Parity Redundancy in a Clustered Storage System

Sumit Narayan John A. Chandy

Department of Electrical & Computer Engineering
University of Connecticut, Storrs, CT USA 06269–2157
Email: {sumit.narayan , john.chandy}@uconn.edu

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. Adding parity redundancy to distributed storage systems has been problematic because of the impact on performance. In this paper, we investigate mechanisms to add redundancy to the Lustre cluster file system with minimal effect on overall system performance. With data spread across multiple nodes, ensuring the consistency of the data requires special techniques. We describe fault tolerant algorithms to maintain the consistency and reliability of the data. We show how these techniques guarantee data integrity and availability of systems for read and write even under failure mode scenarios.

1. Introduction

Direct Attached Storage (DAS) was the first storage solution, and is still the simplest storage solution. In a DAS system, a disk hardware is directly connected to the server using high speed channels like SCSI or IDE. However, development in network speeds and demand for data sharing brought several newer ideas forward while sending the data behind the network. Network file servers allow the local direct attached file system to be visible to client machines attached to the network. Several file systems emerged to distribute data over the network including NFS [28], CIFS [18], AFS/Coda [21, 29] and Sprite [24]. Several companies developed these Network Attached Storage (NAS) appliances, including EMC, Network Appliance, IBM and Sun Microsystems. However, the use of a single file server for a large number of clients raised scalability issues and also presented a single failure point. Significant research work has been done to remove the bottleneck of the single server model. Some of them include storage area networks (SAN), virtual disks [19, 32], network attached storage disks (NASD) [13] and serverless storage systems [14, 4]. Most of these file systems try to improve the performance by reducing the time to satisfy service requests while providing sharing capabilities [15, 16]. Due to the distributed structure of network file servers as against direct attached storage, these file systems have CPU overhead due to protocol handling and are also affected by network bandwidth. Although computing power has improved exponentially, disk bandwidth has struggled to keep up with the pace. The significant improvement in processor performance reduces the concern for CPU overhead. In order to improve data bandwidth, the file systems or storage managers typically stripe the data across the nodes in the storage system. Thus, writes spread the data across the stripe and reads collect the data simultaneously from multiple nodes.

Use of multiple nodes for storage raises concerns for reliability. For increased reliability, a key requirement of these file systems must be the ability to deal with potential storage node failures. With multiple nodes in a cluster, the likelihood of failure of a single node increases. This means that the storage mechanism must be efficient, and also be fault tolerant. Further, the algorithm must not only be able to handle reads, but also address write requests during system failure. 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 [25] techniques such as parity protection or mirroring. However, applying RAID techniques to a clustered storage system introduces several new challenges particularly with regards to performance. Currently, the most common availability solution in clustered storage systems is the use of failover systems which allow functionality to move from failed nodes to other nodes due to shared storage systems. These systems are very expensive and as a result,

the reality is that most of the time, the systems are maintained without redundancy in large clusters.

In this paper, we present algorithms for distributing data reliably and maintain scalability in a clustered file system, in particular for the Lustre cluster file system [6]. We describe techniques used to ensure a functioning system in the face of failures. This paper is organized as follows: we first provide a list of related research work done in Section 2. In Section 3 we give a brief overview of the Lustre cluster file system and storage architecture. Section 4 explains our failure model and describes our algorithms to preserve consistency and data availability. Our results are presented in Section 5. We close with a section on proposed future work and conclusions.

2. Related Work

There are many parallel storage file systems which have been developed recently and are in common use [6, 12, 11, 23, 30, 33, 8]. Most of these file systems are based on the idea of separating the metadata from the data. By separating the metadata, storage management functionalities are kept away from the real data access, thus giving the user direct access to data once the authorization to access the data is received. These file systems can achieve high throughput by striping the data across many storage nodes.

Many commonly used high performance parallel file systems such as Lustre and PVFS have ignored storage node failures but as cluster sizes increase, these failures will become more common and more important. The simplest solution to providing system availability is to use full replication or mirroring, such that data is written to multiple servers in the distributed storage system. Systems that have used replication include Ceph [33], Slice [3], Echo [31], Petal [19] and peer-to-peer file systems such as PAST [10], FarSite [5], Ivy [22], CFS [9] and Pangaea [27]. The problem with replication is the significant overhead costs in terms of storage space. OceanStore [17] uses erasure codes to reduce the storage overhead while providing better availability guarantees at the expense of significant computation overhead.

