A Scalable Architecture for Clustered Network Attached Storage

Jonathan D. Bright
Sigma Storage Corporation
jon@brightconsulting.com

John A. Chandy
University of Connecticut
john.chandy@uconn.edu

Abstract

Network attached storage systems must provide highly available access to data while maintaining high performance, easy management, and maximum scalability. In this paper, we describe a clustered storage system that was designed with these goals in mind. The system provides a unified file system image across multiple nodes, which allows for simplified management of the system. Availability is preserved with multiple nodes and parity-striped data across these nodes. This architecture provides two key contributions: the ability to use low-cost components to deliver scalable performance and the flexibility to specify redundancy and performance policy management on a file-by-file basis. The file system is also tightly integrated with standard distributed file system protocols thereby allowing it to be used in existing networks without modifying clients.

1. Background

The traditional storage solution has typically been direct attached storage (DAS) where the actual disk hardware is directly connected to the application server through high-speed channels such as SCSI or IDE. With the proliferation of local area networks, the use of network file servers has increased, leading to the development of several distributed file systems that make the local server DAS file system visible to other machines on the network. These include AFS/Coda [17, 25], NFS [23], Sprite [20], CIFS [14], amongst others. The desire to increase the performance and simplify the administration of these file servers has led to the development of dedicated machines known as network-attached storage (NAS) appliances by companies such as Network Appliance, Auspex, and EMC. In addition to specialized file systems [12], these NAS appliances are also characterized by specialized hardware components to address scalability and reliability [3].

In an effort to remove the bottleneck of the single server model of NAS servers, there has lately been significant work in the area of distributed or clustered storage systems. These include distributing data amongst dedicated storage nodes as with storage area networks (SANs), virtual disks [15] and network-attached secure disks (NASD) [10], or distributing the data amongst the clients themselves in so-called serverless storage systems [1, 11]. The migration to these systems has been driven by the need to increase concurrent access to shared data. However, all these architectures require new client-to-storage transfer protocols meaning that client software must be modified and standard distributed file systems such as NFS or CIFS are not supported. In addition, some of the architectures require specialized and typically expensive hardware to implement the required functionality.

Another significant issue with NAS systems is their reliability and the most failure prone component of NAS systems is their disk subsystem. The most common and cost effective solution to improve the availability of disk systems is the use of Redundant Array of Independent Disks (RAID) [21]. A RAID system stripes data across multiple hard disks that appear to the user as a single disk. The various levels of RAID specify different methods of redundancy, such as parity and mirroring, to provide reliability. The most commonly used forms of RAID are RAID-1 for mirroring and RAID-5 for parity-rotated striping.

Although RAID improves the reliability compared to single disk data storage systems, NAS systems with RAID still have other significant limitations. For example, the disk arrays are generally embodied in a single NAS server, and are therefore susceptible to machine level failures (e.g., power failure, network connection failure, etc.). Additionally, it is difficult to incrementally increase the storage capacity of a NAS server, because an additional single disk cannot generally be added to a RAID system. Further, NAS systems are typically connected to a network via a limited set of network connections, thereby limiting the data transfer bandwidth to/from the server. Additionally, single machine systems have practical limits on the number of processing units that can be implemented (e.g., to run server processes, parity calculations, etc.), thereby limiting the number of clients that can be effectively served.

In addition to providing potential scalability gains, clus-

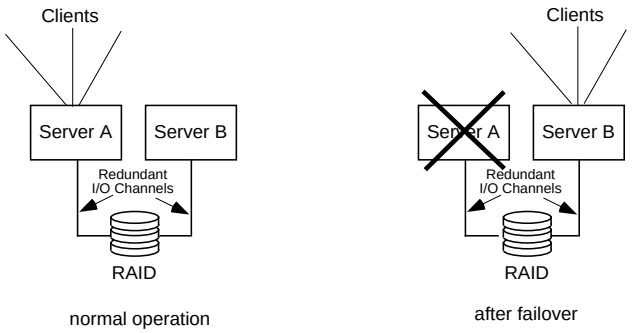


Figure 1. NAS with Failover.

tered storage can also prove to be a solution to the machine level failure problem. In a simple configuration, two servers are connected to a common RAID array through redundant I/O channels, with only one server actively serving clients (Figure 1). If the server stops operating, the system "fails over" to the second server which will then resume serving clients. However, the system requires specialized hardware to handle failover seamlessly, and the cost paid for an extra server does not buy extra throughput. In addition, the disk array subsystem is still a potential single point of failure that must be addressed again with expensive hardware by providing redundant components – controllers, power supplies, fans, etc.

A higher-end clustering solution involves using multiple servers serving clients simultaneously and sharing a pool of SAN block storage devices connected by a high speed connection fabric (Figure 2). The block storage devices may be FibreChannel disks directly connected to the interconnect or intelligent servers servicing block requests through FibreChannel or emerging IP protocols such as iSCSI [24]. Specialized file systems must be used to present a unified and consistent view of the file system to clients and also manage the SAN storage pool from the clustered servers. The multiple servers can provide scalable growth for clients unlike the failover solution. SAN storage backends, however, are very expensive and typically difficult to manage.

It is possible to create a cluster where each server has local storage, thereby eliminating the need for a dedicated storage network and specialized block storage devices. Such an architecture allows for the use of standard servers without any specialized hardware. However, it also necessitates specialized software to aggregate the storage on the multiple nodes into a unified file system.

In this paper, we describe a architecture with local storage called the Sigma Cluster Storage Architecture that addresses some of the shortcomings of existing distributed storage systems. In particular, the system delivers the scalability of a clustered storage system while remaining compatible with existing distributed file systems, and the system uses no specialized hardware to realize the functionality. The other distinguishing contribution of the system is the ability to make redundancy and striping decisions on a file-by-file basis.

