Data Integrity in a Distributed Storage System

Jonathan D. Bright
Sigma Storage Corporation
Emeryville, CA U.S.A.

John A. Chandy
Department of Electrical and Computer Engineering
University of Connecticut
Storrs, CT, U.S.A.

Abstract ***Distributed storage systems must provide highly available access to data while maintaining high performance and maximum scalability. In addition, reliability in a storage system is of the utmost importance and the correctness and availability of data must be guaranteed. We have designed the Sigma cluster file system to address these goals by distributing data across multiple nodes and keeping parity across these nodes. With data spread across multiple nodes, however, ensuring the consistency of the data requires special techniques. In this paper, we describe fault tolerant algorithms to maintain the consistency and reliability of the file system - both data and metadata. We show how these techniques guarantee data integrity and availability even under failure mode scenarios.***

Keywords: clusters, fault-tolerant algorithms, storage systems, distributed locking

1 Introduction

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 [1], NFS [2], Sprite [3], CIFS [4], amongst others. However, with a single file server for a large number of clients, the scalability is limited. In an effort to remove the bottleneck of the single server model, there has lately been significant work in the area of clustered or distributed storage systems. These include distributing data amongst dedicated shared storage nodes [5] and network-attached secure disks (NASD) [6], or distributing the data amongst the clients themselves in so-called serverless storage systems [7]. The migration to these systems has been driven by the need to increase concurrent access to shared data.

Concurrent access to shared data requires unified file systems or storage managers associated with these distributed storage systems [8, 9, 10]. In order to improve data bandwidth, these file systems or storage managers typically stripe the data across the nodes in the storage system. The distribution of shared data, however, introduces difficulties in managing concurrent access to this data. Algorithms to ensure consistency in shared data access have been proposed in previous reported work [11, 12, 5]. In addition to maintaining data consistency, a key requirement of these file systems must be the ability to deal with potential system failures. With multiple nodes in a cluster, the likelihood of failure of a single node increases. This means that any data consistency algorithms must also be fault-tolerant. In addition, the data must not only be correct but also be available in the face of failures in the system. These failures may be at any level from network interface, to processor, to disk. A simple technique to increase data availability is to stripe data using RAID [13] techniques such as parity protection or mirroring. However, applying RAID techniques to a clustered storage system introduces new challenges.

In this paper, we present algorithms for managing consistent access to both the metadata and data in a file system for distributed storage, in particular for the Sigma cluster file system [14]. We describe techniques used to ensure a functioning system in the face of failures. The paper is organized

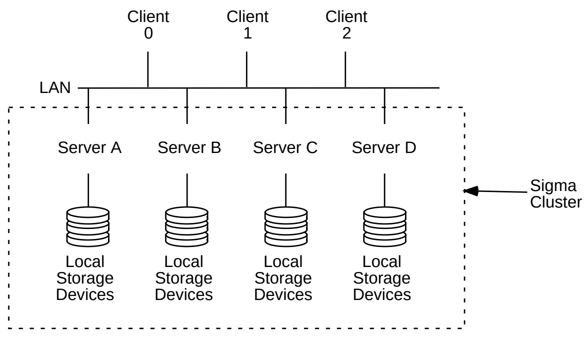

Figure 1: Clustered Storage.

as follows: we first provide a brief background of the Sigma cluster file system and storage architecture. Then in subsequent sections we describe our algorithms to preserve consistency and data availability. In addition to just a theoretical description of the algorithms and techniques, we also discuss some of the issues and challenges of software fault tolerance with an implementation of the file system.

2 Sigma Cluster File System

The Sigma Cluster File System (SCFS) is based on a clustered storage architecture. The physical layout is shown in Figure 1. Clients can connect to the Sigma cluster using a distributed file system protocol such as NFS or CIFS. However, unlike a traditional single-server model, 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 allow the Sigma system to eliminate the single-server bottleneck of a traditional file server system. In this model, applications do not run on the cluster nodes but run on client nodes and communicate with the cluster storage through NFS or CIFS protocols.