Using parity based redundancy can reduce the storage overhead, but is very expensive in terms of performance in a networked storage system. Among the earliest parallel storage systems to support parity based redundancy was the Swift/RAID system [20]. Swift/RAID used transfer plans that are specified at the application level to determine RAID levels and striping characteristics. In spite of using optimized data transfer plans they were able to achieve write performance that was only half of the non-RAID performance. Log-structured file systems such as Zebra [14] and xFS [4] can amortize the cost of parity calculation at the expense of garbage collection complications. Pillai et al. have designed a modification of PVFS called CSAR which uses a hybrid RAID scheme that mirrors partial stripe writes on the server [26]. These mirrored data blocks can incur significant storage overhead since they are not cleaned up and the scheme still incurs a 20-25% performance overhead for both large and small writes. Amiri et al. have discussed concurrency strategies when developing RAID protocols for shared storage [2].

3. Lustre File System

In this paper, we concentrate on working with Lustre file system. We chose the Lustre file system over the Parallel Virtual File System (PVFS) because Lustre modules are present inside the Linux kernel and hence uses the VFS cache. This gives Lustre a fair advantage over PVFS. The Lustre file system is based on an object storage architecture as shown in Figure 1. In an object storage architecture, the data transfer protocol between clients and data devices is on an object rather than a block as in a storage area network or a file as in a distributed file system such as NFS. An object comprises the data (or subset of the data) of a file but its allocation is managed by the target device and a metadata server. Each target device can be considered as a smart disk, but the key differentiating feature of an OSD architecture is the metadata server that is separated from the data targets.

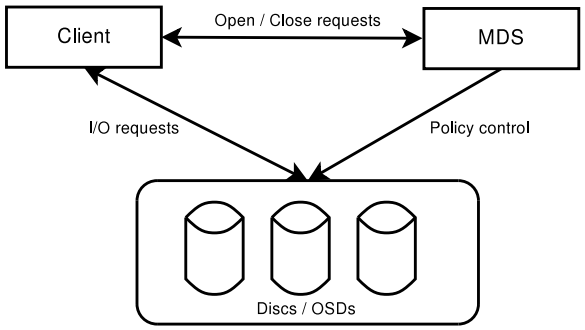

Figure 1. Parallel file system architecture (object-based)

The Lustre file system is basically comprised of three main components – a metadata server (MDS), an object based storage target (OST) and an object storage client (OSC). Any read/write operation on a file has to

be initiated by the client. In order to operate on a file, the client must first obtain the file metadata from the MDS. This metadata information will include the location of the file data, i.e. which objects on which targets, as well as authorization tokens to access those objects. Once this information is acquired, the client need not approach MDS for any subsequent operations. With the metadata separated from the actual data, and with this data striped across several nodes, the Lustre file system can easily obtain very high throughput. However, with this striping of data comes the challenge of keeping the data available. A failure of any node could compromise the complete file system. Lustre does not provide mechanisms to tolerate node failure other than to support for failover at each node. However, implementing failover is a very expensive proposition and not feasible for many small cluster installations.

4. Failure Mode Availability

Lustre makes the system available for write operation also under failure condition, however, the write is performed on the remaining available machines. If the failed node returns, it is not utilized for the writes that were performed while it was down. This means that the system was being under-utilized because of a short failure of access to a node. While this could be acceptable in cases where not many failures occur, in large clusters where there are over 1000 nodes, and where chances of failure is much higher, it is important to have the system utilization to the best.

4.1. Failure Model

Before discussing our algorithms for maintaining data availability, it is useful to explain the failure model assumed. We expect that the the file system should suffer no loss of data under any single failure condition and provide continuous response to client nodes. The potential failures in the system that we consider are transient failures to a single machine which could be because of network failure or a temporary hardware failure. We define these faults as clean errors. Our algorithms are completely clean-error-tolerant. We ideally assume that only one machine will fail during some defined period. Because the failures may be intermittent, the protocols must handle machines temporarily disappearing from the network which may be caused due to a disconnection, or failure of network cable. A disk failure can mean a variety of errors in the low level disk subsystem. Many magnetic media errors will be self-corrected using ECC mechanisms. Other failures of disk hardware or interconnect will manifest themselves as a machine failure, so the same techniques that are used to tolerate machine failures will apply.

