Highly-Available, Scalable Network Storage

Edward K. Lee
Digital Equipment Corporation
Systems Research Center
eklee@src.dec.com

Abstract

The ideal storage system is always available, is incrementally expandable, scales in performance as new components are added, and requires no management. Existing storage systems are far from this ideal. The recent introduction of low-cost, scalable, high-performance networks allows us to re-examine the way we build storage systems and to investigate storage architectures that bring us closer to the ideal storage system. This document examines some of the issues and ideas in building such storage systems and describes our first scalable storage prototype.

1 Motivation

Today, managing large-scale storage systems is an expensive, complicated process. Various estimates suggest that for every $1 of storage, $5-$10 is spent to manage it. Adding a new storage device frequently requires a dozen or more distinct steps, many of which require reasoning about the system as a whole and are, therefore, difficult to automate. Moreover, the capacity and performance of each device in the system must be periodically monitored and balanced to reduce fragmentation and eliminate hot spots. This usually requires manually moving, partitioning, or replicating files and directories. For example, directories containing commonly used commands are often replicated to distribute load across several disks and file servers. Also, user directories are frequently partitioned across several file systems, resulting in naming artifacts such as /user1, /user2, etc.

Another important contributor to the high cost of managing storage systems is component failures. In most storage systems today, the failure of even a single component can make data inaccessible. Moreover, such failures can often bring an entire system to halt until the failed component is repaired or replaced and the lost data restored. As computing environments become more distributed, the adverse effects of component failures become more widespread and frequent. Aside from the cost of maintaining the necessary staff and equipment, such failures could incur significant opportunity costs.

Products such as redundant disk arrays and logical volume managers often simplify the management of centralized storage systems by automatically balancing capacity and performance across disks, and by tolerating and automatically recovering from some component failures [4]. Most such products, however, do not support large-scale distributed environments. They cannot balance capacity or performance across multiple server nodes and cannot tolerate server, network, or site failures. They are effective in managing storage local to a given server but once your system grows beyond the limits of a single server, you must face all the old management problems anew at a higher level. Moreover, because distributed systems, unlike centralized systems, can suffer from communication failures, the difficulty of the problems is significantly increased.

Of course, products to manage storage in distributed environments are also available. They collect information from throughout the system, summarize the information in an easy to understand format, and provide standard interfaces for configuring storage components. We are unaware of any, however, that provides the level of automation and integration offered by the best centralized storage management products. In this paper, we propose properties that are desirable of block-level distributed storage systems and ideas for implementing such storage systems. We describe the implementation of our first distributed storage prototype based on these ideas and conclude with a summary and directions for future work.

2 An Architecture for Scalable Storage

The desirable properties of a distributed storage architecture are straight-forward. First, the system should be always available. Users should never be denied authorized access to data. Second, the system should be incrementally expandable, and both the capacity and throughput of the system should scale linearly as additional components are added. When components fail, the performance of the system should degrade only by the fraction of the failed components. Finally, even as the storage system's size in-

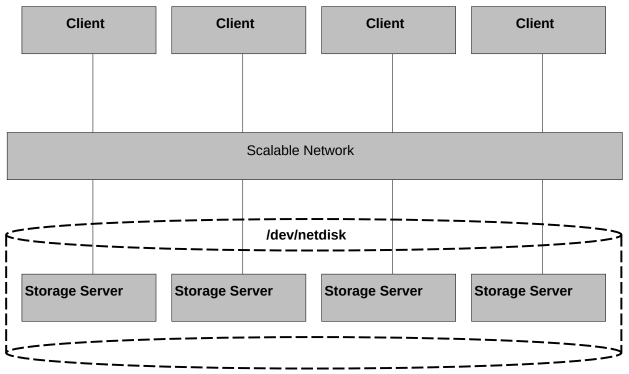

Figure 1: Architecture for Scalable Storage.

