/O Initiative Working Paper No. 4*, 1994.

[deJonge93] Wiebren deJonge, 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 System Principles*, pages 15–28, December 1993.

[Gibson96] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene Fienberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, and David Rochberg. A case for network-attached secure disks. Technical Report CMU-CS-96-142, Carnegie Mellon University, June 1996.

[Hartman95] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. *ACM Transactions of Computer Systems*, 13(3):274–310, August 1995.

[Lee96] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming languages and Operating Systems (ASPLOS)*, pages 15–28, December 1996.

[Long94] Darrell D. E. Long, Bruce R. Montague, and Luis-Filipe Cabrera. Swift/RAID: A Distributed RAID System. *Computing Systems*, 7(3):333–359, 1994.

[Montz94] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical Report 94-20, University of Arizona, June 1994.

[Rosenblum91] Mendel Rosenblum and John K Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 1–15, October 1991.

[Sandberg85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of the Summer 1985 Usenix Conference*, pages 119–130, June 1985.