NFS and CIFS data requests are translated through a POSIX-like API called the *clientlib*. The clientlib takes these requests and conveys them to the SCFS, which is distributed across the nodes of the cluster. The SCFS is responsible for the file management as well as data distribution, i.e. striping data across nodes using varying redundancy policies. The SCFS is implemented using a collection of process objects which provide various system services. These processes run on the cluster machines and communicate with each other using remote procedure calls (RPC) to provide full file system functionality. Each cluster node may run multiple or no instances of particular services.

The SCFS distributes data blocks from a file in a round-robin fashion across a subset of the nodes in its cluster. In other words, even though clients communicate with a single node in the cluster using a file access protocol, the cluster file system will automatically convert that file access protocol into multiple distributed block access requests to other nodes in the cluster. In addition, to improve data reliability, the files are distributed with either parity or mirroring redundancy.

3 SCFS Consistency

In this section, we discuss methods to ensure file data integrity in the SCFS. It is clear that distributing data across multiple nodes can cause concurrency problems. For example, if multiple clients are writing to the same file, the data that is written to the blocks may not be sequentially consistent, unless there is proper access control. If the data is distributed using parity, the problem could become a latent error, as the data may be consistent, but the parity is inconsistent, meaning that the error is not apparent until a failure occurs and parity is actually accessed to reconstruct the data. Under failure scenarios, even access control scenarios may not be enough to prevent data corruption. We discuss algorithms to address this problem in section 4.2.

3.1 Virtual Device Controller

We have introduced a concurrency control mechanism similar to the device-served locking method described in [11]. In order to ensure safe access to the files, we introduce a file-based locking mechanism. Each file is mapped to a per-file abstract object called a virtual device which is in turn managed by exactly one of a set of virtual device controllers (VDCs) distributed throughout the cluster. A VDC is one of a set of processes 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 ensures that any accesses to that virtual device will be serialized through the hosting VDC, and thus guaranteeing safe and consistent access. Moreover, 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. Deadlock is not an issue because the locking/serialization is at the file level rather than at the finer granularity block level where multiple locks may need to be acquired.

In order to guarantee exclusive ownership of a virtual device, the SCFS provides a system service known as the Global Controller (GC). The GC is instantiated on a single machine and it grants exclusive ownership of virtual devices to VDCs. The GC is not a significant bottleneck, since its functionality can easily be distributed amongst many instances. For example, if there are two instances of the GC, even-numbered virtual devices may be assigned to one instance and odd-numbered virtual devices would be assigned to the other instance. We will go into more detail about the GC in the following section.

To make the process more concrete, we offer this example. Client 1 contacts server A to write to file “foo”. Server A contacts the GC to determine who the owning VDC is. The GC responds that the owning VDC is on server C. At the same time, client 2 contacts server D to read from file “foo”. It too is told by the GC that the VDC for the file is on server C. The clientlibs on both server A and server D will forward the client requests to the appropriate VDC on server C. The VDC will then process the requests from client 1/server A and client 2/server D in sequential order. There is no guarantee on ordering, but the VDC will guarantee that only one request proceeds at a time.

Serialization through a VDC allows us to avoid the scalability limitations of centralized locking mechanisms. The drawback is the overhead when there is significant contention on a single file. In typical file usage patterns, contention on a single file is not an issue since it rarely happens.

3.2 Global Controller

We now describe in more detail the GC, and how the GC interacts with the VDCs processes running on every machine. This protocol had several principal design goals. Two VDCs must never host the same VD at the same time. It must handle machine, network, and possibly software failures, without generating inconsistent states and it must be efficient.

The GC must be instantiated by the cluster on one and only one machine. This instantiation process is an example of the well known leader election problem. There has been significant work in this area and our algorithm is similar to previous work [15]. We elect the GC using an asynchronous-iterative election protocol. By asynchronous-iterative, we mean that every machine iterates independently of the other machines in the cluster, and that each machine independently comes to a final consensus about which machine in the cluster is the GC. We don’t describe the algorithm in further detail, since the problem is well studied and our approach is not unique.