As for unclean errors, we mention here the possibility of incorrect data. Examples include network cards flipping bits inadvertently, memory getting corrupted, or processors performing erroneous calculations. For the most part, we assume that these failures do not occur. The reason we do so is that most of these errors are transient and are caught by either automatic error correction methods such as ECC or higher level protocols such as TCP/IP checksums. Incorrect processor calculations are real but extremely rare and so can be effectively ignored.

4.2. Parity

To provide complete availability of the system during single-node failure scenario, the file system must be able to regenerate the data from the remaining nodes. This can be done using long existing RAID techniques. In our work, we have developed a RAID strategy at the target level for the Lustre file system. We have currently implemented a RAID4 mechanism and intend to implement RAID5 in future.

In a naive implementation, one can generate parity in line with the write operations handled by the client. The Lustre file system, which is an object based file system, handles the data in terms of Linux kernel pages due to the limitations of the Linux kernel. Because of this, the complete object is not available to us inside the Lustre file system. Without using a cache maintained within the Lustre file system, generating parity becomes slightly more complicated than it is in a normal RAID system.

While buffering data for a fresh write until all pages from all the objects from a single stripe has been written would be the simplest option, it imposes a heavy hit on the performance of the system. We therefore generate parity with every page that passes through the Lustre file system, and retain just the parity. The page is allocated from within the Lustre file system and is treated as part of the file with a very high page index. Since the parity target is never used for a read/write in normal stream with real data, giving a high page index does not interfere with the object data. A high page index is only given to avoid any clash with any existing page of the same index. The parity page is retained in system memory and used when the page of the same offset within another object from the same stripe is received. The parity is finally sent to the target upon completion of the write operation on that stripe, or on the file. Using this method could however be very costly when doing a

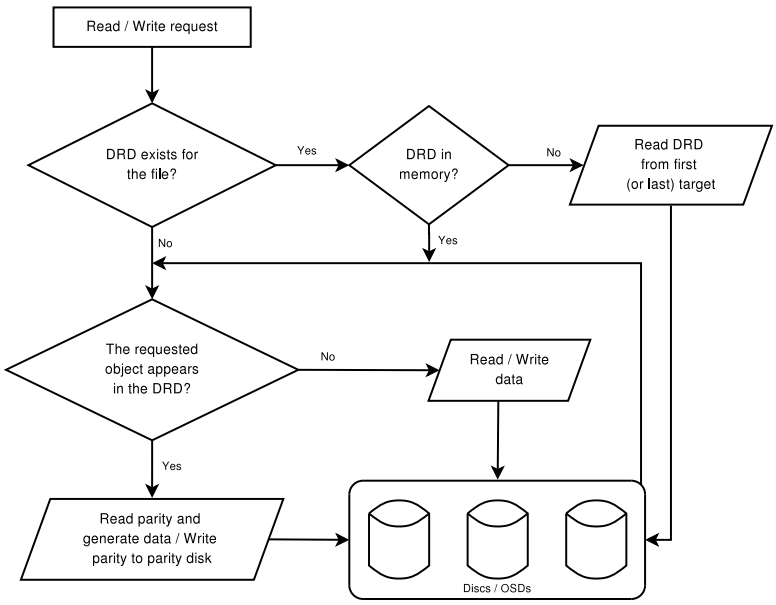

Figure 2. Handling of read/write operation using dirty region database (DRD).

re-write operation. To maintain consistency during re-write, we will have to perform two reads to read the old data and parity and two writes to commit the new data and parity. Experiments show that this step severely impacts the performance of the system as can also be seen in the section 5.

To avoid the costly step of doing the extra two reads and a write during the re-write operation, we have implemented another strategy with the idea of using a thread to handle the operation independently. We introduce a kernel thread running in parallel which would identify and perform the reads and writes required to generate the parity. Since the extra reads are required only when writing a partial stripe, only this data needs to be handled by the thread. A copy of the data is written to the local disk on the client to successfully recover the data in case the target holding the primary copy of the data fails before the thread could complete generating the parity. Information regarding this is also linked with inode structure so that any new reads before the thread completes its operation can be directed to the local disk. At a later point, the thread sends the saved data to the target node using a new macro that has been added into Lustre file system for this operation. With this macro, the target node first reads the original data, generates the intermediate parity using the new data, and then writes the new data. Finally, the intermediate parity is returned to the client node. The client node will then send this intermediate parity information to the parity node where the full parity is calculated and written. By doing this, we save one network data transfer operation. Two read and two write transfers are reduced to just one read and two write data transfers. The read of old data is now done locally on the data target node, and the parity is also generated on the parity target node.