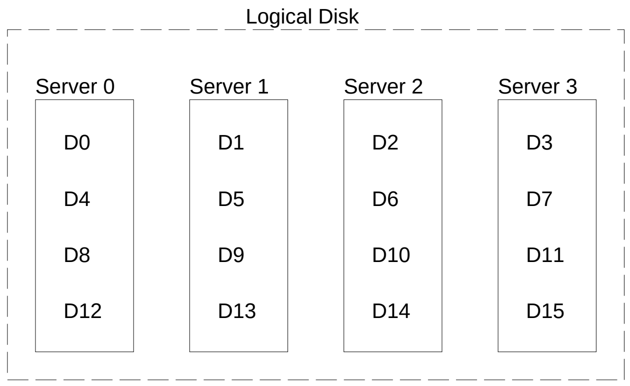

Figure 2: Round-Robin Data Striping.

creases, the overhead in managing the system should remain fixed.

Such a *scalable* storage architecture would allow us to build storage systems at any desired level of capacity or performance by simply putting together enough scalable storage components. Moreover, a large storage system would be no more difficult to manage than a small storage system. We could start with a small system and incrementally increase its capacity and performance as the needs of our system grew.

But how do we design such scalable storage systems? The key technological innovation that makes scalable storage systems feasible is commodity scalable (switch-based) networks and interconnects. Without scalable interconnects, the size of a computer system is limited to a single box or at most a few boxes each containing a few processing, storage and communication elements. Only so many CPU's, disks and network interfaces could be aggregated before the system interconnect, usually a bus, saturates. In contrast, scalable interconnects can support almost arbitrary levels of performance, allowing us to build systems with an arbitrary number of processing, storage and communication elements. Furthermore, we could group many elements together into specialized subsystems that provide superior performance but can be accessed and managed as if they were a single element.

Figure 1 illustrates these concepts when applied to the storage subsystem. Each scalable storage server consists of a processing element, some disks, and a network interface. In the ideal case, we plug the components into the scalable interconnect, turn them on, and the components auto-configure, auto-manage and communicate with each other to implement a single large, highly-available, high-performance storage system. In particular, the servers automatically balance capacity and performance across the entire storage system and uses redundancy to automatically tolerate and recover from component failures. All the intelligence needed for implementing the storage system is contained within each individual scalable storage server; the storage servers do not require help from the clients or a centralized command center. To the clients, the storage servers collectively appear as a single large highly-available high-performance disk with multiple network interfaces. When an additional storage server is added, the storage systems simply looks like a larger, higher-performance disk with more network connections.

To conclude, all systems, whether they are biological, social, or computational, are limited in size and effectiveness by the capabilities of their basic control and transportation infrastructure. The advent of commodity scalable networks and interconnects radically alter the basic assumptions we use for building computing systems. In particular, we believe that they will dramatically alter the way we build and use distributed storage systems.

3 Availability and Scalable Performance

The main technical challenges in designing scalable storage systems are availability and scalable performance. We have found that in large distributed systems, scalable performance is closely tied with good load-balancing. Furthermore, because almost all schemes for providing high-availability impose a performance penalty during the normal operation of the system and also because failures should not cause disproportionate degradations in performance, the availability and performance issues are closely related. This section discusses the basic mechanisms for meeting these challenges: data striping and redundancy.

Figure 2 illustrates round-robin data striping. The solid rectangles represent blocks of storage on the specified storage server, the dotted rectangle emphasizes that the storage

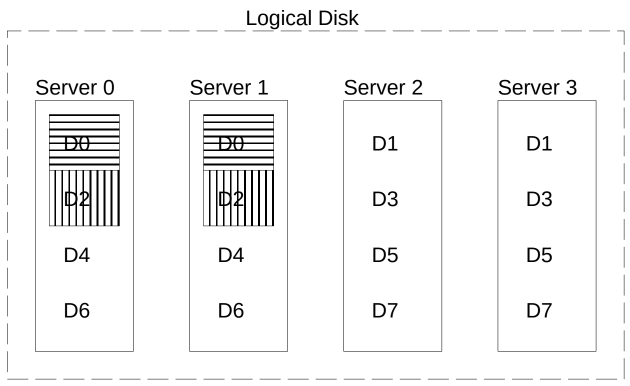

Figure 3: Mirrored-Striping.

