[12] M. Rosenblum and J. K. Ousterhout, "The LFS storage manager," in *USENIX Summer Conference*, pp. 315–324, June 1990.

[13] L.-F. Cabrera, E. Hunter, M. J. Karels, and D. A. Mosher, "User-process communication performance in networks of computers," *IEEE Transactions on Software Engineering*, vol. 14, pp. 38–53, Jan. 1988.

[14] L. W. McVoy and S. R. Kleiman, "Extent-like performance from a UNIX file system," in *USENIX Winter Conference*, 1991.

A more immediate goal is to measure the effect of adding fault-tolerance on the performance prototype, particularly when a failure has occurred.

5 Conclusions

In this article we have presented a general I/O architecture for manipulating very large data objects at high data-rates. The principle behind our architecture is simple: aggregate arbitrarily many (slow) storage devices into a faster logical device, making all applications unaware of this aggregation. In our scheme the maximum data-rate of individual secondary storage devices ceases to limit the maximum data-rate that can be achieved. Known techniques of hierarchical clustering can be used to scale the performance achieved with our approach to exploit advances in technology that become available. This architecture generalizes to distributed systems current I/O channel architectures, disk striping techniques, and the proposed disk array architecture RAID.

We have modeled the data-rate behavior of this architecture in the case of multiple storage agents connected to a common local-area network, where the maximum data-rate of the network is higher than that of each of the individual storage agents. We observed good scaling properties. A simplified implementation of the architecture confirmed that large aggregate data-rates are achieved from slower I/O devices.

The transfer size of objects was found to have a significant effect on the performance of the system. This is because small transfer sizes increase the number of seeks as well as the latency observed at the disks. By using sequential storage preallocation and data staging, seeks can be minimized and performance significantly improved.

Tolerance to failures in the Swift architecture is very flexible. Each component can be hardened individually. Computed data redundancy can be used to protect against failures of the storage agents. The metadata can be hardened using standard data base techniques. Multiple interconnections can be used to guarantee alternative paths to the data and metadata of the system.

Acknowledgements

We are grateful to D. Edelson, A. Emigh, R. Golding, and M. Long for their assistance, and to L. Haas for careful reading of earlier versions of this manuscript.

References

[1] Imprimis Technology, *ArrayMaster 9058 Controller*, 1989.

[2] Thinking Machines, Incorporated, *Connection Machine Model CM-2 Technical Summary*, May 1989.

[3] P. Pierce, "A concurrent file system for a highly parallel mass storage subsystem," in *Proceedings of the* 4th *Conference on Hypercubes*, (Monterey), Mar. 1989.

[4] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the ACM SIGMOD Conference*, (Chicago), pp. 109–116, ACM, June 1988.

[5] S. Ng, "Pitfalls in designing disk arrays," in *Proceedings of the IEEE COMPCON Conference*, (San Francisco), Feb. 1989.

[6] L.-F. Cabrera and J. Wyllie, "QuickSilver distributed file services: An architecture for horizontal growth," in *Proceedings of* 2nd *IEEE Conference on Computer Workstations*, (Santa Clara, California), IEEE Computer Society, March 1988.

[7] M. Nelson, B. Welch, and J. Ousterhout, "Caching in the Sprite network file system," *ACM Transactions on Computer Systems*, vol. 6, pp. 134–154, Feb. 1988.

[8] K. Salem and H. Garcia-Molina, "Disk striping," in *Proceeding of the* 2nd *International Conference on Data Engineering*, pp. 336–342, IEEE, Feb. 1986.

[9] H. Garcia-Molina and K. Salem, "The impact of disk striping on reliability," *IEEE Database Engineering Bulletin*, vol. 11, pp. 26–39, Mar. 1988.

[10] L.-F. Cabrera and D. D. E. Long, "Swift: A storage architecture for large objects," Tech. Rep. IBM Almaden Research Center RJ7128, International Business Machines, Oct. 1990.

[11] L.-F. Cabrera and D. D. E. Long, "Exploiting multiple I/O streams to provide high data-rates," in *Proceedings of 1991 Summer Usenix Conference*, (Nashville, Tennessee), Usenix Association, June 1991.

better than access to the local SCSI disk. When compared with NFS, it provided almost twice the data-rates for reads and exceeded the NFS data-rate for writes by more than eight times. Though Swift differs from NFS significantly, this establishes the ability of Swift to aggregate data-rates of slower I/O devices.

