

A Mechanism for Scalable Redundancy in Parallel File Systems

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

by
Bradley W. Settlemyer

May 2006

Advisor: Walter B. Ligon, III

ABSTRACT

As parallel file systems span larger and larger numbers of nodes in order to provide the performance and scalability necessary for modern cluster applications, the need for fault-tolerance and high data availability file systems has arisen. Modern parallel file systems spanning tens, hundreds, or even thousands of servers will require fault tolerance to avoid job failure and catastrophic data loss due to a single disk failure or server loss. Effective fault tolerance in parallel file systems must provide a high degree of data resiliency, consistency, and scalable performance.

In this thesis we provide an in depth description of the resiliency and consistency requirements of parallel file systems. We then describe a data replication mechanism that meets the resiliency and consistency requirements of parallel file systems and provides scalable performance. We also provide an in depth description of how the file system responds during a system fault and how the system may be recovered to its original, fully redundant state after a failure. Finally, we measure the performance of our proposed mechanism by implementing it in a popular parallel file system, PVFS2. We primarily focus on measuring the performance costs and scalability impacts associated with consistency and resiliency.

ACKNOWLEDGMENTS

First, I would like to thank Dr. Ligon (Walt) and Dr. Ross (Rob) for their support and guidance during the development of my Master's thesis. I would also like to thank Dr. Schalkoff and Dr. Taha for appearing on my committee and providing feedback on my thesis. I would like to especially thank Dr. Carns (Phil) and Dr. Jones (Will) for greatly aiding me in developing and presenting my work. Finally, I gratefully acknowledge use of "Jazz" and "Chiba City," clusters operated by the Mathematics and Computer Science Division at Argonne National Laboratory as part of its Laboratory Computing Resource Center.

TABLE OF CONTENTS

	Page
TITLE PAGE	i
ABSTRACT	ii
ACKNOWLEDGMENTS	iii
LIST OF TABLES	vi
LIST OF FIGURES	vii
CHAPTER	
1 INTRODUCTION	1
1.1 Parallel File System Fault Tolerance	4
1.2 Proposed Research and Goals	7
1.2.1 Data Resiliency	7
1.2.2 File Consistency	8
1.2.3 Performance	8
1.3 Methodology	9
2 RELATED WORK	11
2.1 Fundamental Fault Tolerance Techniques	11
2.2 Fault Tolerance in Cluster File Systems	13
3 ANALYSIS AND DESIGN OF A DATA REPLICATION PROTOCOL	15
3.1 The Fundamentals of Replicating Data	16
3.1.1 Data Distribution	17
3.1.2 Communication Mechanism	19
3.2 Replicated File System Data Consistency	21
3.2.1 Ensuring Data Consistency on Writes	22
3.2.2 Ensuring Consistency During Failures	24
3.3 Failure Scenarios	27
3.3.1 Standalone State	27
3.3.2 Voting and Quorums	28
3.3.3 Failure Protocol Use Cases	29
3.4 Recovery Protocol	36

Table of Contents (Continued)

	Page
4 IMPLEMENTATION AND PERFORMANCE	38
4.1 Experiment Configurations	38
4.2 File System Write Modifications	40
4.3 Write Performance	45
4.4 Read Performance	49
5 CONCLUSION	52
5.1 Contributions	53
5.2 Future Work	54
BIBLIOGRAPHY	56

LIST OF TABLES

Table	Page
4.1 Relative replication performance on Adenine with 8 I/O servers	46
4.2 Relative replication performance on Adenine with 16 I/O servers	46
4.3 Performance of a replicated and non-replicated file system on Adenine	47
4.4 Relative replication performance on Jazz with 16 I/O servers	48

LIST OF FIGURES

Figure	Page
1.1 NFS Network Configuration	2
1.2 Parallel File System Network Configuration	3
1.3 A Single RAID-5 System	5
1.4 A Cluster of RAID-5 Systems	6
3.1 Data Distribution for a Parallel File System	17
3.2 A Simple Data Distribution for a Replicated Parallel File System	18
3.3 An Alternative Data Distribution for a Replicated Parallel File System	18
3.4 Sequence Diagram for Writes	26
3.5 Sequence Diagram for Reads	27
4.1 Data Distribution Providing Replication in Experiments	39
4.2 Comparison of aggregate write bandwidth with multi-threaded writes on Adenine w/ 8 IO Nodes	42
4.3 Comparison of aggregate read bandwidth with multi-threaded writes on Adenine w/ 8 IO Nodes	42
4.4 Comparison of aggregate write bandwidth with multi-threaded writes on Adenine w/ 16 IO Nodes	43
4.5 Comparison of aggregate read bandwidth with multi-threaded writes on Adenine w/ 16 IO Nodes	43
4.6 Comparison of aggregate read bandwidth with multi-threaded writes on Jazz w/ 16 IO Nodes	44
4.7 Comparison of aggregate write bandwidth with multi-threaded writes on Jazz w/ 16 IO Nodes	45
4.8 Performance on Adenine w/ 8 IO Nodes	46
4.9 Performance on Adenine w/ 16 IO Nodes	47
4.10 Performance on Jazz w/ 16 IO Nodes	49
4.11 Performance on Adenine w/ 8 IO Nodes	50

List of Figures (Continued)

Figure	Page
4.12 Performance on Adenine w/ 16 IO Nodes	51
4.13 Performance on Jazz w/ 16 IO Nodes	51

CHAPTER 1

INTRODUCTION

The steady performance increase of commodity processors [9] and the construction of cluster computers composed of larger and larger numbers of nodes has led to a considerable increase in computational power. The Top 500, a list of the 500 fastest supercomputer systems, lists the two newest and fastest supercomputers as containing 32,768 processors and 10,160 processors respectively (as of November 2004) [24]. The number of processors employed in these computers is much greater than the 4,192 processors used to build the previous year's fastest supercomputer [23]. Given that the fastest machine is 45 percent faster (as measured in Teraflops) than the older machine, it appears likely that forthcoming systems will have as many processors or more than even these large machines. The trend in supercomputing toward large numbers of processors may achieve Petaflops ratings; however, without improvements in I/O performance and fault tolerance, I/O intensive scientific applications will not be able to leverage the processing power inherent in these systems.

Many types of scientific applications frequently need to access large datasets, and further, the application performance is primarily limited by I/O performance rather than processing power [26]. The I/O throughput performance of modern clusters has not improved at the same rate as the processing power because main memory, secondary storage, and networking components have not increased in speed as rapidly as CPU performance. The resulting I/O bottleneck for many applications is due to the performance disparity between the I/O components and CPUs [14].

Although individual I/O components are not increasing in performance as quickly as modern processors, scalable systems such as parallel file systems can be employed to achieve high performance by utilizing many slower I/O components simultaneously to achieve high aggregate performance [10]. A parallel file system stripes the contents of

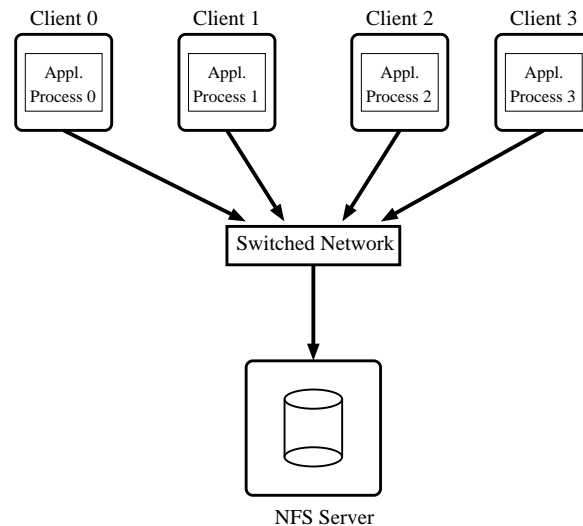


Figure 1.1: NFS Network Configuration

each file across many independent storage servers so that file data is accessed from many storage servers in parallel. The individual storage servers are typically called *I/O nodes* as opposed to the cluster nodes intended for application execution called *computation nodes*. The improved parallelism leads to greater available bisection bandwidth and higher aggregate throughput.

Parallel file systems are similar to traditional network file systems in that they allow many cluster nodes to mount a single file system over the cluster interconnection network and transparently access file data over the network connection. However, as shown in Figure 1.1, network file systems traditionally do not provide support for striping data over multiple I/O nodes. A scientific application running on multiple computation nodes in parallel is able to see a single consistent view of the file system while still achieving the high aggregate bandwidth achieved by distributing the file contents over independently accessible I/O nodes (Figure 1.2).