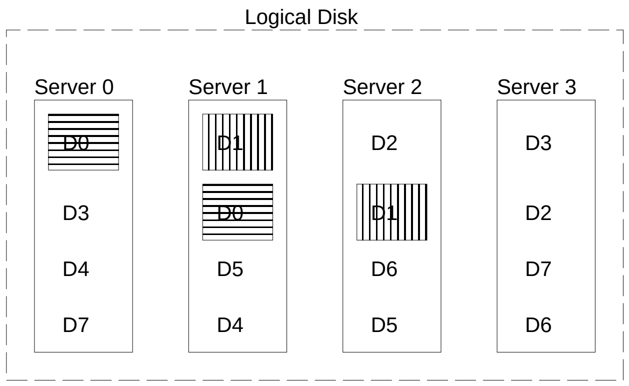

Figure 4: Chained-Declustering.

servers appear as a single logical disk to its clients. Each letter represents a block of data stored in the storage system. The figure shows that the first block of data, *D0*, is stored on storage server 0, the second block of data, *D1*, is stored on storage server 1, and so on. This effectively implements a RAID 0 [4], using storage servers instead of disks. The main problem with this arrangement is the availability of the system. Any single disk failure will result in data loss and any single server failure will result in data unavailability.

In theory, you can implement a distributed RAID 5 using storage servers [2]. However, in practice, the complexity of the additional synchronization required for RAID 5 systems in comparison to a replication-based redundancy scheme is greatly amplified in a distributed environment. Also, RAID 5 has the undesirable property that the failure of a single storage server increases the average read load by a factor of two. This makes RAID 5 unacceptable for many performance-critical applications such as video-on-demand.

Figure 3 illustrates the *mirrored-striping* data placement scheme. Data is striped round-robin across a set of mirrored servers. This provides good reliability by protecting the system from both disk and server failures. However, should Server 0 fail, Server 1 must assume all of Server 0's load while servers 2 and 3 experience no load increase. Clearly, it would be better if all surviving servers experienced a 33 % increase in the read load rather than having a single server experience a 100 % increase in load.

Figure 4 illustrates the *chained-declustered* data placement scheme [7]. Note that Server 0's copy of *D0* is on Server 1, Server 1's copy of *D1* is on Server 2, and so on. Now, if Server 1 fails, servers 0 and 2 share Server 1's read load but Server 3 experiences no load increase. This is not perfect but much better than having a single server bear all of Server 1's load. By performing dynamic load balancing, we can do much better. For example, since Server 3 has copies of some data from servers 0 and 2, servers 0 and 2 can offload some of their normal read load on Server 3 and achieve uniform load balancing.

Although we have presented this example with four servers, the same type of offloading can be done with many more servers. Chaining the data placement allows each server to offload some of its read load to either the server immediately following or preceding the given server. By cascading the offloading across multiple servers, a uniform load can be maintained across all surviving servers. A disadvantage of this scheme is that it is less reliable than mirrored-striping. With mirrored-striping, if Server 1 failed, only the failure of Server 0 would result in data unavailability. With chained-declustering, if Server 1 fails, the failure of either Server 0 or Server 1 will result in data unavailability.

Figure 5 illustrates a variation on the chained-declustered data placement called *multi-chained-declustering*. Instead of having a single chain of stride one, multiple chains of varying strides are used. The system illustrated in Figure 5 uses chains of both stride one and stride two. If dynamic load balancing is not used, this scheme provides better load balancing than chained-declustering. For example, if Server 2 fails, servers 0, 1, 3, 4 each experience a 25 % increase in the read load compared to 50 % with chained-declustering. In large configurations, multi-chained-declustering has an additional benefit over chained-declustering. With chained-declustering, multiple failures can prevent uniform load balancing by breaking the chain in more than one place. This effect is most pronounced if the two failures occur close together in the chain. With multi-chained declustering, this is not problem since the chain of stride two can be used to skip over the failed servers. As expected, however, multi-chained-declustering is less reliable than chained-declustering. If,