To measure the data-rate performance of the prototype, three, six, and nine megabytes were read from and written to a Swift object. To calculate confidence intervals, eight samples of each measurement were taken. Analogous tests were performed using the local SCSI disk and the NFS file service. Maintaining cold caches was achieved by using **/etc/umount**. The three storage agents were Sun 4/20s with 16 megabytes of memory and a local 104 megabyte local SCSI disk also under SunOS 4.1.1. These hosts were placed on a 10 megabit/second dedicated Ethernet. Aside from the standard system processes, each of the servers was dedicated to the Swift storage agent software.

For Swift, the Ethernet was limiting performance factor. Using three storage agents, the utilization of the network ranged from 77% to 80% of its measured capacity of 1.12 megabytes/second. A fourth storage agent would only saturate the network while not significantly increasing performance. The NFS measurements were run over a lightly-loaded shared departmental Ethernet, not over the dedicated laboratory network. The traffic present in this shared network when the measurements were made was less than 5% of its capacity, which should not significantly affect the measured data-rates.

When compared with the local SCSI disk performance, the Swift prototype only performs between 29% and 36% better. This contrasts sharply with previous measurements taken under SunOS 4.1 where the Swift prototype performed about 250% better than local SCSI read access. This change is due to a significant improvement in the SunOS file system under SunOS 4.1.1 [14]. In contrast to its read performance, when writes are considered, the Swift prototype shows between a 274% and a 280% increase over that of the local SCSI disk. The ideal performance improvement would have been 300% if the interconnection medium were not limiting performance. Since its performance is less than 300% of the local SCSI performance, this supports the assertion that the factor most limiting its performance is the Ethernet.

When the Swift prototype is compared with the high-performance NFS file server, its performance is between 180% and 197% better in the case of reads. This shows that Swift can successfully provide increased I/O performance by aggregating several low-speed storage agents and driving them in parallel.

In the case of writes, the Swift prototype performs between 767% and 809% better than the high-performance NFS file server. When interpreting these measurements one should also keep in mind that the **write** data-rate measurements in NFS reflect the write-through policy of the server. We have not yet implemented a write-through policy for the Swift prototype. This makes data-rates for **write** somewhat difficult to compare with those of Swift.

To determine the effect of doubling the data-rate capacity of the interconnection, we added a second Ethernet segment between the client and additional storage agents. This second network segment is shared by several groups in the department. During the measurement period its load was seldom more than 5% of its capacity.

We did not expect to obtain data-rates twice as great as those using only the dedicated laboratory network since we expected the network subsystem of the client to be highly stressed. To our surprise, our measurements show that for **write** operations the Swift prototype almost doubled its data-rate.

In the case of reads, the increase in performance of the Swift prototype is less pronounced. This can be attributed to several factors including the increased load on the client, a lack of buffer space, and to the increased complexity of the read protocol which requires many more packets to be sent than does the write protocol.

These measurements demonstrate that the Swift architecture can make immediate use of a faster interconnection medium and that its data-rates scale accordingly.

The prototype demonstrates that the Swift architecture can achieve high data-rates on a local-area network by aggregating data-rates from slower data servers. The prototype also validates the concept of distributed disk striping in a local-area network by providing data-rates higher than both the local SCSI disk and the NFS file server.

4 Future Work

One area of the Swift architecture that requires further work is eliminating the requirement that resources be preallocated. We are investigating ways to apply real-time scheduling techniques to the problem of providing performance guarantees.

A second area of future work is that of co-scheduling resources. The support of continuous multimedia applications requires that peripheral processors and the communication subsystem be scheduled together.

it.

The Swift architecture provides the distinct advantage that the application can choose its reliability level. Since data transfer is segment based, each transfer plan can specify a required reliability. For a given reliability level and performance constraint, an appropriate group size can be selected based on available resources.

3 Two Validation Studies

We have done two studies to validate the Swift architecture. The first was a simulation study of a possible local-area network implementation of Swift. The second was a proof-of-concept Ethernet-based prototype of a simplified version of the architecture. The complete set of results can be found elsewhere [10, 11].

3.1 LAN Simulation of Swift

The simulator was used to locate the components that were the limiting factors for a given level of performance. The simulator did not model caching, computing data parity blocks, any preallocation of resources, nor did it attempt to provide performance guarantees. Traces of file system activity would have been required in order to model these effectively and such traces were unavailable to us. In addition, the simulator did not model the storage mediator as it is not in the path of the data transmitted to and from clients, but is consulted only at the start of an I/O session.