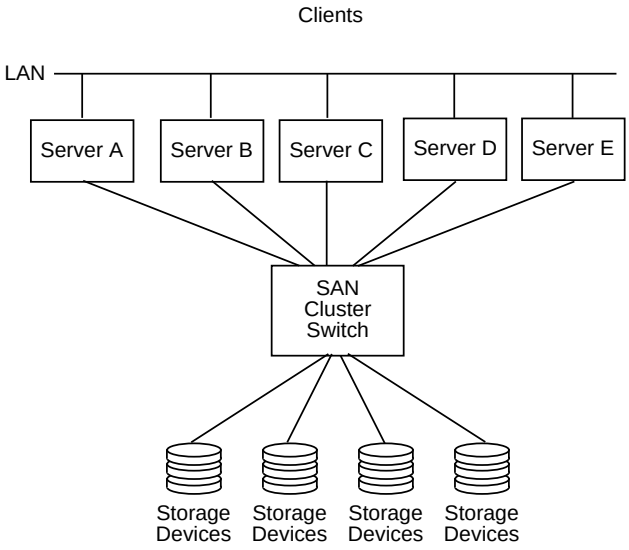


Figure 2. Clustered NAS using a SAN.

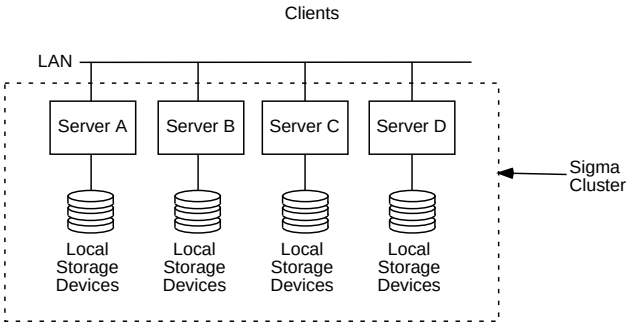


Figure 3. Clustered NAS.

2. Sigma Cluster File System

2.1. Overview

The Sigma Cluster Storage Architecture is an example of a clustered NAS architecture. The physical layout is shown in Figure 3. As with a NAS, clients can connect to the Sigma cluster using a distributed file system protocol such as NFS or CIFS. However, unlike a traditional NAS, the client can connect to any of the nodes in the cluster and still see the same unified file system. The multiple nodes in the cluster also allow the Sigma system to eliminate the single-server bottleneck of a NAS system. NFS or CIFS data requests are translated into requests to the Sigma cluster file system, which is distributed across the nodes of the cluster. The file system is responsible for file management

as well as data distribution, i.e. striping data across nodes using varying redundancy policies.

Though, the physical layout of the Sigma system is similar to the backend of a SAN layout, the difference is apparent at higher levels. The data transfer protocol between clients and the Sigma storage system is at the file level while with SANs, the data transfer protocol is at the block level. The implication, of course, is that with a SAN, the file manager and block allocation must reside at the client, whereas with the Sigma system, the file system resides at the storage system.

Architecturally, the closest comparison to the Sigma system is the NASD system where clients talk directly to "smart" disks. The data transfer protocol between clients and NASD devices is an object, which can be approximated as a file. The smart disks are the equivalent of the storage nodes in the Sigma architecture. The key difference is that the storage manager in a NASD system is located in a unit separate from the smart disks, while the equivalent of the storage manager in a Sigma system is integrated into the file system on the cluster itself. Also, whereas redundancy management is done at the client in the NASD system, the Sigma system integrates redundancy management into the cluster file system. These two differences allow the Sigma system to be compatible with existing distributed file systems.

There are two main components to our clustered file system. The first component is the distributed file system layer that implements the NFS and CIFS protocols. The second is the cluster file system layer referred to as the Sigma Cluster File System (SCFS). Both layers run on the server and require no modifications of the client residing on the network.

The interface between the two layers is defined by an API that is similar to POSIX IO library calls with additional support for NFS and CIFS locking semantics. We have called this API the *clientlib*. It should be noted that client in this context refers to the distributed file system layer, i.e. NFS or CIFS, as a client of the SCFS. For convenience we call each instance of a NFS and CIFS server that uses the clientlib API a clientlib instance or process. A network client will connect to one of the nodes in the cluster using NFS or CIFS file protocols. The distributed file system layer will handle the request, and translate the NFS or CIFS request into a SCFS request through the clientlib. The clientlib is responsible for resolving any directory path names specified in a NFS and CIFS request. Path resolution can involve lookups in multiple directories, and in such a case, the clientlib would perform the necessary communications to each directory object. To avoid excessive communication, the clientlib caches these directory lookups and uses leases to handle cache consistency. Path resolution is also an example of how the clientlib coordinates accesses when a network client request involves multiple SCFS objects. As another example, a rename operation can involve modifications to two different directories, and the clientlib again performs the calls to each directory object. The clientlib supports distributed locking maintaining consistency between network file system daemons running on different servers as well as differing network file system protocols. We omit the details of the clientlib's distributed cache and lock management as they are beyond the scope of this paper.

2.2. Virtual Devices

Whereas the distributed file system layer deals with files, the SCFS is concerned with *virtual devices*. A virtual device is an abstract container for a file or group of files. In the current implementation, each virtual device contains only one file and likewise each file is mapped to exactly one virtual device. The SCFS is responsible for the data striping of each virtual device. With the virtual device construct, the SCFS is able to assign different striping policies to each virtual device and thereby each file. Striping policies include deciding the number of nodes over which to distribute the data, and whether to use parity or mirroring for the redundancy strategy. Without the use of virtual devices, all files would be striped across all the nodes and the choice of redundancy strategy would be the same across all files. Virtual devices, however, allow certain files to be mirrored because they might require high performance and reliability while other files may be striped with no parity because they are not critical.