When incorporating a large number of components into a single system or service, such as a parallel file system, the probability of a single component failure due to a hardware fault grows enormously. Though the mean time to failure (MTTF) for a processor or I/O component may be very high, the large number of components deployed will inevitably

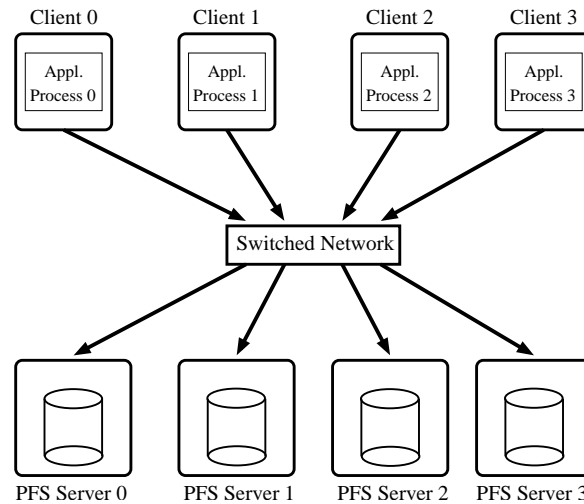


Figure 1.2: Parallel File System Network Configuration

lead to frequent failures [28]. Current processors and computer system components exhibit long lifetimes and infrequent failure rates. For example, some highly reliable power supplies, storage drives, and controller chips are rated at a MTTF of 1,000,000 hours (110 years). However, even with such reliable components, a cluster composed of thousands of reliable components exhibits a much higher overall failure rate. Using an exponential random variable to model component failures, and a MTTF of 10^6 hours for each individual component, a system utilizing 10,000 very reliable components would have a system wide MTTF of only 100 hours.

In addition to hardware failures, computational jobs and services running on a cluster may fail or need to be halted for other reasons. Emergency system maintenance due to a security update may lead to jobs being halted and restarted. System software may fail due to a full disk partition or simply a software bug. In some environments it may be possible to simply restart the application from the beginning and attempt to run the job to completion. However, many scientific applications have run times of several days, and in those cases it isn't desirable to expend highly sought after cluster compute cycles performing duplicate work by re-running large computational jobs.

1.1 Parallel File System Fault Tolerance

Clearly, the growing number of cluster components being simultaneously leveraged demands a high degree of fault tolerance in order to provide the necessary availability and performance for user applications. Ideally, all of the cluster system components would be hardware redundant and support some type of software fault tolerance; the failure of a single hardware or software component would minimally impact service. Unfortunately, the technology to achieve full cluster redundancy does not exist, and many existing fault tolerance techniques are prohibitively expensive or impact performance in such a way as to be unusable in high performance computing.

Parallel file systems pose several fault tolerance problems not addressed by existing fault tolerance mechanisms. Since files are striped across many I/O nodes, every file in the file system is stored across multiple independent machines. If no fault tolerance is present, the failure of a single node will guarantee that any file data stored on that node is irretrievable, and the file cannot be made intact and available until the failed node or node data is brought back on line.

Traditional file system fault tolerance has typically been provided as hardware redundancy at the secondary storage level. The deployment of RAID [7], or redundant arrays of inexpensive disks, is common in many data critical environments. The simplest RAID configuration, RAID-1, ensures that all data is written to both a primary and redundant disk so that there are two copies of the information. If one of the disks experiences a failure, the mirrored disk can then be used exclusively with no service interruption or data loss, and the failed disk can be replaced during a later scheduled maintenance. Another popular approach, RAID-5, constructs parity information for the data stored on each disk, and then distributes the parity information across the remaining disks. The storage of parity information (often computed with Hamming codes or Reed-Solomon Codes) reduces the size of the mirrored data, however it does incur a runtime penalty in that the parity must

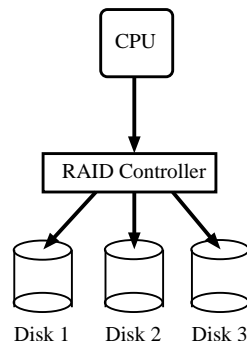


Figure 1.3: A Single RAID-5 System

be calculated for every write, and on failure the original data must be computed using the parity information.

In all hardware-based RAID schemes, the distribution of data among the disks is managed by a single RAID-enabled disk controller (Figure 1.3). Although the controller can be made redundant, there must be only a single controller active and it must be directly attached to all the disks. Implicitly, RAID can only be used for a single physical storage system, i.e. a single RAID controller. The massive clusters we described earlier are composed of thousands of separate computer systems each with its own storage and storage controller. In a distributed environment where each node has its own independent storage controller and independent control of its storage (necessary for any virtual memory-based operating system) a centralized hardware RAID controller does not have physical access to all the storage media, and nor could it provide the required I/O throughput of parallel I/O systems.

One solution to this approach would be to simply implement RAID controllers at each I/O node as shown in Figure 1.4. This configuration effectively makes the system tolerant to a disk failure; however, as the nodes are complete and independent computer systems, an I/O node can fail for reasons completely unrelated to its storage system. Switch malfunctions, network partitioning, and cooling system failures are all likely possibilities of node failure that a RAID system per I/O node cannot handle. The distributed nature of

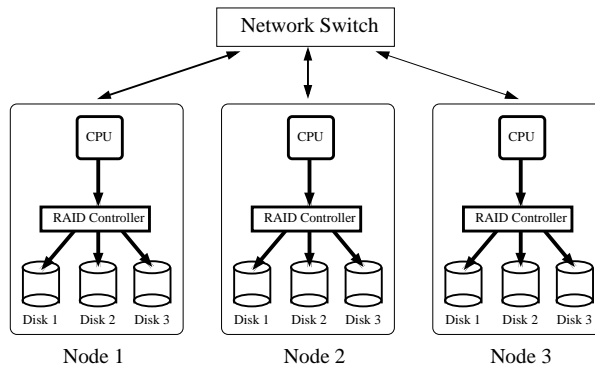


Figure 1.4: A Cluster of RAID-5 Systems

Beowulf clusters make hardware-based redundancy virtually impossible, instead we must look at software-based approaches that provide fault tolerance to an entire node failure rather than just a single physical component within the node.

Another fault tolerance possibility is the use of software RAID and network block devices. A network block device is a storage device that is connected to a computer system by a network connection. The operating system can then enable any storage device located in one machine to be accessed as a network block device by another machine. In some environments this mechanism may be appropriate; however, cluster environments typically use interconnection networks based on sending and receiving packetized data or remote memory access [22]. The networks are optimized for message passing rather than sending disk requests so the transfer of raw block data may not perform optimally on cluster interconnection networks. Furthermore, scientific applications often perform complicated, strided file access patterns [19], and it is not clear that software RAID and network block devices can easily be tuned to provide high performance for such applications. That is, the operating system level block data representation is not designed to describe non-contiguous file regions, whereas parallel file system request structures are explicitly designed to efficiently represent non-contiguous file access patterns.

1.2 Proposed Research and Goals

Existing RAID mechanisms are only able to provide intra-node disk failure support. RAID techniques cannot provide fault tolerance or scalability support for the I/O node failures that would make a parallel file system unavailable. In order to develop an appropriate fault tolerance mechanism for parallel file systems, it is critical to analyze data resiliency, file consistency and performance for any candidate approach. Using those criteria as the basis for judging the effectiveness of a fault tolerance mechanism, *we propose that implementing data redundancy as a server-to-server replication mechanism within the file system is an effective method to achieve fault tolerance for parallel file systems on cluster computers.*

1.2.1 Data Resiliency

Resilient data is data that has been stored so that the occurrence of hardware or software faults will not lead to the loss of data or the loss of data services. The technique of replicating data to create a mirrored copy of each basic data block is well understood and offers high availability and resiliency guarantees. Traditional RAID approaches to replication may not be an appropriate fault tolerance mechanism for parallel file systems, but the use of data replication is a powerful technique.

By implementing replication within the parallel file system we are able to achieve several benefits. First, replication ensures that a redundant copy of all file data exists, and the file system can experience an I/O node failure without losing any data. Additionally, it is possible to implement replication in a fashion where the failure of an I/O node does not lead to any interruption of service by developing an online fail-over protocol. By implementing replication within the file system it will also be possible to leverage the existing file system facilities for providing data transfer and access to storage.

1.2.2 File Consistency

The design of a fault tolerance mechanism for parallel file systems must ensure that the file system provides adequate data consistency guarantees. A scientific application running as thousands of processes needs to have simultaneous access to a consistent, or identical, view of the file system for each of the computation nodes. Parallel applications accessing parallel file systems cause significant difficulty in maintaining file consistency because of the probability of many clients performing simultaneous reads and writes to regions of the same file. Furthermore, the primary and mirror copy of the data must be consistent after each successful write to ensure that an I/O node fault does not cause the file system contents to change from the perspective of client processes.

1.2.3 Performance

Finally, we must ensure that the additional latency and bandwidth costs of data redundancy do not overly interfere with file system performance and scalability. The existence of efficient, scalable network protocols for parallel file systems [5] ensures that the additional delay and bandwidth requirements imposed by data replication will be due to additional resource usage rather than an “impedance mismatch” between the network and the storage device. Scalability is one of the most critical aspects of parallel file system design. As the number of I/O servers is increased, the file system must achieve higher aggregate bandwidth with little additional latency. One mechanism to achieve greater scalability in file systems is the use of intelligent servers [6], or I/O servers that communicate with each other to provide better file system support. By implementing our replication scheme as an intelligent server extension that performs inter-server communication to achieve replication, it is possible to achieve scalable performance while performing replication and fault tolerance.