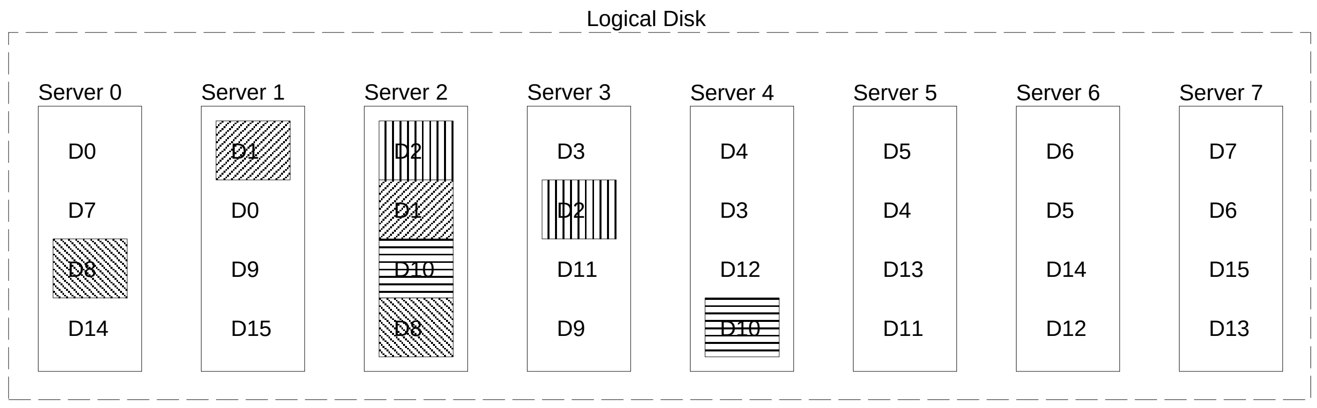

Figure 5: Multi-Chained-Declustering.

for example, Server 2 fails the failure of any one of servers 0, 1, 3, and 4 results in data unavailability. As this example illustrates, in the absence of dynamic load balancing, a tradeoff exists between load balancing in the face of failures and the availability of the system.

4 Virtual Volumes

The primary practical, in contrast to technical, problem in designing scalable storage systems is making them easy to manage. Designing systems to automatically tolerate and recover from component failures and to automatic balance load is an important part of the solution, but it is not sufficient. Although we have described how to build a single large block-level device with high-availability and scalable performance, we have not provided any mechanism for sharing this device in heterogeneous distributed environments or for making consistent backups and restores of such large devices. For the scale of storage systems we are considering, the standard UNIX mechanism of statically partitioning a device into several fixed-size devices is inadequate. Our solution to the problem applies a well known technique for sharing resources in computer systems; we virtualize the resource.

The primary storage management abstraction we support is the *virtual volume*. A virtual volume looks like a large block-level device but unlike a real device, it can be created and destroyed as needed by clients of the scalable storage system. Each virtual volume gives each client access to the performance capabilities of the entire scalable storage system. Similar to the virtual address space abstraction supported by many operating systems, the creation of a virtual volume only allocates a range of virtual disk addresses and does not in itself allocate any physical storage.

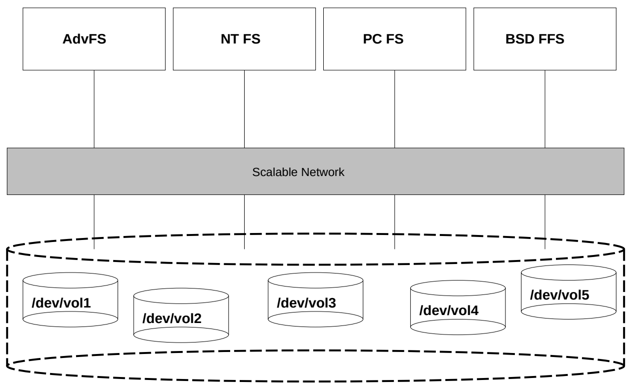

Figure 6: Virtual Volumes.