In the current implementation, each virtual device contains one file along with the metadata for that file. By grouping the file data along with its metadata, we are able to benefit from locality properties. Since directories are treated as special files, they get their own virtual device as well. The distribution of metadata into virtual devices also improves the scalability with respect to metadata operation. In general, most metadata operations can be done directly to a virtual device rather than through a centralized locking resource that can prove to be a bottleneck.

Each virtual device is identified by a 64-bit identifier known as the GID. This is the equivalent of an inode number. While, on most file systems, the inode number is sufficient to locate a file in the inode table and from there the actual data blocks, with the Sigma file system, the GID must also be grouped with a locator that identifies the virtual device. This locator specifies the type of striping – parity or mirroring – as well as the machines on which the data is located. The GID and locator information are stored in the directory entry that refers to the file, so a centralized database is not needed to maintain this information.

For the purpose of comparison, an entry in the UNIX

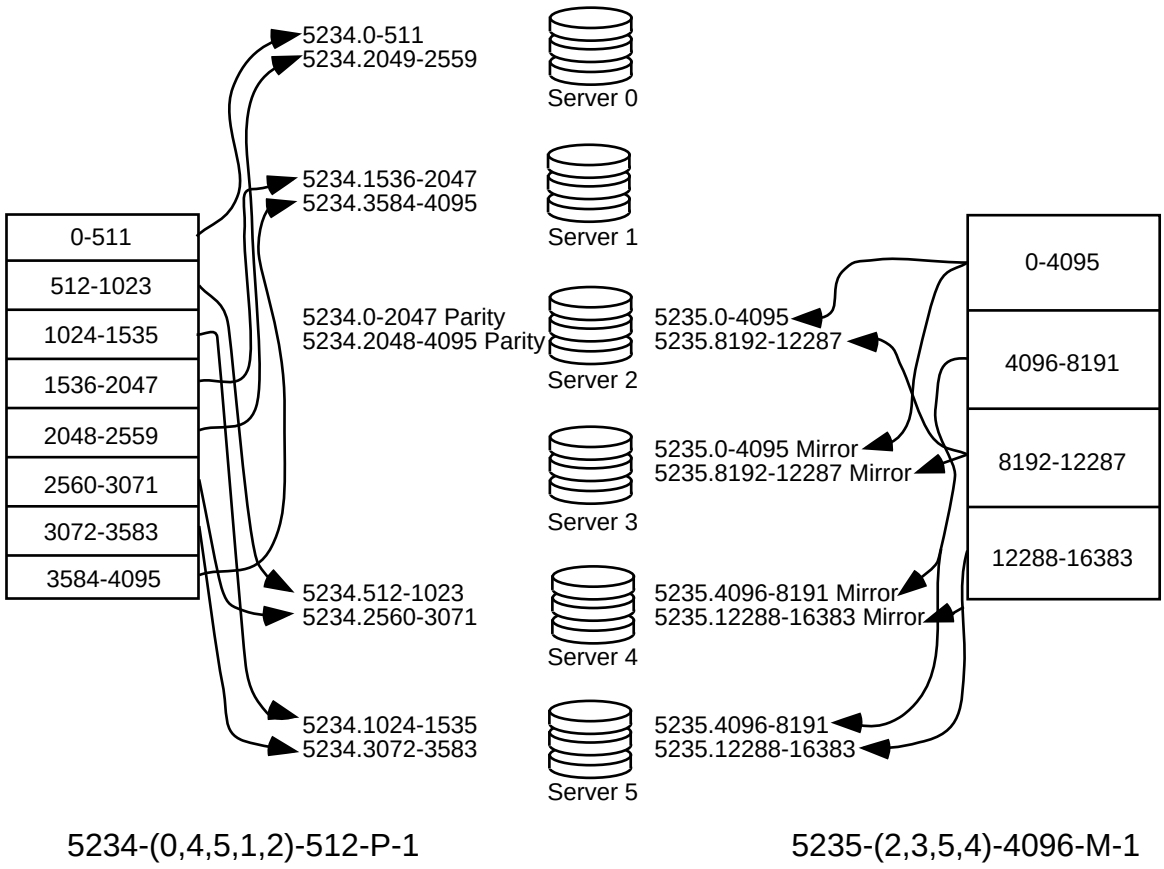


Figure 4. Virtual Device Data Distribution.

internal directory format contains two data fields, file name and inode number. The inode number is an index into a table stored on disk that allows a UNIX file system to locate the actual disk data blocks for this file. An entry in the SCFS internal directory format consists of the file name, the GID of the virtual device, and the locator. The locator gives the SCFS the information that allows it to locate the machines on which the data for the particular virtual device are located.

The format of the locator specification is as follows: *(GID-(MSPEC)-BLKSIZE-TYPE-REDUNDANCY)*, where *GID* is the GID of the virtual device, *MSPEC* is a tuple representing the machines hosting the data, *BLKSIZE* is the block size of the stripe, *TYPE* identifies the redundancy mode (P for parity striping, M for mirroring, and N for no redundancy), and *REDUNDANCY* specifies the level of redundancy. For parity striping, the redundancy level specifies the number of parity blocks per stripe. For mirroring, the redundancy level means the number of mirrors. The flexibility of this specification allows us to vary the redundancy level and striping mode per virtual device and thus on a file-by-file basis. Changing the block size also allows us to tailor performance characteristics depending on the usage patterns of the file. For example, large files with streaming access would have larger block sizes and smaller files could have smaller block sizes. It is also possible to redistribute files to a different set of machines if the file policy has changed – for example, a file has become higher priority and thus needs mirroring redundancy instead of parity.