1.3 Methodology

In order to explore how a replication protocol can provide effective fault tolerance for a parallel file system, we first determine what data resiliency, data consistency, and performance requirements exist for a parallel file system. We then analyze the methods a replication protocol can employ to achieve a high degree of resiliency, consistency, and performance. As we will show, sometimes these three objectives are incompatible. In these cases we evaluate the possible alternatives and choose the design details that are appropriate for the workloads often encountered by parallel file systems. After analyzing the system requirements and describing the design of our replication protocol, we provide an exhaustive list of all the failure scenarios that may occur in our protocol and a description of how the system responds in each of these scenarios. We also provide a description of a simple recovery protocol.

In order to measure the performance of our protocol in non-failure scenarios, we have implemented the protocol in PVFS2. PVFS2, the Parallel Virtual File System 2 [27], is a joint development effort of Clemson University and Argonne National Laboratory and serves as both a test bed for research in parallel I/O and a fully functional parallel file system. The file system's high performance and modular design makes it an excellent platform to use for adding data redundancy extensions. Our quantitative analysis of the performance impacts of data replication file access times are compared under several scenarios. Performance is measured using both high performance cluster interconnects and typical commodity network hardware, both of which are common in many cluster environments. By measuring performance with both types of hardware we are able to gain valuable insight into performance in the case where the network is the performance limiting factor, and when the speed of secondary storage is the limiting factor.

In Chapter 2 we describe other research projects that address fault tolerance for distributed systems and parallel file systems. In Chapter 3 we describe the fault tolerance requirements in depth, and describe how our scheme addresses those requirements. In

Chapter 4 we explore our implementation of parallel file system fault tolerance and examine its performance characteristics. Finally, in Chapter 5 we summarize our work results and identify several future research possibilities related to parallel file system fault tolerance.

CHAPTER 2

RELATED WORK

The development of fault tolerant mechanisms for data storage systems has been a popular topic for researchers in database systems, distributed systems, and file systems. The fundamental techniques used to implement fault tolerance in all of these fields are very similar; however, the specific details of each technique need to be modified for the individual problem domains. Here we will cover the techniques that most influenced our mechanism for data redundancy for parallel file systems. We will then describe fault tolerance mechanisms used in other network and parallel file systems, and how and why they differ from the mechanism described in this thesis.

2.1 Fundamental Fault Tolerance Techniques

The basis of all software-based redundancy mechanisms is data replication. The primary-copy technique was the first replication technique to emphasize both consistency and resiliency [1]. Initially designed to provide distributed resource sharing, the advantages of this protocol have led to its adoption in many fault tolerance schemes. In a two server setting, one of the servers is designated as the primary, while the other server is designated as the backup (the protocol has also been generalized to allow more than one backup server). All update requests are directed to the primary server, which forwards the request to the backup. If the backup successfully completes the update, it sends an acknowledgment to the primary, which then also performs the update and signals success to the client. In the case that the backup server fails, the primary server only performs updates locally until the backup can be recovered. In the case where the primary server fails, the backup server assumes the role of primary and only performs local updates until another backup can be created. The strong consistency and resiliency of this approach is very attractive for a

distributed database management system; however, a parallel file system does not need to implement journalized transactions capable of rollback. Parallel file systems do not need to support the notion of atomic updates, meaning that if an update to the file system is canceled during the middle of writing, the file system does not need to be reverted to an earlier state. Because of the less strenuous consistency requirements, we were able to modify the primary-copy approach to lower latency, while still following its general protocol description.

The other major replication approach is based on majority consensus, known interchangeably as a voting [30] or quorum system [12]. These replication algorithms are notable for their distributed control and high degree of fault tolerance. In the basic system, several servers exist that maintain a copy of the data. On every update request, the participating servers vote on whether to commit the update, and then apply the update to all available copies if enough servers participated in the vote. Similarly, on reads the servers vote on which distributed copies contain the correct results, and one of the correct copies then returns the result. Although the lack of emphasis on deterministic performance and scalability is inappropriate as a primary replication mechanism, we do employ voting as a scheme to safely determine which nodes are failed. The high degree of distributed control and fault tolerance available in consensus-based replication makes it an excellent scheme for making distributed decisions.

Some of the most recent work in replication techniques has extended the above consensus approaches to handle Byzantine faults [2]. Byzantine faults are created when one of the participating servers, due to a failure or security compromise, sends out invalid or incorrect responses to the remainder of the distributed systems. We anticipate that Byzantine faults may be frequent in clusters due to network partitions and frequent failures, and for that reason our consistency scheme is fault tolerant for many typical Byzantine failures.

Finally, our approach to ensuring sequential consistency of replicated file data is strongly influenced by the Harp file system [21]. Harp used a modified version of the

primary-copy technique called viewstamped replication to ensure that multiple copies of data are identical. Viewstamped replication uses a timestamp ordering approach to ensure that data is equally up to date. Although the mechanism assumes that global clocks exist and can be accessed by distributed machines (a mechanism that is not readily available in modern cluster environments), it does illustrate that the primary-copy approach can be modified to use an optimistic concurrency control. Our optimistic concurrency control achieves the same goals as viewstamped replication, namely, reduced latency compared to two-phase locking, and our protocol can be implemented on existing cluster hardware.

2.2 Fault Tolerance in Cluster File Systems

One of the best documented fault tolerant parallel file systems is the Google File System [11]. A modified primary-copy replication technique is used with a single primary server called a master and two layers of backup servers called primary replicas and secondary replicas. The existence of multiple primary and secondary replica servers is critical to achieving performance in the application environment. The Google File System also does not guarantee that all replicas are kept in a consistent, or identical state. Instead the client application is responsible for handling inconsistent file regions, typically by doing nothing since data is typically appended to the end of files rather than overwritten. Because the Google File System is so finely tuned for appends, rather than general reads and writes, it is not an appropriate approach to a general parallel file system.

The River environment [3] uses a modified form of chained declustering [15] to achieve data mirroring. Graduated declustering implements the basic primary-copy technique, however the primary and backup are no longer single servers. A collection of nodes, each with an entire copy of the data, acts as the primary and another collection of nodes acts as the backup. The existence of multiple servers acting as the primary allows file region accesses to be load balanced across all the nodes, however file writes will still experience

the delay associated with the primary-copy technique further multiplied by the number of disks participating in the primary and backup disk clusters.

GPFS [29], the general parallel file system, provides support for software-based data replication and online fail over. All write requests first obtain a lock token, and then the client sends the data to two separate storage nodes. Because GPFS uses client-based data replication (i.e. the data is sent directly from the client to 2 separate storage nodes), it must use a distributed locking subsystem with support for two-phase locks and lock timeouts to achieve sequential consistency. Our mechanism has rejected the use of distributed locking schemes because of the inherent complexity, and difficulties in achieving predictable, scalable performance. In conjunction with the locking subsystem, GPFS uses a heartbeat system to detect storage node failures. Heartbeat systems (particularly those without dedicated hardware support) also exhibit considerable complexity in determining the timeout threshold for heartbeat messages and were rejected for use in our protocol due to complexity of such systems.

CEFT-PVFS [31] also uses a heartbeat system to detect failures and a centralized lock server with two-phase locking to ensure that writes are properly serialized. CEFT-PVFS is very similar to GPFS in that it requires a locking subsystem and heartbeat monitoring system to provide consistency and resiliency. The mechanism described in this thesis attempts to avoid such complicated infrastructure. The authors of CEFT-PVFS have also contributed a reliability model that allows some degree of quantitative comparison between synchronous and asynchronous replication.

The Seneca protocol [17] also attempts to define a data redundancy protocol for cluster file systems. However, it does not promise that the file system is consistent under certain failure scenarios. Our protocol guarantees consistency in all non-byzantine failure scenarios.

CHAPTER 3

ANALYSIS AND DESIGN OF A DATA REPLICATION PROTOCOL

Typical faults that impact parallel file systems include hardware and disk failures, network outages, intermittent network faults/partitioning, or unplanned power outages that lead to corrupt file system data. Exactly which types of errors and how many of those errors the system can withstand are important considerations when building a redundancy mechanism for parallel file systems.

In general, we expect all redundancy schemes to provide data resiliency in the face of these faults. Data replication schemes in particular have the opportunity to provide excellent data resiliency. In the simplest case, where every strip of file data is stored on two independent storage nodes, a replication mechanism can absorb failure of 1/2 the I/O nodes and still provide a complete, consistent view of the file data. In the case where node failure probabilities are independent, the expected number of failures that can occur before data loss is 1/4 of the I/O nodes. Of course, in order to ensure data availability we expect that system administrators will intervene and replace failed I/O nodes. However, in truly large systems it may be a requirement to absorb more than one simultaneous failure.

Another critical factor in the design of a data replication system is the level of consistency maintained for the file data. In the case of an unexpected fault, is it possible to ensure that the file system data will appear identical to the user both before and after the fault? If the I/O node notifies the client of the write completion before the data has been replicated, any failure before the data replication is complete may lead to a change in the client's file system view. Our protocol maintains file data consistency for most types of file system faults.

Alternatively, conservative locking systems can be used to enforce consistency semantics that ensure that replicated data is perfectly synchronized. In this case both repli-

cated copies of the data are locked until the modification of both is complete. This locking is not in the classic mutual exclusion sense, but rather as an implementation of a pseudo-transaction for a file system. Conservative locking systems also impose significant overhead in terms of latency and complexity (e.g. timeout mechanisms, stale state detection, etc.) [18]. To avoid the difficulties of this system, we propose a system that guarantees sequential consistency without locking.