Recall, that every machine runs zero or more instances of the VDC service. A separate process, called the Local Controller (LC) manages all of the VDC processes on a machine. When the LC starts up, it spawns (using POSIX fork()) all of the VDCs for the machine. This (in the POSIX sense) parent/child relationship allows for signals to be sent from the LC to the VDCs without the worry of race conditions (i.e. if a VDC has abnormally terminated, but not been ”waited” for by the LC, the process ID will not be reused by the system, preventing the undesirable situation of an unknown process receiving the signal). Also, when the LC terminates, the VDCs will automatically be sent a signal by the kernel, and will immediately terminate themselves.

The interaction between the GC, the LC, and all of the VDCs on a machine can be quite complex. These services are normally invoked by an RPC mechanism, though the critical point is that only one protocol method is ever invoked at a time. Deadlock could occur if two services ever try to synchronously invoke each other’s methods at the same time (or if a cycle existed amongst the services.) But there are a few simple rules governing which services are allowed to invoke protocol methods of which other services, and a quick analysis makes it clear that deadlock is always avoided.

The VDC makes synchronous RPC calls to both the GC and to its controlling LC. The LC only ever makes asynchronous calls, and these are initiated only during LC startup and during error conditions. The GC makes asynchronous calls to the LC during error conditions, but also makes what might be considered semi-synchronous calls to the LC during startup, and we present more details in the following section. It should be clear that the described sequence of calls between the VDC, LC, and GC will never cause deadlock.

The GC also periodically checks to see if a machine has failed, using a service called the Cluster Status Monitor. This service periodically queries daemons on all the nodes to determine the status of all nodes in the cluster. In the case when the GC detects that a machine has failed, it revokes all the assignments to that machine and queues up an asynchronous call to LC shutdown() in case the machine was only temporarily unavailable.

4 Failure Mode Availability

4.1 Failure Model

Before discussing our algorithms for maintaining data availability, it is useful to explain the failure model assumed. We define availability of the file system as no loss of data under any single failure condition and continued response to client nodes. The potential failures in the system that we consider are permanent and transient failures to a single machine, cluster partitioning, and disk failures. We define these faults as clean errors. Our algorithms are completely clean-error-tolerant. Unclean errors are faults such as multiple machine failures and incorrect data.

4.2 Writes Under Failure

In the context of a distributed system, writes are more interesting than reads - particularly with 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. Problems can arise even with a single failures due to latent parity errors. 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 that are storing the data for this device. For the purpose of clarity in this example, we will call these nodes the storage nodes, though in reality they are equivalent to all other nodes in the cluster. The parity is also calculated and sent to the storage node hosting the parity for this device. However, the storage 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 storage nodes, the VDC will then notify a “shadow” VDC running on another node that a write has been initiated to a particular set of storage nodes. Then, the VDC will issue commit commands to the respective nodes, which will then complete the write.

If there is a single failure of any of the involved storage nodes before the commit has been issued, the primary VDC will still issue the commit. However, since one of the storage nodes has failed, its data and thus a region of the virtual device is no longer consistent. Since the VDC will not have received a confirmation response back from the failed storage node, it will know that it has failed. It will use that information to mark the inconsistent region of the virtual device as dirty. This information is kept local to the VDC. Therefore, during a subsequent read to that inconsistent region, the VDC will notice that it has marked that region as dirty and thus reconstruct the data using parity rather than reading from the failed storage node. The VDC will do so even if the failed storage node has in the mean time recovered from what was a transient error. This is key because if the VDC was to read directly from the previously failed storage node, it would get inconsistent data. The use of dirty region marking prevents this from occurring.

If the node hosting the primary VDC itself fails before the commit phase, the data will never get written to stable storage. The higher level clientlib call will eventually timeout leading to a timeout to the client application. The client can then reissue the request to write to the file - this process is automatic when using NFS. The clientlib upon receiving the new request will find that the VDC originally responsible for the file is no longer available. It will contact the GC which will grant ownership of the file to a different VDC on a working node. The write request can then be forwarded to the new VDC, and since we assume only a single failure, this time the write will succeed.