The example in Figure 4 shows the data distribution for two virtual devices whose specifications are given as follows: (5234-(0,4,5,1,2)-512-P-1) and (5235-(2,3,5,4)-4096-M-1). The file on the left has been divided into 512 byte blocks and parity striped across servers 0,4,5,1, and 2. For parity striping, the last machine is reserved for parity. The file on the right has been striped into 4096 byte blocks and mirrored across servers 2,3,5, and 4. Note that the order of machines in the *MSPEC* tuple need not be sequential. For parity striping, this is significant; since different virtual devices will have different *MSPEC* distributions, the machine reserved for parity differs for all virtual devices. Therefore, even though we use a RAID-4 type parity scheme for each virtual device, across all virtual devices we do not suffer from the typical RAID-4 parity bottleneck. For mirroring, the mirrors are assigned dependent on the redundancy level. With a redundancy level of one, the cardinally odd machines in the tuple are the primary data machines and the even machines are the mirrors. In

the example shown, the data is striped across machines 2 and 5 with mirrors on 3 and 4.

The use of virtual devices also enables easy addition of new machines into the cluster. When a new machine is added into the cluster, there is no need to do a reformat as is necessary when adding single drives to a RAID-5 array. The new machine will initially have no data located on it, but as new files are created, the associated virtual devices will include the new machine. If the cluster is particularly unbalanced, whereby the existing machines have no storage space left, virtual devices can be rebalanced to include the new machine. This rebalancing process can proceed with a live system – i.e. the system does not have to be brought down while the rebalancing takes place.

Rebalancing will cause the locator information to change and the corresponding directory entry copy of the virtual device locator may be out of date. After a failure, during reconstruction of data, the same situation may arise causing invalid directory entry copies of locator information. The SCFS has the ability to determine if locator information is invalid, and then automatically find the correct locator information and update the incorrect directory entries.

The SCFS is implemented using a collection of process objects which provide various system services. Of particular importance is the Virtual Device Controller (VDC) object which performs the actual striping functions for a virtual device. A VDC or set of VDCs is instantiated on each node and at any point in time, a VDC may host zero, one, or more virtual devices. However, a critical point is that a virtual device will be hosted by only VDC at a time. This allows us to avoid the difficult issues associated with concurrent access/modification to a block device across a storage area network. For performance reasons, a virtual device is usually hosted by a VDC running on the same machine as the distributed file system process accessing it. Since there is one virtual device per file, there could potentially be millions of virtual devices in the system. To avoid the overhead of a VDC managing a million virtual devices, in practice only "active" virtual devices are managed by a VDC, where active is defined as a virtual device having recent activity.

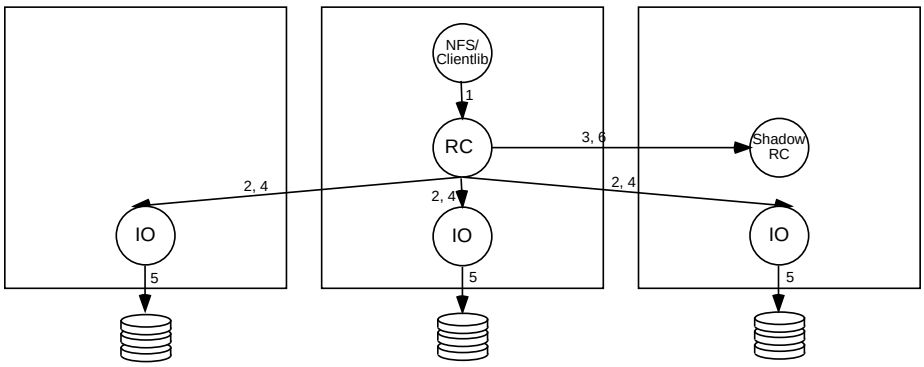
To further describe the SCFS, we examine the flow of an NFS read request. Assuming that the client has already mounted the cluster file system locally, it sends a read request to any node in the cluster. The node, which we will call the *receiving* node can be arbitrary since all nodes present the same view of the file system. In particular, with NFS, any subsequent requests could also be sent to a different node, since the NFS protocol is stateless. The read request contains a unique filehandle identifying the file to be read, an offset into the file, and the number of bytes to read.

The NFS layer on the receiving node will query the SCFS to identify which virtual device contains the requested file. In addition, the SCFS will return an identifier specifying which VDC is responsible for the particular virtual device. The NFS layer will then translate the NFS read request into a read request for the VDC. The VDC responsible for the file need not be located on the receiving node, and in such a case the VDC read request must be sent to a remote node. In practice, because of locality, the VDC is almost always on the same node as the receiving node.

The VDC upon receiving the request will determine the actual location of the data. Since the data has been striped across multiple nodes, it must fetch the data from each of those nodes. Which nodes to contact is determined by the striping policy associated for the particular virtual device. After receiving the data from the nodes, the data is gathered and reconstituted and then returned to the NFS layer which then forwards it on to the NFS client.

In the context of a distributed system, writes are more interesting – particularly on parity striped writes. Concurrent access to shared data introduces difficulties in managing consistency of data, and in the presence of failures, these difficulties become even more challenging. In a typical RAID-5 disk array care must be taken such that partial writes do not occur. As an example, consider the situation where we are writing data that spans disks A, B, and C with parity on disk D. Since it is not possible to atomically write to all disks simultaneously, it is possible that the parity disk D may be updated before the data disks. If there is a system failure during the writes, the data will be corrupted, since the write has been only partially completed. Because the parity is inconsistent, the data will be irrecoverable on a subsequent disk failure.