4.3. Writes Under Failure

In the context of a distributed system, writes are more interesting than reads - particularly with parity striped writes. In this section, we discuss methods to ensure that write operations will still succeed even under a single node failure. In most distributed or cluster file systems, if a failure occurs during write, the operation is either blocked, waiting for the failed node to return, or is terminated immediately. In Lustre, when the client observes a failure, before starting the write operation, it automatically reduces the number of nodes, and continues to write with a degraded system not at full storage capacity. Thus, a file which is written during this period would be striped on fewer nodes than originally desired. When a failure occurs during the write operation, Lustre loops its operation waiting for the node to return. This continues until the node returns or the operation times out, ultimately leading to an incomplete write. The user may have to re-issue the write command for Lustre to write it to the remaining nodes this time, or wait for the failed node to return to have access to all nodes.

To handle this situation, and to ensure that the system returns to its full capacity upon return of the failed

node, even for the write which was missed during the node failure, we introduce a database within the file system called *dirty region database (DRD)* similar to the fact server defined in [7]. The DRD contains the information about the data which could not be written to the target during the write operation, but had its parity calculated and stored on the parity target. The DRD also contains information regarding the incomplete stripes which are being written to the local disk and will be transferred to the local node by the kernel thread. We refer to these regions of the file as the *dirty regions*. For any file, information regarding existence of DRD is present as a flag in the file's inode structure. For any subsequent read or write, the Lustre file system first checks if a DRD entry exists for that file. If a DRD entry exist, the client will have to read the DRD from the storage node into the client's local memory. This needs to be done only once, as the DRD entry is locked to that particular process and cannot be modified by any other file I/O thread on the client machine. The DRD will inform the client about the dirty regions present in the file and will automatically transfer read requests for the file's dirty node to the parity node and regenerate the data using parity. Even if the failed node returns to operation, the DRD will let the client know that read requests must be satisfied by regeneration because the data present on the failed node is old and inconsistent. A flow chart showing the handling of read/write operation when using the DRD is shown in Figure 2.

The information of the DRD is stored on the first and last node of the stripe to which the file is being written onto. We store two copies of all DRD data to handle the case of failure of one of the nodes which could be containing the DRD information. A failure of both the nodes would result in a two-node failure scenario and is not handled in this work. The structure is stored as a regular file in a local directory on the target. Since this file is already mirrored on another storage node, local redundancy to protect this file is not required. A special macro to handle this request was added to the Lustre file system. Existence of this information is updated on the MDS using a special flag in the private structure of the inode. This ensures that the client knows if there exists any information in the database and not send any unnecessary requests to the target to check for DRD. Under multi-user scenario, a lock would have to be set on the database to keep it consistent across different client nodes.

4.4 Reads under failure

Reads under failure in our system is very similar to the handling of read under disk failure in a normal RAID system. While doing a read, if the client notices that a node has failed, it will have to regenerate the data by reading the contents of the remaining nodes and also the parity. This operation has to be done from within the Lustre file system and not be visible to the user. The client issues the read requests to the remaining nodes in the stripe, including parity node, for data belonging to that offset within the object of that particular stripe. When all the data is available, the client can simply XOR them to regenerate the original data without the notice of the user. However, if this object had recently been re-written and was currently being used by the thread to generate the parity, the system would read directly from the thread without going to the target. The system can easily find out if this data is available in the thread by checking the DRD.

It could be contended that this scenario under large clusters could cause the DRD to grow to very large sizes, and hence could affect the performance due to long search times. However, since we have a thread running constantly to clear the DRD, the DRD does not typically grow large.

5. Results

5.1. Single Client

For single client tests, we conducted our experiments on 7 Dual Core AMD Opteron dual-processor machines with 2GB RAM and a 80GB SATA disk drive. Five nodes were used as storage targets, one as metadata server and one as the client node. We conducted our tests using the file system benchmarking tool IOZone [1]. The original Lustre file system was tested with 5 targets, with an object size of 64KB for a total stripe size of 320KB. For testing our algorithms using RAID mechanisms, we had 4 target nodes and 1 parity node. We used the same object size of 64KB, thus setting the total stripe size to 256KB.