It is only when locations in the virtual volume are used that physical storage is allocated and mapped to an address in the virtual volume. Thus, a virtual volume need not have a definite size. This can be an advantage for advanced file systems that can dynamically grow in size. For some applications, it may be desirable to guarantee each virtual volume some minimum amount of physical storage. Likewise, it is probably desirable to place maximum size restrictions on volumes to prevent it from using up all the physical storage. Figure 6 illustrates how virtual volumes might fit into heterogeneous distributed environments.

Virtual volumes are implemented by explicitly mapping each virtual volume identifier and offset to a corresponding physical disk address. The existence of this map between virtual and physical addresses allows us to play the same type of mapping games found in virtual memory systems. For example, by adding a facility for making virtual volumes copy-on-write, we can support an efficient mechanism for creating instantaneous snapshots of volumes. Such a snapshot facility is needed to make consistent file system dumps without making file systems unavailable for long periods of time. Periodic snapshots can also be used to recover accidentally deleted files and directories without resorting to tape backups. Finally, read-only snapshots of volumes can be mounted as read-only file systems on several file servers simultaneously, allowing commonly accessed files such as executables to be servered from multiple machines without having to partition or replicate the files.

5 Our First Prototype

Over the past year, we have applied the ideas described in this paper to construct our first scalable storage prototype. The primary goal of the prototype is to study the high-availability and scalable performance characteristics of the scalable storage system. Currently, the prototype survives any single component failure including disk, server, and network failures. Appropriately partitioning the servers over two or more sites should allow the system to survive site failures, but this has not been tested. The prototype currently does not support virtual volumes. The prototypes is implemented using Alpha/OSF workstations and the AN1 [9] (similar to switched FDDI) network. Due to the current lack of spare disks in our computing environment, we have thus far simulated the disks.

Our first prototype uses the chained declustered data placement described previously. One nice property of this data placement is that, assuming an even number of servers, placing the even and odd numbered servers at different sites results in a system that can survive any single site failure. A system using the multi-chained declustered data placement described in this paper can also be partitioned to survive site failures, however, this placement requires a minimum of three rather than two sites.

We use the fail-stop failure model in designing the prototype's redundancy mechanisms. That is, we assume that when a component fails, it ceases to operate. In particular, a failed component does not continue to operate incorrectly. A distributed voting mechanism based on timeouts ensures that all servers can agree on the failed or operational status of each other in the face of communication failures.

The software components of the scalable storage prototype can be broken into roughly two basic components: the server software that runs on each scalable storage server, and the client software that runs on each client of the scalable storage system. The client software looks just like a standard block-level device driver to its host operating system. From the clients' perspective, adding a scalable storage system is similar to adding a new disk controller. Once the scalable storage driver is installed, a client can access the scalable storage system just as if it were a locally attached disks.

A basic design principle in implementing the server and client software was to implement all the necessary functionality in the server software and to maintain only "hints" in the client software. This makes it unnecessary to implement complicated consistency protocols to constantly update information cached at clients, and ensures that a client cannot compromise the logical consistency of data stored in the scalable storage system. The client currently maintains only a small amount of high-level mapping information that is used to route read and write requests to the most appropriate server. If the request is sent to the wrong server, the server updates the information at the client and the client retries with the new information.

A primary example of this design philosophy, is that data redundancy is implemented completely within the servers; the clients are completely unaware of the redundancy scheme that is used or, indeed, that any redundancy scheme is being used. This results in a particularly simple network interface that makes it easier to experiment with different redundancy schemes and makes it impossible for the clients to compromise the data redundancy mechanisms. This also means that the clients are not burdened with the additional work required for supporting data redundancy. For example, when a write request is generated by a client, the appropriate scalable storage server is responsible for generating any additional network requests needed for updating the corresponding replica.