Finally, a data replication system for parallel file systems must provide high performance. The difference between performance efficiency and scalability can be subtle. Performance efficiency is the idea that a system provides a high degree of total performance for a set of components. For example, a disk that was ten times faster than any other disk in existence would provide very efficient performance. Scalable performance is the notion that as computing resources are added, system performance improves. In order to achieve significant scalability, a system will often need to exhibit a high degree of performance efficiency. In our protocol, we have provided a highly efficient implementation, and while the raw performance is lowered due to the increased bandwidth costs of replication, the scalability of our protocol is very similar to the scalability of the non-replicated reference implementation.

3.1 The Fundamentals of Replicating Data

Fundamentally, a data replication technique must manage to duplicate data to at least two independent, persistent storage devices. Whether for a parallel file system, a database, or a distributed object system, all data updates must be written twice. Clearly, in the case of a parallel file system we will strive to write the additional data in parallel to whatever degree possible; the expected performance impact of replication is a doubling in the bandwidth cost of each write (both in terms of network bandwidth and disk bandwidth). Assuming the available bandwidth on a cluster is the same, whether using replication or not, a doubled bandwidth cost will translate into at least a doubled latency. Data reads are not similarly

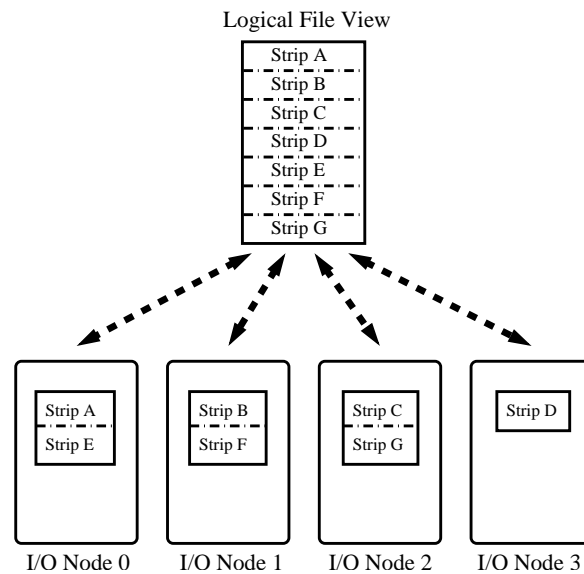


Figure 3.1: Data Distribution for a Parallel File System

impacted in terms of increased bandwidth costs. For reads we do not need to transfer the same data twice, and further, we can use the replicated copies of the data to maintain the same level of parallelism available in a non-replicated parallel file system. The only additional overhead related to reads and writes is ensuring that the replicated data is kept consistent, a step not necessary in a non-replicated parallel file system.

3.1.1 Data Distribution

In order to achieve greater aggregate bandwidth, parallel file systems stripe file data across several I/O nodes. This enables file system clients to access all the I/O nodes in parallel, reducing the amount of time required to perform reads and writes of file data. The *data distribution* describes how the logical file is mapped onto physical storage. For example, a simple striping distribution is shown in Figure 3.1. The individual components of the file data stored on each individual disk are called *data objects*. A file in a parallel file system is composed of all of the data objects for the file and the file metadata, which is typically stored in a structure and location separate from the file data.

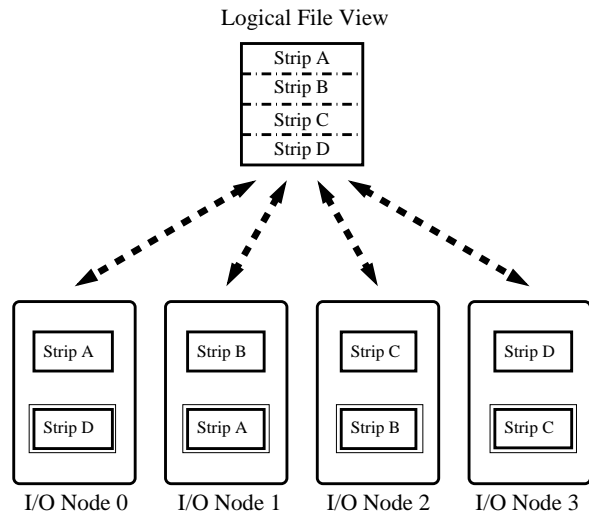


Figure 3.2: A Simple Data Distribution for a Replicated Parallel File System

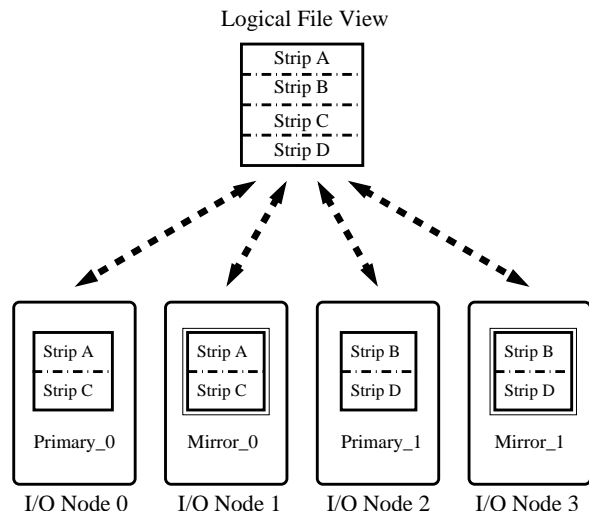


Figure 3.3: An Alternative Data Distribution for a Replicated Parallel File System

Figures 3.2 and 3.3 show possible data distributions for a replicated file system. The replicated data objects, or secondary copies, are shown with a double outline. In both distributions, each data object is stored on two separate and independent I/O nodes. For example, in the first distribution, strip B is stored on nodes 1 and node 2. In the second data distribution, file strip B is stored on nodes 3 and 4. The first distribution scheme doubles the total number of data objects and ensures that data is duplicated to data objects residing on different servers. In the second scheme, the I/O nodes are partitioned into exact copies of one another, with node 1 mirroring the contents of node 0 and node 3 mirroring the contents of node 2.

The varied data distributions possible allow individual implementations of our technique to modify the replicated data mapping for specific cluster environments and applications. Due to the ease of implementing the approach shown in Figure 3.2 in PVFS2, it is this replicated data distribution that we will be using for performance evaluation in Chapter 4. In order to create the replicated data distribution, we create one additional data object per I/O node, doubling the number of data objects per file. In the next section we will describe how data is duplicated and written for the additional data objects.

3.1.2 Communication Mechanism

To replicate data, it is necessary to develop a robust scheme for transmitting multiple copies of file data to independent I/O nodes in a safe, efficient manner.

The simplest way to achieve file replication is to modify the file system client so that instead of sending one copy of the data to an I/O server, the clients send identical data to two I/O nodes. One problem with this approach is that the client processes are now responsible for ensuring that both file requests are identically applied to the file system. For example, if the client attempts to write 100 bytes to a file, the 100 bytes will need to be written to two independent I/O nodes. If one of the write requests completes immediately, there is a danger that an error may occur before the write is committed to the second I/O server. The

severity of this problem becomes evident when we consider that client processes are often terminated during execution due to user or system interactions (e.g. a parallel job may be terminated prematurely if it executes longer than the estimated run time). We will see later that early client termination also makes it difficult to guarantee data consistency in many failure scenarios.

Another approach to replicating data to independent I/O nodes is the use of server to server communication. In a server to server communication scheme, the client sends only one request to an I/O server. The server then performs the file system modification and forwards the request to another independent I/O node to achieve data replication. Server to server communication approaches have the advantage of not being dependent on the client to achieve replication. In typical computing environments where client processes may be aborted by the user (or may simply be less reliable than a server process), server to server communication offers advantages in reliability.

Another key factor in comparing client-based replication and server to server communication is the possible differences in the architecture of compute nodes and I/O nodes. Client-based replication doubles the network bandwidth requirements for the outbound client link and the inbound server link (on file writes). A server to server approach still doubles the network bandwidth requirements of the inbound server link, but instead of affecting the client's outbound network connection, the server's idle outbound link is used instead. For many clusters this configuration is preferred because the much smaller number of I/O nodes can have their network infrastructure more easily enhanced rather than the more numerous computation nodes. For example, on the Blue Gene/L systems that populate many of the first 100 slots on the Top 500 List, the ratio of computation nodes to storage nodes ranges from 8/1 to 64/1 [8]. Additionally, computation nodes are more likely to have very low latency message passage networks, whereas I/O nodes are bandwidth constrained and may be architected with higher bandwidth networking infrastructure.

Server to server communication provides greater resiliency to client failures and improves architectural flexibility for designing cluster interconnection networks. Additionally, in the next section we will show that it is easier to ensure data consistency using server to server communication. For these reasons, we have chosen to implement server to server communication for replication in PVFS2. The addition of server to server communication requires extensively modifying the file system's I/O server so that it can construct a data flow that simultaneously commits data to disk and forwards the data to another I/O server.

3.2 Replicated File System Data Consistency

A data consistency model specifies the constraints on the order in which file system operations can appear to occur from the viewpoint of the file system clients. We would minimally expect a consistent file system to return data that was previously written. The problems become far more difficult when two overlapping operations (reads or writes) are issued to the file system simultaneously or a fault occurs during a write.