With a clustered system, this problem is magnified. To solve this problem, we use a modified two-phase write-commit protocol. In the first phase, the VDC will issue write commands to the appropriate nodes. The parity is calculated and sent to the node hosting the parity for this device. However, the nodes do not actually flush the data to stable storage at this time. They hold on to the data waiting for a commit from the VDC. After sending the data to the node, the VDC will then notify a "shadow" VDC running on another node that a write has been initiated to a particular set of nodes. Then, the primary VDC will issue commit commands to all the involved nodes, which will then complete the write. See Figure 5. If the primary VDC fails during the commit phase, the shadow VDC will notice this and will finish issuing the commits. If at any point during the commit phase, any of the involved nodes fail, the primary VDC will notice this and mark that particular region dirty in its local memory. This dirty region information is also conveyed to a SCFS service called the *fact server* that persistently maintains this information across the dis-



1. NFS receives write request and through clientlib forwards request to RC responsible for file
2. RC splits write data and sends it to appropriate IO nodes
3. RC notifies Shadow RC which IO devices are involved in write
4. RC sends commit commands to IO nodes
5. IO nodes commit data to disk
6. RC notifies Shadow RC that write is complete

Figure 5. Two-phase Write Commit Protocol.

tributed cluster. If during a subsequent read, the VDC sees that the requested data has been marked dirty, the VDC can then retrieve the data using the parity.

2.3. System Services

In addition to the VDC, the SCFS is implemented with the use of a collection of system processes or services that run on the cluster machines. These processes communicate with each other as well as with the distributed file system layer to provide full file system functionality. The main cluster file system services are the Global Controller, the IO Daemon, Status Monitor, and the File System Integrity Object. Each server may run multiple or no instances of particular services.

2.3.1. Global Controller The Global Controller (GC) ensures that access to each virtual device is granted exclusively to only one VDC at a time. This is not a potential source of deadlocks, because the assignment of a virtual device to a particular VDC does not restrict access to the virtual device. Rather, all access to the virtual device will now be serialized through the VDC to which it is assigned.

In order to be sure that there are no access conflicts between VDCs running on different machines, the GC runs on only one machine. At start-up, the cluster machines engage in a multi-round election protocol to determine which will host the GC. In order to avoid potential bottlenecks, multiple GCs can be used. Access conflicts can be successfully avoided when using multiple GCs by assigning each of the virtual devices to only one GC according to some deterministic hash function. For example, it is possible to use two GCs, by assigning the odd numbered virtual devices to one of the GCs, and assigning the even numbered virtual devices to the other.

In the event of a GC failure, the cluster machines will once again engage in an election protocol to determine the new GC machine. Any new requests to the GC will stall until the new election is complete and the new GC has been able to determine the existing assignment of virtual devices to VDCs in the cluster.

2.3.2. IO Daemon The IO daemon handles the actual transfer of data from the disk storage system. The current implementation uses the underlying file system to store data. The only services that the IO daemon requires from its underlying file system are random access to file data a single level name space, the ability to grow files when written to past the EOF, and the only required metadata is file size. Thus, the IO daemon could use a simplified file system to write to raw disk to provide a more efficient implementation.

2.3.3. Status Monitor The Status Monitor determines the status (up or down) of the servers comprising the cluster and makes this information available to other processes running on the same server. The status of the other servers is determined by polling special monitors running on those machines. The shadow VDC makes use of the status monitor to determine the status of the primary VDC.

2.3.4. File System Integrity Object The File System Integrity Layer (FSI) performs two functions. First, it prevents two clientlibs from performing conflicting File System Modification (FSM) operations, for example, both clientlibs renaming the same file at the same time. Before the clientlib performs an FSM, it attempts to lock the necessary entries with the FSI. After the necessary File System Objects have been modified, the clientlib unlocks the entries. Second, it acts as a journaling layer, by replaying operations from clientlibs that are on machines that fail. In

the current implementation, there is only one FSI for the entire cluster and this is clearly a scalability issue as cluster sizes get very large. We anticipate that we will have to support multiple FSIs by partitioning the file system.

2.4. Performance Characterization

As with any storage system, performance is a critical feature of the system. However, striping data across servers for reliability is in potential conflict with performance. The overhead of communicating with multiple servers to satisfy data access can be significant. To address this, the system uses data caching at three levels: the block layer, the VDC layer, and the distributed file system layer.

The lowest level of caching is in the underlying file system. The IO daemon takes advantage of the page block caching present in most file systems. On writes, the IO daemon also uses caching and does not force a sync to disk. We do not need to do a sync because both the VDC and shadow VDC will monitor the IO machine. If the IO machines fails before the underlying file system flushes its caches (typically 30 seconds), the VDC will mark any regions to which writes are pending as dirty, thus alerting future reads to reconstruct the data from the parity or mirror.

The second level of caching is at the VDC layer. All reads are cached and the hit rate using common benchmarks was seen to be over 95% using a 10 megabyte cache. Writes are cached using three different synchronization policies: cluster sync, local sync, and async. Cluster Sync means writing through the cache all the way to the cluster; in other words, all writes are committed to the destination machines through the IO daemon as described above.

Local sync means that the write data is committed only to the local disk. The local disk sync data is flushed to the cluster storage every few seconds. This presents the possibility of possible data loss if the local machine suffers complete failure before the data has been flushed. The data is still recoverable if the local disk is recoverable through RAID. However, there is no corruption of the cluster storage and data is still sequentially consistent. Also, the file will be inaccessible if any local sync data has not been flushed to the cluster. Depending on the application, this synchronization policy may be acceptable. Since this policy is set on a per-write basis, not on a file system or file basis, it is possible to tune this as the application demands.

Asynchronous synchronization is the highest performance synchronization scheme, since the write data is only kept in memory and not pushed to any form of stable storage. As with local sync, data loss is possible if the local machine completely fails before the data has been flushed. In this case, even if the disk is recoverable, the data is still completely lost. In certain applications, this behavior may be acceptable. For example, NFS (version 3) allows for asynchronous writes whereby writes need not be sent to stable storage until a commit command has been issued.