Preliminary performance measurements of our untuned first prototype show that each server can service 1100 512-byte read requests per second and 400 512-byte write requests per second. For 8 KB requests, each server can service read requests at 4.8 MB/s and write requests at 1.6 MB/s. The reader may have noticed that the write performance for both small and 8 KB requests is approximately one-third the read performance. This is because for a read request, the storage system need only read the requested data from, the simulated, disk and send it out over the network, whereas for a write request, the storage system must receive the data, send it to the corresponding replica server and the replica server must subsequently receive and process the data. Thus, a write request generates three times the network processing overhead generated by a read request. As expected, preliminary performance analysis of the system indicates that optimization of the networking code should result in significant performance improvements. The reader should bear in mind that the above describes the performance that is achieved by each server in the scalable storage system. We expect the performance of the system to scale linearly up to a hundred or more servers. The largest scalable storage system we have tested to date is with four servers.

6 Summary, Conclusions and Future Work

The ideal storage system is always available, incrementally expandable, scales in performance as new components are added, and requires very little management. When additional capacity or performance is needed, a new storage server is plugged into the network and turned on. The addition storage is then immediately and transparently accessible to all clients of the storage system. In particular, the capacity and load is automatically distributed uniformly across all storage servers in the system.

In this paper, we have presented ideas for building storage systems that come closer to this ideal. We've outlined some of the data striping and redundancy schemes that are suitable for distributed storage systems and briefly mentioned why RAID level 5 systems, while good for centralized storage systems, may not generalize well to distributed storage systems. We've also motivated the usefulness of virtual volumes in managing and allocating storage in distributed environments and some of the additional functionality such as snapshots that they can efficiently support. Finally, we have briefly described our first scalable storage prototype and tried to give an idea for the level of performance that can be expected.

Over the next year or two, we plan to finish our implementation of virtual volumes, include support for adding disks and servers on-line without making the storage system unavailable, build a larger prototype using real disks, and look for interesting applications for the system. One class of applications that currently appeals to us is providing a wide-area information service over the Internet.

In conclusion, we believe that large-scale distributed storage systems represents a very important opportunity for both research and product development. The availability, performance and manageability of existing storage systems is inadequate for building the large-scale information systems that are envisioned for the future. If we truly wish to build the massive video-on-demand, regional medical information systems, enterprise-wide information systems, and consumer-oriented information systems that the recent advances in telecommunications make possible, we must reexamine the basic building blocks we use in building large distributed systems. Irregular stones may be fine for building cottages but we need modular bricks to build cathedrals. (Yes, it's not a very pretty analogy but I think that it makes a point.)

References

[1] Luis-Felipe Cabrera and Darrel D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *ACM Computing Systems*, 4:405–436, Fall 1991.

[2] Pei Cao, Swee Boon Lim, Shivakumar Venkataraman, and John Wilkes. The TickerTAIP parallel RAID architecture. In *Proc. International Symposium on Computer Architecture*, pages 52–63, May 1993.

[3] Peter M. Chen, Edward K. Lee, Ann L. Drapeau, Ken Lutz, Ethan L. Miller, Srinivasan Seshan, Ken Shirriff, David A. Patterson, and Randy H. Katz. Performance and design evaluation of the RAID-II storage server. *Journal of Distributed and Parallel Databases*, 2(3):243–260, July 1994.

[4] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[5] Joy Foglesong, George Richmond, Loellyn Cassel, Carole Hogan, John Kordas, and Michael Nemanic. The Livermore distributed storage: Implementation and experiences. Technical Report UCRL-102663, Lawrence Livermore National Laboratory, May 1990.

[6] John H. Hartman and John K. Ousterhout. Zebra: A striped network file system. In *Proc. USENIX File Systems Workshop*, pages 71–78, May 1992.

[7] Hui-I Hsiao and David J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machines. Technical Report CS TR 854, University of Wisconsin, Madison, June 1989.

[8] A. L. Narasimha Reddy and Prithviraj Banerjee. Gracefully degradable disk arrays. *Proc. International Symposium on Fault-Tolerant Computing*, June 1991.

[9] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. Technical Report SRC-59, Digital Equipment Corporation, Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301-1044, April 1990.