A file system supports two fundamental data operations: read and write. Other data operations, such as truncate, have a subset of the consistency semantics of reads or writes. The read and write consistency semantics for file systems are not a universally agreed upon standard. Instead, the exact behavior of read and write usually depends upon the file system's targeted domain. Posix compliant file systems require that reads and writes to the same file must be serialized by the programmer with advisory locking to ensure consistent results [16]. These strong guarantees are easy to understand and leverage, but are often difficult to provide in even a non-redundant high performance file system. Popular network file systems, such as NFS, implement client-level caching that removes all strict guarantees of what file data is returned during overlapping reads and writes [13].

PVFS2, a high performance parallel file system, provides read and write semantics suitable for the access and performance requirements of scientific applications. Due to the

performance demands placed upon parallel file systems, PVFS2 provides data coherence semantics rather than sequential consistency. In both sequential consistency and coherency schemes no concurrent read serialization is performed. That is, if a read occurs on a region that is being simultaneously written to, the result of the read is undefined. With sequential consistency, simultaneous overlapping writes are serialized so that the overlapping region will wholly contain the data from one of the writes. The coherence scheme used in PVFS2 instead ensures that only each individual byte of the overlapping region will contain data from one of the competing writes.

Although coherence provides more parallelism because all write requests can proceed simultaneously, it does pose severe difficulties for a replicated file system. Since our protocol is implemented in the file system rather than at the device level, we cannot guarantee which data will be written to a given byte for two concurrent and overlapping writes. Because of this, the write data may be committed to disk in one order for the primary data object, and in a different order for the secondary data object. Out-of-sync primary and secondary servers are likely to lead to errors in applications and difficulties for application developers.

3.2.1 Ensuring Data Consistency on Writes

One mechanism to ensure data consistency in a replicated file system is to extend traditional file locking support to allow distributed file locking. A distributed locking scheme employs a global lock server that clients access in order to lock all of the data objects for a file. The lock server then becomes a single synchronization point (and possibly a bottleneck) for all file system accesses. While a client holds the lock tokens for a file (or a file region), no other client may access that file, thus ensuring that the replicated file contents are identical.

Unfortunately, the difficulties of building an intelligent lock server able to handle failures while clients are holding onto the locks are well known [18]. In particular, the lock tokens must have timeouts to handle the case where a client application is terminated

before releasing all of its outstanding lock tokens. In addition to the locking infrastructure accessed by clients, it is also necessary to have a scheme for detecting failures within the file system. For example, when an I/O node experiences a failure, the lock server or clients will need to recognize that one copy of the data no longer exists or is no longer valid. Intermittent failures are particularly difficult to handle with these types of schemes and may be common in clusters connected with fragile, high speed interconnection networks.

Heartbeat systems are another mechanism for detecting server failures in conjunction with file locking [29, 31]. A heartbeat system periodically sends small status messages to its mirror ensuring that both machines are functioning properly. When a server misses a heartbeat message, the lock server is notified and no longer directs clients to access the failed server. Heartbeat systems typically require a dedicated network connection to ensure that timeouts and dropped packets do not occur. This is particularly important for parallel file systems where large data transfers over the interconnection network are expected to occur continuously.

The primary-copy technique simplifies resiliency and consistency for parallel file systems because the client does not have to deal with inconsistency issues between the primary and replicated data stores. Instead, the primary server and secondary are responsible for determining their own consistency status using server to server communication. For example, on a file write, a client will begin sending data to the primary server. The primary server will first determine if the locally held data is consistent with the secondary server's data (the copy). If the primary is no longer consistent, then it will signal an error and the client then contacts the secondary server directly to perform the file write. The I/O servers are solely responsible for determining the consistency status of data stored locally.

When originally describing the primary-copy technique, Alsberg and Day also described a journalized transaction system to be used in conjunction with the primary-copy technique to ensure data consistency. The effectiveness of journalized transactions (and the two-phase locking used to implement it) to achieve identical primary and secondary stores

is well known by database implementors. But while transactions are required to implement sequentially consistent relational database systems, replicated file systems do not require such sophistication.

For replication in PVFS2 we have implemented the basic primary-copy mechanism; however, we have not implemented any type of journalized transaction schemes. We have chosen what we believe is the simplest scheme to achieve sequential consistency for simultaneous overlapping writes: we serialize overlapping writes in each server's request processing queue.

PVFS2 is based on a client-server architecture that causes each file system client to directly contact the I/O nodes it wishes to write data to. The write requests are passed into a processing queue that typically starts all read and write requests immediately (the queue exists to enable atomic metadata operations). Since all write requests to the same region in a file will occur on the same I/O servers (both the primary and secondary server), each I/O server can ensure that no overlapping writes are allowed on that server by serializing the overlapping requests.

Additionally, because the primary server initiates the write request on the secondary server, we can further ensure that the order of the overlapping operations is the same for both servers. The process involves modifying the request processor on the I/O server to detect overlapping client write requests and to service overlapping write requests in serial order rather than simultaneously. This approach is not ideal from an overhead standpoint and other serialization schemes may need to be explored for scientific workloads with a large number of overlapping writes. However, past studies have shown that writes to the same file region occur in less than 1 percent of file accesses [4, 25].

3.2.2 Ensuring Consistency During Failures

The other critical component of a consistency mechanism is ensuring that the primary and secondary servers are synchronized. In a cluster interconnection network where temporary

outages are common, the primary or secondary server may be unavailable during a write, but become available again during a later read from the written region. The client then must determine whether the primary or secondary data is the most up-to-date (note that we do not assume that a global clock exists). This issue is a special case of the classic computer science problem called the Byzantine generals problem [20].

Byzantine failure problems arise commonly in distributed environments where one distributed component must determine the reliability of another distributed component. One well known solution for this class of problems is to use majority voting; however, to ensure consistent file access it would be necessary for all the I/O servers to vote on every read and write. Our approach removes the need for frequent voting by allowing the primary and secondary to easily check their own consistency state against the other. Voting is used to resolve certain hard to handle error cases as described in Section 3.3.

In order for primary and secondary servers to determine their relative consistency status, we add a variable indicating the consistency state to each data object on an I/O node. When a file is created, a data object, or file storage area, is setup on each I/O node. At creation time, we initialize the consistency status to the *consistent* state, indicating that the data object is consistent with its mirror data object (i.e. both objects are empty and therefore consistent). Before any change (i.e. a file write) is performed to the data object, the consistency state must be checked against the secondary to determine if normal operation can proceed or if failover must occur. Similarly, before any read completes, the consistency states must be checked against the mirror copy to ensure that both objects are mutually consistent.

Consistency During File Writes

Figure 3.4 shows a sequence diagram documenting the interaction between the primary server and secondary server when their respective data objects are mutually consistent during a file write. The primary server must first ensure that both it and the secondary server

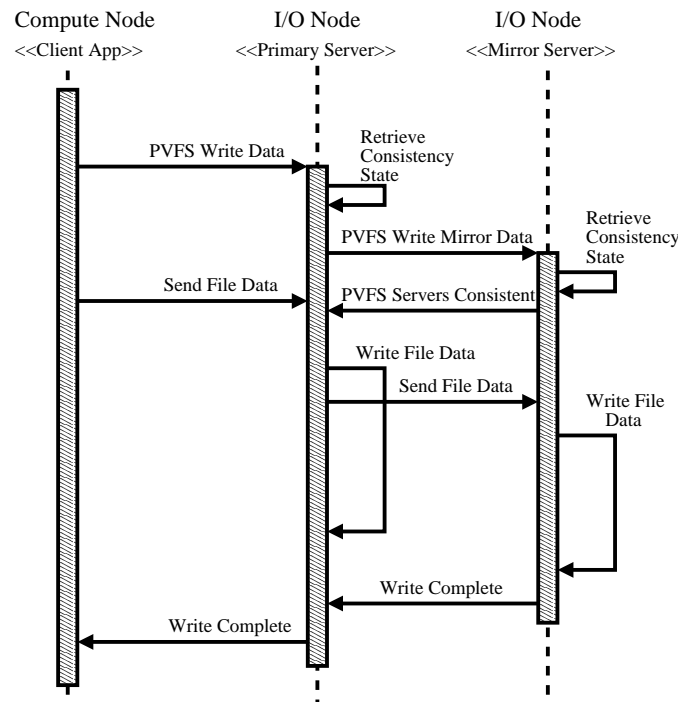


Figure 3.4: Sequence Diagram for Writes

are in the *consistent* state before the write can be committed to the repository. If the state values indicate that either the primary data object or the secondary data object are not consistent, then a fail-over protocol will be activated.

Consistency During File Reads

Figure 3.5 shows a sequence diagram documenting the interaction between the primary and secondary servers for a successful read. In this diagram, the objects are consistent and the read is successful, see sections 3.3, 3.4 for a description of how data objects in an inconsistent state are handled. Though the consistency status must be checked on all reads, there is no requirement that clients read from only the primary server. Instead, clients may read from both the primary and secondary servers simultaneously, with each server checking its relative consistency status on every read. The servers must ensure that the repositories are consistent before completing the read; however, it is possible to begin returning read data before the consistency check has completed.