If the primary VDC fails after the commit phase has started but before it completes, the shadow VDC will notice this and will finish issuing the commits. If at any point during the commit phase, any of the involved storage nodes fail, the primary VDC will notice this and mark that particular region dirty in its local memory. As with the above scenario, the use of marked dirty regions prevents the use of inconsistent data.

4.3 Distributed Dirty Regions

Marked dirty regions are a key component to failure tolerance and recovery. However, since they are kept in memory local to a VDC, the information could be lost on a subsequent failure to the node hosting the VDC. To guard against this, the SCFS includes a service called the *fact server*. Whenever a VDC updates the dirty regions for a particular virtual device, it also conveys that information to the fact server. The fact server service acts like a distributed fault tolerant datastore, using a data replication scheme based loosely on quorum-based methods [16].

The fact server is implemented as a collection of processes - one instantiated on each node. A client of the fact server, such as the VDC, will send its fact, to the local instance of the fact server. In the case of the VDC, the fact is the dirty region. When the local instance receives the fact, it will send copies of that fact appended with a timestamp to two other fact server instances on other nodes in the cluster. In this manner, the fact is guaranteed to be existing on some node in the cluster in case of any failure. Moreover, the fact server will periodically flush its data to stable storage to guard against total system failures.

To illustrate the fact server behavior, let us return to the use of the fact server with dirty regions. When the VDC for a particular virtual device fails, the virtual device will get rehosted on another VDC. When control of the virtual device has passed to the new VDC, the new VDC can fetch the dirty region from the fact server and thus keep its internal copies of the dirty regions up to date.

4.4 Recovery and Reconstruction

After a machine fails, a human administrator can declare to the cluster that the machine has permanently failed, and initiate a reconstruction. While the software can automatically detect failures, human administrators can provide more meaningful information such as the cluster node being down simply for maintenance. In the normal case, a new machine will be added to the cluster before reconstruction is started, though this is not strictly required provided the remaining cluster machines have enough physical storage space to perform the reconstruction. Reconstruction proceeds in several phases. In the first phase, exactly which virtual devices that have been damaged is determined. In the second phase, the virtual devices are repaired.

Phase One As there is no centralized database which stores the stripe patterns of all of the virtual devices, this phase must necessarily be distributed. Every machine scans its local storage pool to determine all of the virtual devices which had a data stripe stored on the failed machine. Based upon a simple hash function of the virtual device number, this information is distributed to the cluster machines for analysis. For example, if there are four machines in the cluster, machine 0 would receive stripe information for the virtual devices numbers (0, 4, 8, etc). After all of this information has been distributed, each machine analyzes the stripe information for the subset of virtual devices that it is responsible for, and then determines which virtual devices have a missing data stripe. This phase runs relatively quickly, as it involves only a small amount of disk activity, and the mount of data transmitted over the network is small.

Phase Two In this phase, the damaged virtual devices are reconstructed. Based on a tunable parameter, one, two, or more virtual devices can be reconstructed in parallel by each machine.

This parameter affects the percentage of cluster resources devoted to reconstruction, and hence the more virtual devices reconstructed in parallel, the higher the response time for NFS/CIFS requests during reconstruction. When machine M reconstructs device D, it uses the standard cluster protocols for modifying virtual devices. Specifically, the GC/VDCs protocol is used to assign virtual devices to VDCs, and the fact server is updated as the reconstruction takes place. Because of this, NFS/CIFS clients can access and modify data while the reconstruction is taking place. Also, reconstruction can be safely and efficiently stopped and restarted.

Although most of the reconstruction algorithm is straight-forward, some care needed to be exercised in places. For example, before a new stripe column for a damaged virtual device is created on a new machine, the fact server needs to be notified that the data in the new stripe column is incorrect. Then the data is reconstructed and written to the stripe column, and finally the fact-server is notified that the stripe-column is now correct. By following this order, we guarantee correctness even if reconstruction is halted partially through.