In our simulation of Swift, to **read**, a small request packet is multicast to the storage agents. The client then waits for the data to be transmitted by the storage agents. A **write** request transmits the data to each of the storage agents. Once the blocks have been transmitted the client awaits an acknowledgement from the storage agents that the data have been written to disk.

Our model of the disk access time is conservative in that the seek time and rotational latency are assumed to be independent uniform random variables, and no attempt was made to order requests to schedule the disk arm, pessimistic assumptions when advanced layout policies are used [12].

The data transfer processing costs were taken into account by assuming that protocol processing required 1500 instructions [13] plus 1 instruction per byte in the packet.

The load that could be carried depended both on the number of disks used and the block size. The delay was dominated by the disk, with an average seek time of 16 milliseconds, an average rotational delay of 8.3 milliseconds and a transfer rate of 2.5 megabytes per second. The result was that transferring 32 kilobytes required about 37 milliseconds on the average. As the block size was increased, seek time and rotational delay were mitigated and the transfer time became more dependent on the amount of data transferred.

As small transfer sizes require many seeks in order to transfer the data, large transfer sizes have a significantly positive effect on the data-rates achieved. For small numbers of disks, seek time dominated to the extent that its effect on performance was almost as significant as the number of disks.

When 4 disks were used, the system saturated quickly. For larger numbers of disks, the response time was almost constant until the knee in the performance curve was reached. For 32 disks, the maximum sustainable load was reached at about 22 requests per second. At this point the disks were 50% utilized on the average. The rate of requests that are serviceable increased almost linearly in the number of disks. Increased rotational delay and a slight loading of the communication medium prevents it from being strictly linear.

The maximum sustainable data-rate is that which is observed by the client when the average time to complete a request is equal to the average time between requests. For transfer units of 4 kilobytes, the maximum sustainable data-rate for 32 disks is approximately 2 megabytes per second. When transfer units of 32 kilobytes are used, the maximum sustainable data-rate increases to nearly 12 megabytes per second for the same 32 disks. The increase in effective data-rate is almost linear in the size of the transfer unit.

The clear conclusion is that when sufficient interconnection capacity is available the data-rate is almost linearly related to both the number of storage agents and to the size of the transfer unit. The reason the transfer unit impacts so much the data-rates achieved by the system is that seek time and latency at the disks are enormous when compared to the speed of the processors and the network transfer rate. This also shows the value of careful data placement and indicates that resource preallocation may be very beneficial to performance.

3.2 LAN Implementation of Swift

In a simplified Ethernet-based prototype of Swift we found that its performance was limited by the speed of the Ethernet. The prototype provided network data-rates that were between two and three times more detail.

2.1 Distribution Agent

The distribution agent acts on behalf of its clients, the data producer and the data consumer, in the storage and retrieval of all data. Although not required, we expect that in practice both the data producer and the data consumer be co-resident in the same host as the distribution agent.

The distribution agent interacts with the storage mediator to obtain directory service, access rights to objects, encryption keys, and transfer plans. In addition, all computed transformations of the data, such as encryption and erasure correcting codes, are done by the distribution agents. Authentication is accomplished through a secure exchange of keys with the storage mediator to obtain a trusted channel.

The primary task of the distribution agent is to implement distributed striping of the data over several storage agents. When reading, it assembles the object from the incoming data streams according the the transfer plan. When writing, it distributes the data object among the several storage agents. In both cases it performs any parity computations necessary to provide fault tolerance.

2.2 Storage Mediator

The storage mediator is central to establishing and administering resources. It negotiates with the storage agents to reserve sufficient space and transfer capacity. It also determines how to best meet the resiliency requirements and returns this as part of the transfer plan.

The transfer plan contains the list of segments making up the object, the transfer unit for each segment, the transfer unit for each storage agent, a list of storage agents to hold the data, a list of storage agents to act as checks on the data.

Encryption is the mechanism used to provide authentication, access control, and security of the data. The storage mediator is the sole repository for encryption keys. It will use a secure key exchange protocol to authenticate the distribution agents.

Since the Swift architecture is based on preallocation, it easily provides sequential write sharing, namely the ability for two clients to have alternate access to the same data. The storage mediator will use a call-back mechanism to provide cache coherency. When a distribution agent requests access to an object which still may exist in the cache of some other client, the storage mediator will cause that cache to be flushed as part of the resource allocation protocol.