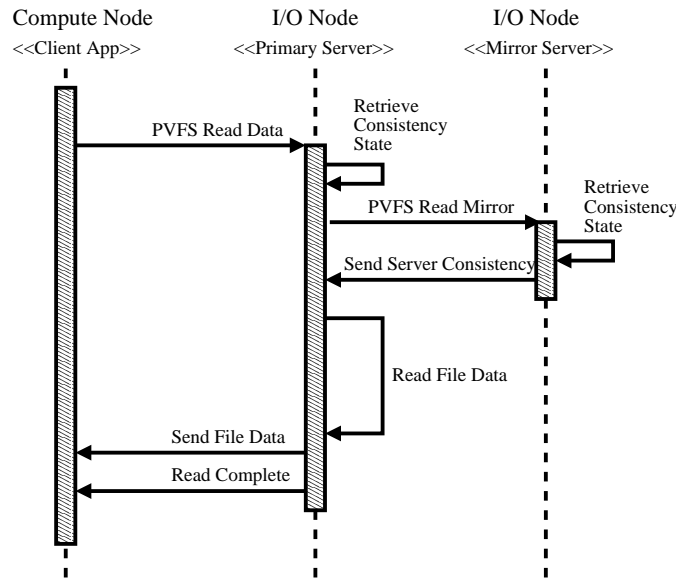


Figure 3.5: Sequence Diagram for Reads

3.3 Failure Scenarios

A failure scenario occurs when either the primary or secondary servers cannot be validated as mutually consistent or become unavailable. Thus far we have described how the consistency checks work in the case that both the primary and secondary data are in the *consistent* state. It is also important to describe how the consistency mechanism detects the case where the stored copies are not identical.

3.3.1 Standalone State

In the case of a failure for a primary or secondary server, the remaining server will enter the *standalone* state. The standalone state ensures that once a server fails, the failed server can not service any future requests (at least until recovery is performed). In effect, a standalone server operates similarly to a server in a non-replicating parallel file system. No checking with the mirror server is performed by a standalone server, instead the standalone copy of the data object is automatically deemed consistent.

In order to enter standalone state, an I/O server must request a vote with a majority of the active servers agreeing that the mirror has not already entered standalone mode. Once one server has entered the standalone state, the quorum that participated in that vote will not allow the mirror server to enter the standalone state until a recovery to normal operation has occurred.

Depending on the outage that occurs, either the primary or secondary becomes the standalone server. Once one server has entered standalone state, the failed mirror is unavailable until a recovery process is initiated. Even if the standalone copy is temporarily unavailable, it will still be valid when it becomes available again because it is not possible to write to the mirror copy after one server has entered the standalone mode. So when a previously failed server requests its partners consistency status, the standalone server will send a message informing the failed server that it may not continue. If the standalone server is not available, the previously failed server will not be able to achieve a successful quorum vote, so under either circumstance, the failed server will not be able to satisfy the clients request.

3.3.2 Voting and Quorums

Establishing a quorum and voting on which repository holds the most up-to-date data is a technique commonly employed in distributed systems. Such voting is able to provide high levels of resiliency and consistency; however, the process of establishing a quorum can be incur considerable latency. Due to the additional delay associated with voting, we only employ our voting protocol whenever a failure is first detected. During normal operation (i.e. the primary and secondary servers are consistent) no voting is required. In the case where hardware has failed, we do not think performance is the foremost concern, and instead our technique endeavors to reliably determine the most up-to-date copy of the data.

A valid quorum is achieved when more than half of the servers (a simple majority) send a previously agreed upon response to the requesting server. In the case of our replication mechanism, the possible quorum responses are that no server has entered standalone mode for the failed data object, or that a server has already entered standalone mode for the specified data object.

In the case where quorum cannot be achieved (i.e. half of the servers cannot be contacted by the primary I/O node or its backup) the client is notified of the file I/O failure and the file is lost until a system administrator can interact with the system to attempt recovery. In general, we would only expect this behavior to occur in a catastrophic failure scenario involving switches and network partitions. In such a case, system maintenance is warranted and an unavailable (but not corrupted) file system is not unreasonable.

3.3.3 Failure Protocol Use Cases

Server failures can be broadly classified into two categories: failures that occur causing a server to enter standalone state and failures that occur after standalone state has been entered. In order to describe the possible failure scenarios it is important to first clarify our terminology.

During normal operation, that is when both the primary and secondary server data is perfectly synchronized, both servers are in the consistent state. A server continues to be consistent until it misses an I/O request (read or write), forcing one of the servers into standalone mode. Strictly speaking, if the I/O operation “missed” was a read, both servers still maintain identical data. However, because of the performance advantages of entering standalone mode rather than requesting quorum operations, it is preferable to enter standalone mode on any missed I/O operation.

When a server no longer provides a timely network response, we call that server unavailable. Unavailability is the first failure that occurs, and generally results in the re-

maintaining server entering standalone mode. The following failure cases lead to one server entering standalone mode.

- Primary Consistent, Secondary Unavailable (requires quorum)
- Primary Unavailable, Secondary Consistent (requires quorum)
- Primary Unavailable, Secondary Unavailable

The above scenarios can all be considered “first failures;” that is they lead to one server entering standalone state (if a server is available). The other class of faults our protocol handles are the more difficult class of Byzantine faults. Although many types of Byzantine faults are possible, our protocol is focused on handling failure scenarios brought about by intermittent failures rather than true sabotage.

A server initially enters standalone state because the mirror server is unavailable. After some period of time it is possible the unavailable server will become available again. One possibility is a server misses a write due to a network congestion induced timeout. Once the congestion dissipates, the unavailable server will be available again – and it will not be aware that its mirror has entered the standalone state.

The remaining four failure scenarios are:

- Standalone(Primary) Available, Secondary Inconsistent
- Primary Inconsistent, Standalone(Secondary) Available
- Standalone(Primary) Unavailable, Secondary Inconsistent (requires quorum)
- Primary Inconsistent, Standalone(Secondary) Unavailable (requires quorum)

Additionally, it is also possible that both the standalone and its mirror are unavailable; however, the behavior is identical to the case where both the primary and secondary are unavailable so we have not added the duplicate scenario behavior.

In the following sections we describe hypothetical scenarios that lead to the failure case and a detailed description of the server and client interactions. An io-request refers to either a single read request or a single write request from a file system client. We have constructed hypothetical messages called standalone-request, failover-successful, and failover-unsuccessful to represent querying the quorum and the possible collective responses from the quorum.

Primary Consistent, Secondary Unavailable

Description: A client attempts to perform an io-request while the secondary server is not available on the network. The primary server will request a successful quorum vote and enter the standalone state.

1. A client sends an io-request to the primary server.
2. Simultaneously, the primary server:
 - (a) begins service of the client's io-request.
 - (b) queries the consistency state of the secondary server.
3. The secondary server does not respond to the primary request.
4. The primary sends a standalone-request to all data servers.
5. A quorum of servers responds failover-successful to the primary server.
6. (Write service completes for primary server).
7. Primary returns successful ack status to Client.

Primary Unavailable, Secondary Consistent

Description: A client attempts to perform an io-request while the primary server is not available on the network. The client will then contact the secondary server which will request a successful quorum vote and enter the standalone state.

1. The client sends an io-request to the primary server.
2. The primary server does not respond.
3. The client sends the io-request to the secondary server.
4. Simultaneously, the secondary server:
 - (a) begins service of the client's io-request.
 - (b) queries the consistency status of the secondary server.
5. The primary server does not respond to secondary.
6. The secondary sends standalone request to all data servers.
7. Quorum of servers respond failover-successful to the secondary server.
8. The secondary returns successful ack to the client.

Primary Unavailable, Secondary Unavailable

Description: Both data servers are down/unavailable, or the client has become disconnected from the data servers.

1. Client sends an io-request to the primary server.
2. Primary server does not respond.
3. Client sends an io-request to the secondary server.
4. Secondary does not respond.

5. Client cannot perform request.

Primary Standalone Available, Secondary Inconsistent

Description: Client 1 attempts to perform an io-request, but the secondary server is temporarily unavailable and the primary enters standalone state. Client 2 attempts to perform a read request from the secondary server which has become available. The secondary server will attempt to determine the consistency state of the primary before completing the read request, however, since the primary is in standalone mode the secondary cannot successfully complete the read request.

1. Client 2 sends a read request to the secondary server.
2. Simultaneously, the secondary server:
 - (a) begins service of the client's read request.
 - (b) queries the consistency status of the primary server.
3. The primary server responds that it is in standalone state.
4. The secondary server returns a failure message to the client.
5. The client contacts the primary to complete the read.

Primary Inconsistent, Secondary Standalone Available

Description: Client 1 attempts to perform an io-request, but the primary is temporarily unavailable and the client fails over to the secondary which successfully enters standalone state. Client 2 now attempts to perform a write to primary, which is again available, but not consistent.

1. Client sends an io-request to the primary server.
2. Simultaneously, the primary server:

- (a) begins service of the client's io-request.
 - (b) queries the consistency status of the secondary server.
3. The secondary server does not respond to the primary.
 4. The primary sends a standalone request to all data servers.
 5. Quorum of servers respond failover-unsuccessful to the primary.
 6. The primary returns failure status to the client.
 7. The client initiates the io-request with the secondary/standalone.