Our results show that with a very little performance hit, we could achieve a single fault-tolerant storage system. From Figure 3 we can see that calculating parity during the write operation could be very expensive due to the reads involved to generate the parity. We see that throughput for all file sizes at best reaches up to 18% for large file writes of Lustre without parity. However, if we use a separate thread to perform the partial I/O as proposed in this work, we can gain a lot on performance.

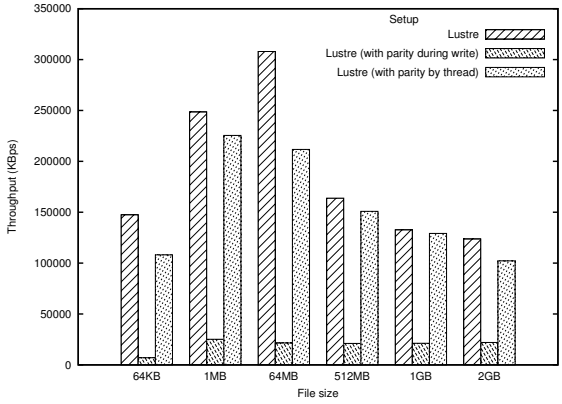

Figure 3. Write with no failure(object size - 64KB).

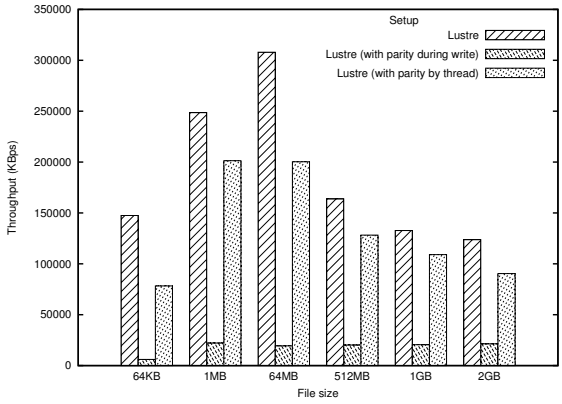

Figure 4. Writes under failure compared to Lustre file system (object size - 64KB).

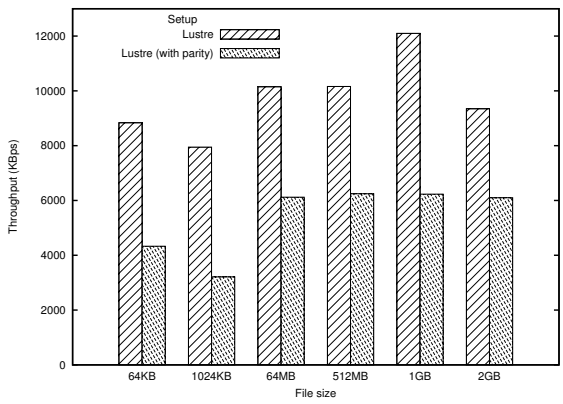

Figure 5. Reads under failure(object size - 64KB).

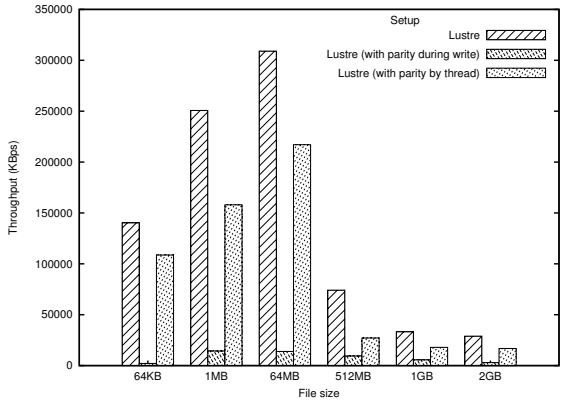

Figure 6. Write performance with 8 clients (object size - 64KB).

The parity performance in this technique ranges from 68-97% of normal Lustre. The thread which handles the writing of partial stripes and updating parity, flushes out data continuously. But since this is done using an independent thread, an impact of this is not visible on write. On average, the degradation in performance due to using parity is about 15%. The overall load on the system due to the thread was minimal because the parity was generated on the storage node.