In order to achieve high performance, a pessimistic storage allocation strategy is used. Since all resources are preallocated, requests that would exceed current storage capacity will be denied. Similarly, requests that would exceed the current transfer capacity will be denied. These requests can be reissued at a later time when more resources are available.

The storage mediator must be highly available and the metadata it maintains be highly fault tolerant. For example, each directory entry contains the name of the object, its protection status, a list of data segments and storage agents that hold the object. A hot stand-by approach can be used to ensure that the storage mediator will be able to provide services. Load sharing among the copies of the storage mediator can improve performance of the system. The integrity of the storage mediator's data can be insured in several ways. One method is to let Swift administer the metadata specifying a high degree of resiliency. Another would be to use standard data base techniques [6].

2.3 Storage Agents

The storage agents administer all aspects of secondary storage media, including data layout optimization and off-line data alignment. Each storage agent may administer many storage devices that can be disks, or other high speed devices including disk arrays.

Since the Swift architecture is intended for objects much larger than any cache, we believe that caches will be used most often for staging data into transfer units than for storing complete objects. For small objects we expect caches to be as beneficial as in other systems [7].

Object descriptors store redundant information. This allows the reconstruction of all objects by scavenging the data in the storage agents, should a catastrophic failure, or a software error, render the storage mediator inoperative.

For any long-term storage system, reliability is an important concern. In an architecture that uses disk striping [8], the increased number of devices increases the probability that some will be inoperative [9]. Through the appropriate use of redundancy, the reliability of the system can be enhanced to any desired level.

The solution we have chosen for Swift is to use redundant storage for erasure correcting codes [4]. By using the error detecting capabilities of the disks, a single parity disk is sufficient to tolerate a single failure [9, 4]. In this way, if a disk fails then the information on the other disks can be used to reconstruct

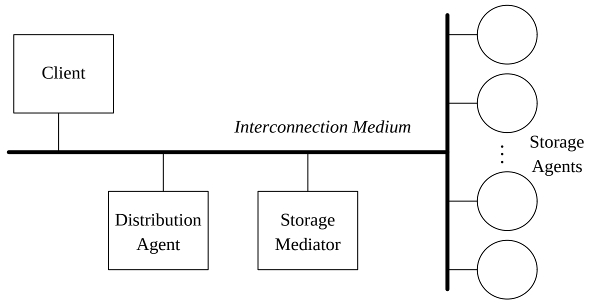

Figure 1: Components of the Swift Architecture

work stations if data redundancy is not used.

We also present the summary of an Ethernet-based proof-of-concept local-area prototype of Swift. This implementation demonstrated the validity of disk striping over a set of servers placed on the network. Our prototype achieved data rates higher than writing to the local disk and also higher than those obtained by the NFS file service.

2 The Swift Architecture

The only way to address the disparity between the transfer rate of disk devices and the higher data-rates mandated by new applications is to use several disks in parallel. Much like memory interleaving, faster secondary storage systems can be built from a collection of slower storage devices. Several concurrent I/O architectures, such as Imprimis ArrayMaster [1], DataVault [2], CFS [3] and Raid [4, 5], are based on this observation. Swift uses this approach to achieve any required data-rate to secondary storage up to saturation of the interconnection medium. Figure 1 has a diagram of the architectural structure of Swift.

The Swift architecture distinguishes four logical components: *distribution agent*, *data producer*, *storage mediator*, and *storage agents*.

A *distribution agent* operates in close cooperation with the *data producer*. It is responsible for assembling the data streams coming in from the multiple data repositories (in the case of reading), and distributing the data to be striped over the several data repositories (in the case of writing). Encryption and decryption are also the responsibility of the distribution agent.

The *storage mediator* is central to Swift. It operates in close cooperation with the *storage agents* and is responsible for providing directory services, authentication, enforcing access control, producing transfer plans, administering storage allocation, cache coherency, and ensuring serializability of concurrent activities.

The *storage agents* drive the storage devices at the data repositories and are responsible for storing and retrieving data at negotiated rates. They are also responsible for keeping enough information to allow the reconstruction of objects from its constituent parts should the storage mediator fail.

Swift assumes that an interconnection medium with sufficient capacity will be available for the applications. This assumption is reasonable since high-speed communications networks that operate in excess of 1 gigabit per second are being developed.