Primary Standalone Unavailable, Secondary Inconsistent

Description: The secondary has previously failed, causing the primary server to enter standalone mode. Client 2, a new client that did not participate in the request that led to standalone behavior, attempts to contact the primary. The primary/standalone is unavailable, so the client then attempts to contact the secondary, which is inconsistent.

1. Client sends write-request to the primary server.
2. Primary does not respond or sends an unavailable-response to the client.
3. Client sends write-request to the secondary server.
4. Simultaneously, the secondary server:
 - (a) begins service of the client's io-request.
 - (b) queries the consistency status of the secondary server.
5. The primary server does not respond to secondary.
6. The secondary sends standalone request to all data servers.
7. Quorum of servers respond failover-unsuccessful to the secondary server.

8. Secondary returns failure status to client.
9. Client cannot perform the request.

Primary Inconsistent, Secondary Standalone Unavailable

Description: A failover to secondary has previously occurred for Client 1. Client 2 now attempts to contact the out-of-date primary server, the standalone secondary server cannot be contacted by the primary. The primary will fail to enter standalone mode and respond to all future queries as unavailable until recovery is performed. The client then attempts to contact the standalone secondary directly. When the secondary becomes available, it will be able to service io-requests successfully.

1. Client sends an io-request to the primary server.
2. Simultaneously, the primary server:
 - (a) begins service of the client's io-request.
 - (b) queries the consistency status of the secondary server.
3. The secondary server does not respond to the primary.
4. The primary sends a standalone request to all data servers.
5. Quorum of servers respond failover-unsuccessful to the primary.
6. The primary returns failure status to the client.
7. The client initiates the io-request with the secondary.

Finally, four of the described failure cases are dependent upon a voting quorum. If a quorum cannot be achieved because half of the servers are unavailable the read or write operation is unsuccessful. The client must be notified of the failure, and in general the file system should be considered unavailable until at least a majority of the servers are available.

3.4 Recovery Protocol

After a failure has occurred and one or several servers have entered standalone state it is important that the file system can be recovered to a fully replicated state once all the I/O nodes have been repaired. To that end, each file system server can be started with a recovery flag to repair itself and re-enter normal operation. The following steps are performed to achieve recovery:

1. The failed/recovering server sends messages to every server to find the file handles which have experienced a loss of replication and a checksum for each failed file.
2. The standalone server performs a checksum on its local data object and sends the checksum to the failed server.
 - (a) If the checksums are identical, proceed to next step.
 - (b) If the checksums are not identical, the failed server must perform a read of the standalone data object to synchronize both data copies.
3. The recovering server resets the consistency status for the data object so that the primary and mirror copy are both consistent.
4. The standalone server sends a quorum message to notify a majority of the remaining servers that the failed file handle is no longer failed.
5. Normal replicated service resumes.

One limitation of this recovery protocol is that it is not an online recovery protocol. In order to perform recovery, the file system cannot allow simultaneous modification of data that is being recovered. The checksum mechanism does not work reliably if the file is changed during or after checksum computation.

In order to add online recovery, it would be possible to lock the file on the standalone server while the mirror copy is being synchronized; however, this would require

development of all the infrastructure necessary for a robust locking system (timeout support, etc.). A better system may use a journalized transaction log so that all subsequent writes to the file can be applied from the journal onto the mirror copy. In this thesis we have focused on keeping the recovery system simple so that the file system can be reliably recovered.

CHAPTER 4

IMPLEMENTATION AND PERFORMANCE

In order to verify the scalable performance of the modifications described in this thesis, we have implemented the proposed replication mechanism in the Parallel Virtual File System version 2 (PVFS2). PVFS2 is a popular parallel file system that is designed to run on Beowulf clusters using either commodity networks or high performance cluster networks.

To determine the real world effects of our proposed replication scheme we performed a series of verification experiments using an aggregate I/O bandwidth benchmark. Our goal is to show that although the effective client bandwidth may be cut in half while writing to a parallel file system with replication, the scalability of the file system should stay consistent.

4.1 Experiment Configurations

In order to measure the performance impacts of replication we measured two different distributions in our experiments. As a performance baseline, we always used the standard PVFS2 distribution (called `simple_stripe`) which does not perform any replication. We also used the replication data distribution described in Section 3.1 and shown in Figure 4.1.

The distribution scheme shown in Figure 4.1 allows all the servers to act simultaneously as primary and secondary servers. Consider that a contiguous read of the data strips labeled 'A' and 'B' would result in the client reading the first half of strip 'B' from node 1, and simultaneously the second half of strip 'A' is read from node 1 (the remaining data for 'A' and 'B' would be read from nodes 0 and 2 respectively). In this example, server 1 is acting as the primary server for strip B and the secondary server for strip A. File writes exhibit similar behavior, though in that case the clients access the primaries only, and the server to server communication writes the secondary data object.

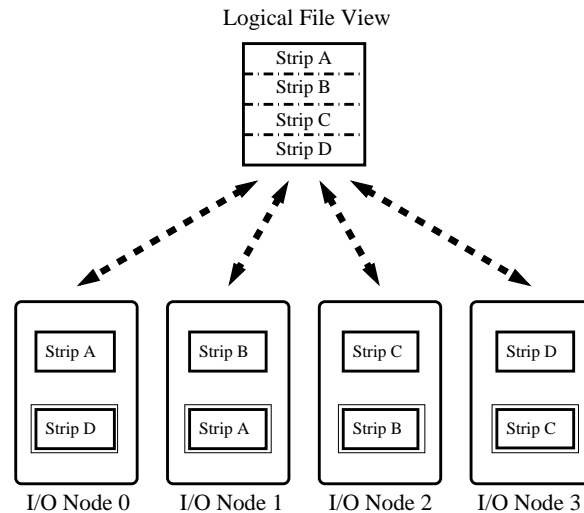


Figure 4.1: Data Distribution Providing Replication in Experiments

PVFS2 includes in its distribution a program called `mpi_io_test` that we use to measure aggregate bandwidth as seen by the client. The test program uses the standard MPI collective, `MPI_File_write` so that each of the computation nodes simultaneously writes 16MB of local non-contiguous and non-overlapping data to construct a single contiguous file that spans all the I/O servers in the PVFS2 file system. We measure the write bandwidth as the time it takes for the clients to finish writing all of the data, irrespective of the amount of data committed to disk being doubled due to data replication.

Similarly, the program performs a collective `MPI_File_read` to read the contiguous data file to construct local non-contiguous data arrays on each of the computation nodes. The read bandwidth is measured as the time it takes for the clients to finish receiving a single copy of all of the file data without respect to whether or not the data has been replicated.

The I/O bandwidth of a parallel file system is primarily constrained by the network throughput or by the disk transfer rate. In order to measure the effects of replication in realistic scenarios we performed our tests on two different clusters: one with performance primarily limited by the network and another cluster with performance primarily limited by the disk speed.

For the network constrained cluster we used Adenine, the Clemson University PARL lab's Beowulf cluster. Adenine is composed of 48 dual CPU nodes interconnected with 100Mbit Fast Ethernet. In all of our test configurations we assigned each node to be either a computation node or an I/O node. No more than one application process (whether a client process or a PVFS2 server) ran on any of the nodes.

For the disk constrained cluster we used Argonne National Laboratory's Jazz cluster. Jazz is composed of approximately 250 single CPU nodes connected with a Myrinet interconnection network. The Myrinet interconnect (Myricomm 2000) uses cut through switching for low latency and can achieve a sustained bandwidth of approximately 1.2Gbit.

In all of our graphs, the number of I/O nodes labeled in the key refers to the number of I/O nodes in the entire file system. It does not refer to the number of nodes deployed as primary or secondary servers. For these tests all I/O nodes acted as both a primary and secondary server.

4.2 File System Write Modifications

Providing support for data replication in PVFS2 was not a simple matter of adding a replication mechanism to the file system. In order to achieve scalable write performance while replicating data it was necessary to modify the file system write infrastructure to allow individual writes to be multi-threaded.

In the stock version of PVFS2 distributed by its developers, PVFS2 uses only a single thread per file write request. The write thread fills a 256KB buffer with data and uses an asynchronous I/O call to write the data to disk. Upon completion of the write request, the thread then refills the 256KB buffer and writes the next data chunk. Because the operating system and network hardware provide full buffering support, a single threaded approach is sufficient for writing a single requests data.

Unfortunately, our replication modifications lead to slower data throughput than necessary when using the single thread per write request model. Recall that our replication

mechanism forwards data from the primary server to the secondary server. If a single execution flow is used it would not be possible to send and receive data at the same time, even though most network hardware is full duplex and can support simultaneous sending and receiving of data over the network. So even though the network hardware and operating system are buffering incoming data effectively, we are still increasing the delay by devoting a single thread to transferring data (an I/O intensive activity) while more processing could be done to transfer the operating systems buffers into program memory and scheduling the disk transfers.

In order to overcome this limitation, we have modified PVFS2 to allow several threads to be allocated to a single write. This allows a primary server to package the next incoming data packet for disk I/O while simultaneously sending the current packet to the secondary server. In order to provide deterministic performance, we have implemented the data receipt, write, and forwarding mechanisms as a simple 3 stage pipeline. The use of standard non-deterministic threads sometimes leads to unpredictable delays and lower aggregate bandwidth utilization, however a deterministic pipelined approach provides stable performance and high bandwidth utilization.