For testing under a failure scenario for read or for write, we modeled the failure condition of a target by forcing the client to view a random target as a failed node. Figure 4 shows the performance of writes under a single node failure scenario compared to the normal Lustre file system performance. It can be seen that under failure, the writes do not differ much from no failure condition. It can also be observed that the use of the thread does improve reconstruction time significantly. The little loss of performance is due to writing of the DRD which is done only under failure cases. In terms of read under failure condition, we observe regenerating data using parity could hit us on performance in the range of 40-65% of a read when there is no failure, which is not bad since we were able to maintain availability of the system despite failure of one node. We can gain some performance benefit on read under failure by doing a read ahead of parity data.

5.2. Multiple Clients

We conducted another test to monitor the performance of the system with 8 clients doing simultaneous

I/Os on the same set of target nodes. All machines used for the test were dual core AMD Opteron dual-processor with 2GB RAM. 8 of them were used as clients, one as the metadata server and 5 of them as storage nodes.

Figure 6 shows the write performance of our algorithm as compared to the original Lustre performance when put under stress by 8 clients simultaneously on different files. We ran the benchmarking tool IOZone on each client for independent files. From Figure 6, we can observe that with multiple clients, write performance suffers more with parity because the data which under normal Lustre was distributed among 5 nodes by 8 clients is now being distributed to just 4 nodes. This implies that there would be more load on each target node, and hence relatively poorer performance. Using a RAID5 scheme could possibly improve this performance.

6. Future Work

We plan to include several performance improvements to our work. We intend to implement a two-phase commit protocol for committing data on the target nodes, particularly for parity information. If a client node fails while the parity is being written, and before the node data was committed, the system would result in an inconsistent state. We could avoid this by using a two-phase commit for parity and data. That means, the parity would only be committed after it is guaranteed that the data was committed to the disk on the other target nodes.

We also plan to add the feature of regenerating data on the run. Thus, if there had been a failure and an entry in the DRD was made, and if we receive a read or re-write request on the same portion of file, we could regenerate the data using the parity and write it to the storage nodes and clean up the DRD. Further, using the DRD with multiple clients while maintaining consistency across all clients need to be implemented.

7. 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 transient failures. The protocols are designed with fault-tolerance as a principal design commitment rather than as an afterthought. The main contribution is the notion of storing data locally to a client as well as to the cluster in order to reduce the impact of parity generation on writes. We have shown that this improves performance significantly.

8. Acknowledgements

This work was supported in part by the National Science Foundation under award number CCF-0621448 (for research support by NSF HECURA). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

References

[1] Iozone file-system benchmark, http://www.iozone.org.

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

[3] D. Anderson, J. Chase, and A. M. Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Oct. 2000.

[4] 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.

[5] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 34–43, June 2000.

[6] P. J. Braam and R. Zahir. Lustre technical project summary. Technical report, Cluster File Systems, Inc., Mountain View, CA, July 2001.

[7] J. D. Bright and J. A. Chandy. Data integrity in a distributed storage system. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 688–694, June 2003.

[8] 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.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Oct 2001.

[10] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HOTOS VIII*, May 2001.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SOSP'03: Proceedings of the nineteenth ACM symposium on Operating System Principles*, pages 29–43, 2003.

[12] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Harding, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

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

[14] J. H. Hartman and J. K. Ousterhout. Zebra: A striped network file system. In *Proceedings of the USENIX 1992 Workshop on File Systems*, May 1992.

[15] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[16] J. E. Johnson and W. A. Laing. Overview of the spiralog file system. *Digital Tech. J.*, 8(2):5–14, 1996.

[17] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[18] 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.

[19] 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.

[20] D. D. Long, B. R. Montague, and L. Cabrera. Swift/RAID: A distributed RAID system. *Computing Systems*, 1994.

[21] 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.

[22] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[23] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale Storage Cluster: Delivering scalable high bandwidth storage. *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC'04)*, page 53, Nov 2004.

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

[25] 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 Management of Data*, pages 109–116, June 1988.

[26] M. Pillai and M. Lauria. A high performance redundancy scheme for cluster file systems. *International Journal of High Performance Computing and Networking*, 2(2/3/4), 2004.

[27] Y. Saito, C. Karamonolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system, December 2002.

[28] 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.

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

[30] 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*, pages 231–244, Jan. 2002.

[31] G. Swart, A. Birrell, A. Hisgen, and T. Mann. Availability in the echo file system. *Research Report, Systems Research Center*, September 10 1993.

[32] 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.

[33] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA*, page 22, Nov. 2006.