When creating a directory that will store Swift objects, its name and protection status, as well as the degree of redundancy of all objects stored in it, are specified. The degree of resiliency of the directory itself is also specified to the storage mediator.

In order to store an object in Swift, the data producer contacts the distribution agent with the name, estimated size, estimated maximum data-rate, and resiliency requirements of the object to store. The distribution agent in turn contacts the storage mediator with these requirements. The storage mediator determines the degree of redundancy required for the object. It then determines which of the storage agents will service this request and how the object will be striped across them. The storage mediator requests internal object repository handles to each of the storage agents. Upon completion of this request each storage agent has reserved the necessary resources in terms of storage and transfer capacity for it to store its assigned part of the object. The storage mediator collects all the handles from the storage agents, returning the collection of internal object repository handles to the distribution agent as part of the transfer plan for the object.

Once the storage plan is created, the distribution agent does *not* use the storage mediator as an intermediary; it sends data directly to the storage agents. The distribution agent returns a handle for the object to the data producer. This handle, which is internally generated by the distribution agent, is used with each request to transfer data. The distribution agent routes the (possibly modified) data to the appropriate storage agents. This scenario assumes that the data producer knows how to contact a distribution agent.

The scenario for retrieving data is analogous, with the only difference being that the transfer plan tells the distribution agent from which storage agents the data must be fetched. The final collation and presentation to the client is done by the distribution agent.

In the following subsections the distribution agent, storage mediator, and storage agent are presented in

Swift: A Storage Architecture for Large Objects

Luis-Felipe Cabrera
Computer Science Department
IBM Almaden Research Center

Internet: cabrera@ibm.com

Darrell D. E. Long
Computer & Information Sciences
University of California at Santa Cruz

Internet: darrell@sequoia.ucsc.edu

Abstract

Managing large objects with high data-rate requirements is difficult for current computing systems. We describe an Input/Output architecture, called Swift, that addresses the problem of storing and retrieving very large data objects from slow secondary storage at very high data-rates. Applications that require this capability are poorly supported in current systems, even though they are made possible by high-speed networks. These range from storage and visualization of scientific computations to recodring and play-back of color video in real-time. Swift addresses the problem of providing the data rates required by digital video by exploiting the available interconnection capacity and by using several slower storage devices in parallel.

We have done two studies to validate the Swift architecture: a simulation study and an Ethernet-based proof-of-concept implementation. Both studies indicate that the aggregation principle proposed in Swift can yield very high data-rates. We present a brief summary of these studies.

1 Introduction

The disparity between processing speed, network transfer rates, and the performance of disk storage systems will increase in the future. The processing speed of computing systems continues to increase at an exponential rate. Advances in communications technology are providing increased transfer rates even more rapidly than the increases in processing speed.

In contrast to these advances, disk storage technology remains much the same. Although the density of the media has greatly increased, there has been little improvement in either access times or data transfer rates. In the case of optical storage, the access times have increased and the data transfer rates have decreased relative to magnetic media. Due to physical considerations, substantial increases in disk storage data transfer rates seem unlikely.

Because of increased processing power and the potential for high network transfer rates, new applications are emerging. These applications range from bulk data transfer for super computers to managing digital color video in real-time. Today, managing digitized color video in real-time is impossible. Storing just a few minutes of digitized color video requires gigabytes of storage. Storing or retrieving it in real-time requires sustained transfer rates on the order of 20 megabytes per second.

Our architecture, called Swift, addresses the problem of storing and retrieving large data objects from slow secondary storage at very high data-rates. Swift is based on the premises that: (1) the network interconnection will be capable of supporting much higher data-rates than individual storage agents; (2) resources can be preallocated for storing and transmitting data; (3) multiple storage agents can be driven concurrently using data striping; and (4) failures of storage agents can be masked using data redundancy.

Swift is based on a client-server model and addresses the issues of authentication, access control, and encryption. Since it is a distributed architecture made up of independently replaceable components, it can provide very high reliability. It is adaptable to different network interconnection topologies and technologies. Swift operates by having a storage mediator reserve resources from storage agents in a session-oriented manner, and then presenting a distribution agent with a *transfer plan*. The distribution agent stores or retrieves the data at the storage agents following that plan.

Even though Swift was designed with very large objects in mind, it can handle small objects such as those encountered in normal file systems with two penalties: one round trip time for a short network message to consult the storage mediator, and computing the required data redundancy. Swift is also well suited as a swapping device for high performance