The highest level of caching is done at the clientlib level. The first form of caching is to allow the clientlib to cache directory and meta-data information in its local address space with the server process promising to increment a counter maintained in shared memory when the information becomes stale. When the invalidation occurs, the network clientlib process communicates to the server processes to get the updated information. An alternative to maintaining only cache invalidation information in shared memory would be to maintain the metadata and directory information itself in shared memory, though these data structures are complicated to implement.

The network client processes are also allowed to obtain leases for the file system objects on which they are working. When there is contention among several network client processes for the same file system object, the network client releases its lease, and both client processes then use the standard mechanisms to access the file system object. This reduces potential context switching between network client processes and processes providing system services.

The most aggressive form of clientlib caching is what is known as preborn caching. We take advantage of the fact that many files are very short lived. This is what is observed in typical office environments as evidenced by the VeriTest NetBench benchmark [30] and also in development environments as seen in the BSD and Sprite studies [4] and HP/UX studies [22]. The NetBench benchmark is a simulation of typical office usage drawn from actual traces. During runs of the benchmark, we saw that 90% of files were deleted within 10 seconds of being created. Likewise, the Sprite trace-driven study showed that 50 to 70% of file lifetimes are less than 10 seconds, and the HP/UX study found up to 40% of block lifetimes were less than 30 seconds. This short-livedness property allows us to use a form of local sync caching where the clientlib creates new files on the local storage system and then pushes them to the LCC after 10 seconds. By doing so, we avoid making costly metadata updates and directory operations. The same caveats that apply for local sync apply here as well, in that files may not be accessible if the server hosting the preborn file fails before the file has been pushed to the LCC.

2.5. Implementation

In keeping with the low cost philosophy of the system, the target architecture chosen for the cluster servers was off-the-shelf x86 PCs running the 2.4 Linux kernel. However, because of the portability of the file system, the software has also been effectively ported to Solaris and OpenBSD as well and can be ported to any POSIX OS with

minimal difficulty. Interprocess communication is accomplished using the SunRPC remote procedure call library. The software architecture is very modular allowing for the replacement of specific modules for more machine-specific implementation if appropriate. For example, the RPC module could be easily replaced if the target architecture supports a higher performance IPC mechanism. Likewise, the IO daemon could be replaced to take advantage of low-level I/O calls that may be available in the host system.

The current version of the software contains support for both the CIFS and the NFS protocol. As NFS is a fairly simple protocol, we implemented our own NFS server, which accessed the Sigma Cluster File System using the clientlib. Our support for CIFS was provided by implementing a module (again using the clientlib interface) that plugs into Samba, an open-source CIFS implementation. Our clientlib used a VFS style API that was introduced in Samba, as of version 2.2. However, this VFS API has some shortcomings. For example, CIFS "oplocks" are not sufficiently exposed in the API, and it required some additional work to get oplocks and other features working in a cluster setting.

Each system service is implemented as a separate process, but individual services are themselves multithreaded. We chose to keep services as processes rather than threads to improve reliability. If one process used many threads to provide several system services, an errant system service taking an exception could crash the entire process thereby bringing down several system services. While the file system could temporarily tolerate the absence of some system services, the scenario does open the system to additional failures while the downed system services are brought back up. By putting each system service in a separate process, errors are localized to a particular service. As the reliability of the code improves to the point where fatal exceptions are nonexistent, we will gradually move to a fully multithreaded system.

The entire system was implemented in user space, using standard POSIX interfaces. This was a natural decision for several reasons. First, the file system was distributed, and the kernel is not the best place to write network client code. Secondly, the file system was designed to support access from NAS clients only as opposed to applications running on the local system. In addition, while there was a temptation to implement the file system according to the Linux kernel VFS API as opposed to developing our own clientlib interface, forcing data bound for the CIFS world to pass through the Linux Kernel VFS would have made implementation of some CIFS semantics more difficult.

The potential drawback to a user space implementation is performance, with the primary concern being potentially excessive context switching between the UNIX processes dedicated to serving network clients and the UNIX processes dedicated to serving requests to the file system objects. Context switching can also be apparent between the various file system service processes as well. In a file system implemented in the kernel, all network client processes would just switch to kernel mode and access and modify the data in the kernel data structures, protected by locks as appropriate. The mechanisms that we have used to do clientlib leasing of file system objects drastically reduces the context switches since the clientlib is able to do most operations without involving the file system service processes. While a kernel implementation would provide some additional performance benefits, since we were in a cluster setting, we felt it was more important to first focus on protocols and scalability rather than maximizing the performance of a single box. As we move to a fully multithreaded system service model, the cost of context switching between system service processes should decrease as well.

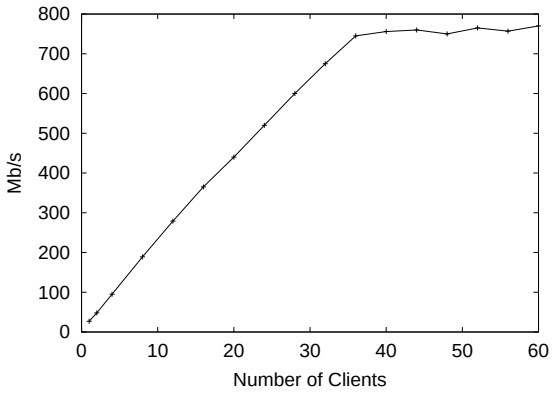


Figure 6. NetBench Results

3. Performance Results

For the purpose of our experiments, we constructed a small cluster with five equivalent dual Pentium-III 1GHz PCs running a version of the 2.4 Linux kernel. Each PC was equipped with a single gigabit network card as well as a single 40G IDE drive to provide storage. No special kernel optimizations were done to optimize I/O or inter-process communications. One server was dedicated to servicing FSI requests and the other four servers were available to service clients. Each was running a distributed file system layer for the CIFS protocol as well as the SCFS layer and all its required services. Each client was a Windows PC running the NetBench suite of enterprise tests. NetBench is a standard CIFS network file system benchmark published by VeriTest [30].