5 File System Integrity

Most of the discussion so far has been concerned with ensuring the consistency and fault tolerance of reads and writes to data files. This section discusses the File System Integrity Layer (FSI), which ensures consistency and fault tolerance at the *file system* level, e.g. operations such as “create file”, “rename file”, etc. The FSI performs two main 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, the FSI cannot run on any machine that also has a clientlib running, otherwise, the replay will not occur on failure. Secondly, there is one FSI for the entire cluster. This is clearly a scalability issue as cluster sizes get large. We anticipate that we will have to support multiple FSIs by partitioning the file system.

When the FSI shuts down cleanly, for example, due to impending power outage, it flushes the state of all pending FSM operations. When the FSI restarts after a clean shutdown, it reads its prior state and resumes operation. When the FSI restarts after an unclean shutdown, before allowing new FSMs to proceed, it waits approximately 30 seconds to allow all existing clientlibs to re-register the FSMs that they were working on. A clientlib trying to register a new FSM during this time will block.

The FSI is implemented using a special virtual device for persistence, accessed through the VDC interface discussed earlier. This way, reliable persistence is achieved, and we are able to reuse the existing infrastructure to insure that at most one machine runs the FSI service at a time.

The FSI also allows the clientlib to lease an entire directory, and in this mode, the FSI maintains the contents of the entire directory, as opposed to only the outstanding FSM operations on that directory. (An implementation based on maintaining ”diffs” would be more efficient for extremely large directories.) In this mode, the clientlib does not need to perform the usual ”lock/unlock” protocol methods, and instead, just informs the FSI of the change to the directory. In addition, the update of the actual directory object can be delayed until the directory lease is released. These are useful performance enhancements when dealing with the rapid creation and deletion of temporary files.

6 Conclusions

In this paper, we have presented a series of algorithms for a fault-tolerant distributed storage system. The algorithms preserve data and file system integrity and consistency in the presence of concurrent reads and writes. We have shown how these protocols are designed to be tolerant of single-fault failures both transient and permanent. The protocols are designed with fault-tolerance as a principal design commitment rather than as an afterthought. We have also described practical implications of fault tolerant algorithms and their relationship to software in a real system.

References

[1] 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*, vol. 39, pp. 447–459, Apr. 1990.

[2] 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*, pp. 119–130, June 1985.

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

[4] 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.

[5] 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*, pp. 84–92, Oct. 1996.

[6] G. A. Gibson and R. Van Meter, "Network attached storage architecture," *Communications ACM*, vol. 43, pp. 37–45, Nov. 2000.

[7] 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*, pp. 109–126, Dec. 1995.

[8] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of USENIX Conference on File and Storage Technologies*, pp. 231–244, Jan. 2002.

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

[10] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka, "File server scaling with network-attached secure disks," in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.

[11] K. Amiri, G. A. Gibson, and R. Golding, "Highly concurrent shared storage," in *Proceedings of the International Conference on Distributed Computing Systems*, Apr. 2000.

[12] D. Anderson and J. Chase, "Failure-atomic file access in an interposed network storage system," in *Proceedings of the IEEE Symposium on High Performance Distributed Computing*, Aug. 2000.

[13] 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*, pp. 109–116, June 1988.

[14] J. D. Bright and J. A. Chandy, "A scalable architecture for clustered network attached storage," in *Proceedings of the IEEE Symposium on Mass Storage Systems*, 2003. To appear.

[15] J. Brunekreef, J.-P. Katoen, R. Koymans, and S. Mauw, "Design and analysis of dynamic leader election protocols in broadcast networks," *Distributed Computing*, vol. 9, no. 4, pp. 157–171, 1996.

[16] R. H. Thomas, "A majority consensus approach to concurrency control for multiply copy databases," *ACM Trans. Database Systems*, vol. 4, pp. 180–209, June 1979.