As the results in Figure 6 show, we were able to scale linearly to 750 Mb/s until 40 clients. After that, the servers became saturated, but throughput was maintained without

	Peak Throughput (Mbps)
Base Performance	24.91
No VDC caching	23.76
No preborn caching	21.77

Table 1. Single Client Results

dropping. The single machine throughput for each server is 200 Mb/s, so the efficiency with four active servers was nearly linear as well. Since we moved the FSI, the primary potential bottleneck, to a separate server, we were able to measure the effect as clients increased. The load on the FSI is also a good measure of the system's metadata scalability. At maximum throughput, the FSI machine experienced only 15% CPU load, so it is expected that we could comfortably expand the cluster size by a factor of six with little impact on scalability. A cluster of 24 machines could potentially provide over 4 Gb/s of throughput.

For further analysis, we also adjusted various features in the file system to see the effects on performance. For this set of tests, we used a single client with the NetBench benchmark. These single client results aren't comparable with the above because we decreased the waiting time between client requests. The results are shown in Table 1. The base performance is 24.91 Mbps for the single client. When we turn off VDC data caching or the preborn caching, we see that performance does not decrease significantly. Since NetBench is very cache intensive, it can be argued that the above numbers were achieved because of the cache. The numbers in Table 1 show that even when the cache is turned off in the Sigma system, the performance is not affected appreciably.

4. Related Work

Since a key differentiator of the SCFS is the notion of virtual devices, it may be instructive to compare virtual devices to similar concepts in the storage area, such as logical devices, NASD objects [10], derived virtual devices [29], and virtual disks [15].

A logical device is a common layer underneath most modern file systems that presents a single monolith block device view of a collection of block devices. As far as the file system is concerned, the logical device looks like a single large device from which it can allocate blocks. In contrast, a virtual device is a file oriented abstraction rather than block oriented, and as such it is an integrated part of the file system. In other words, a traditional file system resides on a *single* logical device, but the SCFS is composed of *multiple* virtual devices. With a logical device, redundancy and file system caching are implemented at the device level, meaning policy decisions about caching behavior and redundancy levels can only be made for the entire file system not on a file-by-file basis. Virtual devices allow these decisions to be made on a file-by-file basis.

NASD and derived virtual devices are similar in that they both are proponents of the "smart" disk concept. NASD objects exhibit properties similar to virtual devices, in that they may act as a collection of files. However, virtual devices lie above the striping layer, whereas a NASD object falls below the striping layer. By doing so, in a NASD system, the client is responsible for doing data distribution for striping. Since NASD objects are block based and thus requires changes to the client software, it is not appropriate for NAS environments.

A similar idea is seen in the evolving T10 SCSI Object Based Storage specification [2]. Using Object Storage Devices (OSD), the lower level of a traditional file system, i.e. block allocation and mapping to physical storage, is moved from the server to the actual storage device. The specification also allows for the aggregation of OSDs to provide striping and redundancy on an object by object basis. However, the current understanding of the OSD specification is that it is directed to DAS systems, though it could be easily moved to a network setting using protocols such as iSCSI [24]. The SCFS would be an ideal backend for such a system because of the natural mapping between OSD objects and virtual devices.

Petal introduced the concept of "virtual disks" which can aggregate storage from multiple servers into a single unified block disk. The virtual disks allow for varying redundancy policies such as mirroring, parity, level of redundancy, etc. Virtual disks differ from Sigma's notion of virtual devices in that Petal's virtual disks are block-oriented while virtual devices are file based. This is due to Petal's separation of the file system into the separate Frangipani layer that sits at the client [28].

File systems such as GFS [5, 27], Calypso [9], and CXFS [26] have been created to enable multiple servers to share a pool of SAN block storage. CXFS offers journaling capabilities built upon SGI's commercial journaling XFS file system. GFS maps the clustered file system across a non-homogeneous "network storage pool" from which space is allocated. Calypso has a sophisticated recovery protocol to reconstruct state in the case of failure. Unlike the Sigma architecture, all these file systems depend on an architecture where the storage devices are expected to provide redundancy, usually RAID, to maintain data availability. This increases the cost of the system.

In the parallel computing arena, there has been a lot of work in providing file systems for supercomputing applications. Initial I/O architectures dedicated nodes in a massively parallel processor (MPP) to storage. File systems such as PVFS [7], PIOUS [18], PPFS [13], Galley [19] and RAMA [16] provide the mechanisms to distribute I/O across the MPP. These file systems typically assume local

storage at each node in the MPP, and the data is distributed across the nodes. However, unlike the Sigma system, data is not striped with parity across the nodes, meaning less reliability. Though each node may use redundant storage, if the node has lost connectivity to the rest of the MPP, the data from that node is no longer available.

Serverless storage file systems offer the closest comparison to the Sigma system. Previous work on such architectures include Zebra [11], xFS [1], and LegionFS [31]. Zebra and xFS both stripe data across the nodes in the cluster using a log-structured approach. xFS is a more scalable architecture because of its distributed metadata management. As with Petal/Frangipani, neither allow for file-level redundancy policy management. LegionFS is an object-based distributed file system, but it has no redundancy features.

None of these file systems have been designed with standard network file systems in mind. Either the applications are required to run on the cluster nodes as with GFS and the MPP file systems, or the network clients must support a non-native distributed file system such as with xFS or Frangipani. As such, they are not easily integrated with standard network file systems such as NFS or CIFS, particularly with respect to the locking semantics of these file systems.

5. Conclusions and Future Directions

In this paper, we have described a highly scalable and reliable system for network-attached clustered storage. We are able to achieve throughput of 750 Mb/s for a modest 4-machine cluster and a theoretical 4 Gb/s throughput for a 24-machine cluster. Because of the design, we are able to use low cost components to achieve these numbers and still maintain high availability. The new contributions are the ability to provide file-by-file redundancy management and seamless ability to add new nodes. In addition, unlike many other clustered file systems, the Sigma file system is not dependent on client modifications since it is fully compatible with common distributed file systems such as NFS and CIFS.

As we move to larger clusters, the FSI object becomes more of a bottleneck. We plan to provide mechanisms to partition the file system so that multiple FSI objects can be instantiated to improve scalability. Another issue is that most intra-cluster communications is done using Sun-RPC over TCP. The overhead of TCP can be quite significant. We are in the process of investigating the use of low-latency communications protocols such as Myrinet [6] and VIA [8].

Another candidate for replacement is the SunRPC remote procedure call mechanism. In the context of a homogeneous cluster such as we have targeted, some of the features of RPC, particularly XDR, are not relevant and could be removed. Moreover, RPC is an inherently synchronous communications mechanism. This limits the scalability since it causes all interprocess communications to block. Using multiple processes to service requests can partially offset this effect. However, increasing the number of processes can tax the available resources on a server. We are investigating asynchronous communication libraries to replace the use of RPC.

There is also room for improvement in the protocols between various parts of the cluster file system. For example, the VDC hosting a virtual device might detect that all accesses to the virtual device are read only. In this case, the VDC could grant the readers the right to contact the IO daemons directly. A method to revoke this access when necessary would need to be provided.

6. Acknowledgments

We are indebted to the support of the Sigma Storage Corp. team including Matthew Ryan, Eric Fordelon, Neil DeSilva, Charles Katz and Brian Bishop. We are also grateful to Quantum/Snap Appliances, particularly Luciano Dalle Ore, for providing access to their QA lab so we could conduct the NetBench testing.

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proceedings of the Symposium on Operating System Principles*, pages 109–126, Dec. 1995.

[2] ANSI. *Information Technology - SCSI Object Based Storage Device Commands (OSD)*, Mar. 2002.

[3] Auspex Systems. *A Storage Architecture Guide*, 2000.

[4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Symposium on Operating System Principles*, volume 25, pages 198–212, Oct. 1991.

[5] A. Barry and M. O'Keefe. Storage clusters for Linux. Whitepaper, Sistina Software, 2000.

[6] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[7] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the Annual Linux Showcase and Conference*, pages 317–327, Oct. 2000.

[8] Compaq Computer Corporation, Intel Corporation, and Microsoft Corporation. *Virtual Interface Architecture Specification*, Dec. 1997.

[9] M. Devarakonda, A. Mohindra, J. Simoneaux, and W. H. Tetzlaff. Evaluation of design alternatives for a cluster file system. In *Proceedings of the USENIX Technical Conference*, Jan. 1995.

[10] G. A. Gibson and R. Van Meter. Network attached storage architecture. *Commun. ACM*, 43(11):37–45, Nov. 2000.

[11] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Trans. Comput. Syst.*, 13(3):274–310, Aug. 1995.

[12] D. Hitz, J. Lau, and M. Malcolm. File systems design for an NFS file server appliance. In *Proceedings of Winter USENIX*, Jan. 1994.

[13] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the ACM International Conference on Supercomputing*, pages 385–394, July 1995.

[14] P. J. Leach and D. C. Naik. A common internet file system (CIFS/1.0) protocol. Draft, Network Working Group, Internet Engineering Task Force, Dec. 1997.

[15] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Oct. 1996.

[16] E. L. Miller and R. H. Katz. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing*, 23(4–5):419–446, 1997.

[17] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Commun. ACM*, 29(3), Mar. 1986.

[18] S. A. Moyer and V. S. Sunderam. Pious: A scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[19] N. Nieuwejaar and D. Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.

[20] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, pages 23–36, Feb. 1988.

[21] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Mangement of Data*, pages 109–116, June 1988.

[22] D. Roselli, J. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the USENIX Technical Conference*, June 2000.

[23] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Technical Conference*, pages 119–130, June 1985.

[24] J. Satran, D. Smith, K. Meth, O. Biren, J. Hafner, C. Sapuntzakis, M. Bakke, R. Haagens, M. Chadalapaka, M. Wakeley, L. Dalle Ore, P. Von Stamwitz, and E. Zeidner. iSCSI. Internet Draft, IPS, Apr. 2002.

[25] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, Apr. 1990.

[26] Silicon Graphics, Inc. *SGI CXFS Clustered Filesystem*, July 2000.

[27] S. Soltis, G. Erickson, K. Preslan, M. O'Keefe, and T. Ruwart. The design and implementation of a shared disk file system for IRIX. In *Proceedings of the NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, Mar. 1999.

[28] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the Symposium on Operating System Principles*, pages 224–237, 1997.

[29] R. Van Meter, S. Hotz, and G. Finn. Derived virtual devices: A secure distributed file system mechanism. In *Proceedings of the NASA Goddard Conference on Mass Storage Systems and Technologies*, Sept. 1996.

[30] VeriTest. NetBench 7.0.2. "http://www.veritest.com/benchmarks/netbench/netbench.asp", 2001.

[31] B. S. White, M. Walkder, M. Humphrey, and A. S. Grimshaw. LegionFS: A secure and scalable file system supporting cross-domain high-performance applications. In *Proceedings of Supercomputing 2001*, 2001.