Figures 4.2 and 4.4 are the bandwidth curves generated while performing non-replicated writes using the standard PVFS2 distribution and our modified version using a multi-threaded (pipelined) write. The horizontal axis shows the number of computation nodes participating simultaneously in the collective I/O operation. The vertical axis shows the aggregate data bandwidth as seen by the computation nodes. Figures 4.3 and 4.5 showing the bandwidth curves for aggregate read performance are included for posterity.

No data forwarding is performed (making the pipeline only 2 stages), but as is illustrated these modifications do not have a significant impact on the performance or scalability of PVFS2 in their own right, they only affect performance during replication. That is, modifying PVFS2 to allocate multiple threads per file write does not significantly impact the file system write performance in either a positive or negative fashion.

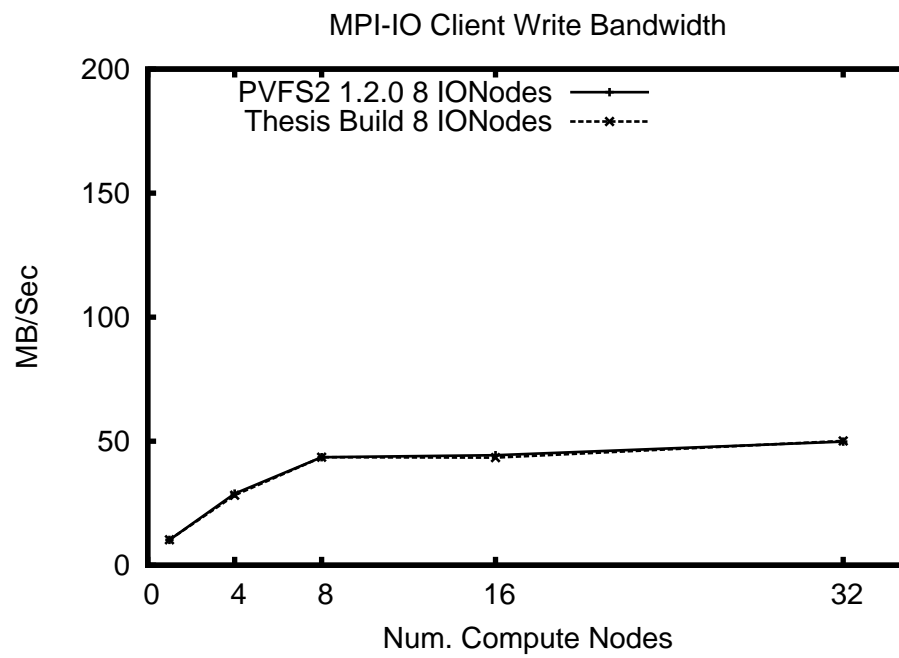


Figure 4.2: Comparison of aggregate write bandwidth with multi-threaded writes on Adenine w/ 8 IO Nodes

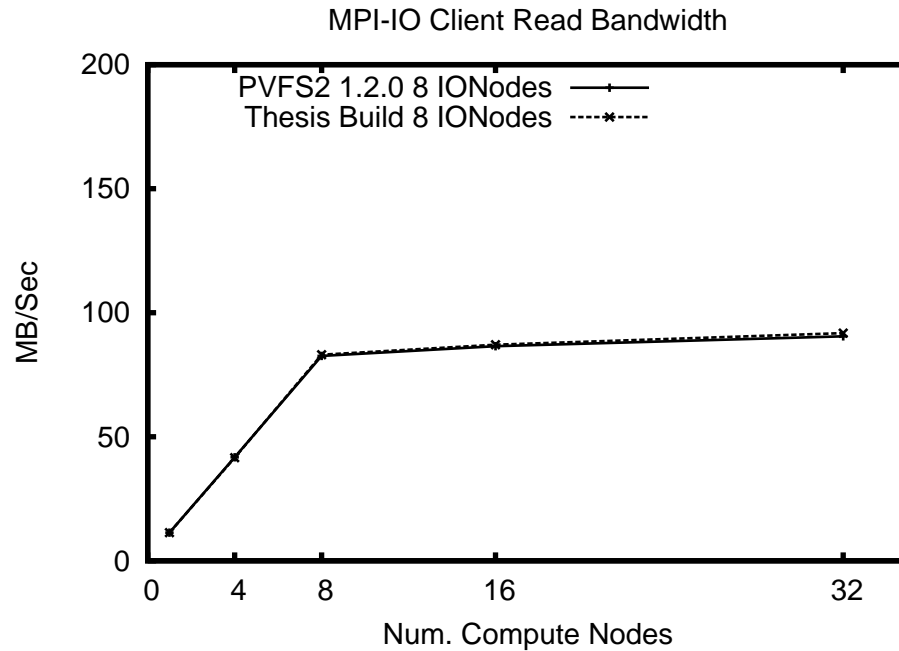


Figure 4.3: Comparison of aggregate read bandwidth with multi-threaded writes on Adenine w/ 8 IO Nodes

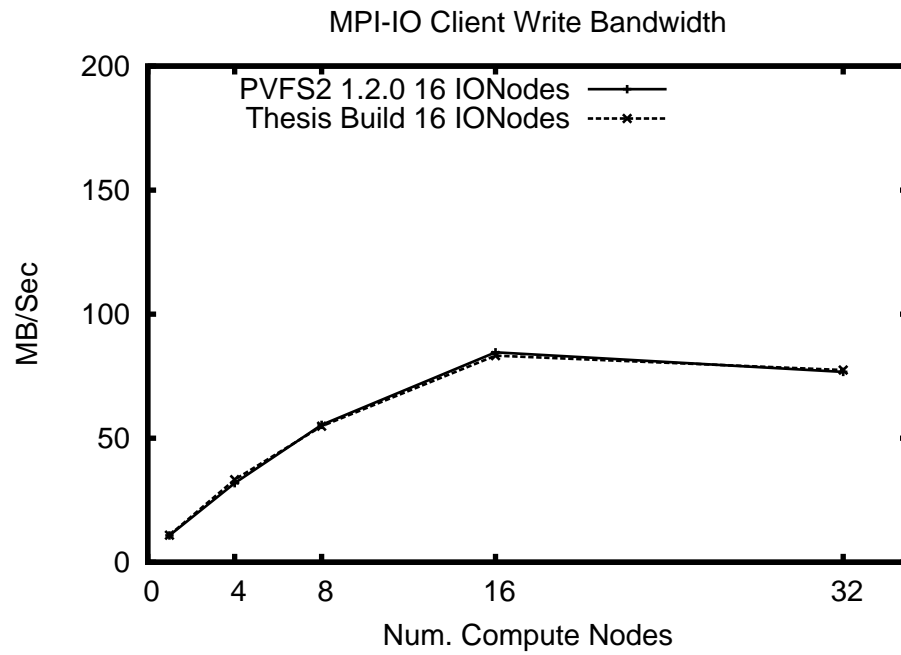


Figure 4.4: Comparison of aggregate write bandwidth with multi-threaded writes on Adenine w/ 16 IO Nodes

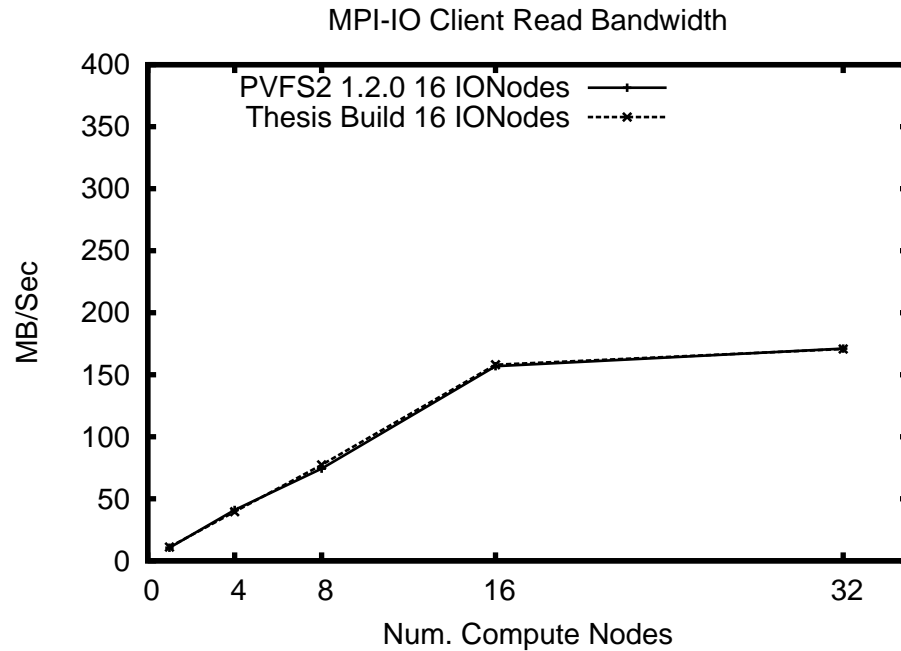


Figure 4.5: Comparison of aggregate read bandwidth with multi-threaded writes on Adenine w/ 16 IO Nodes

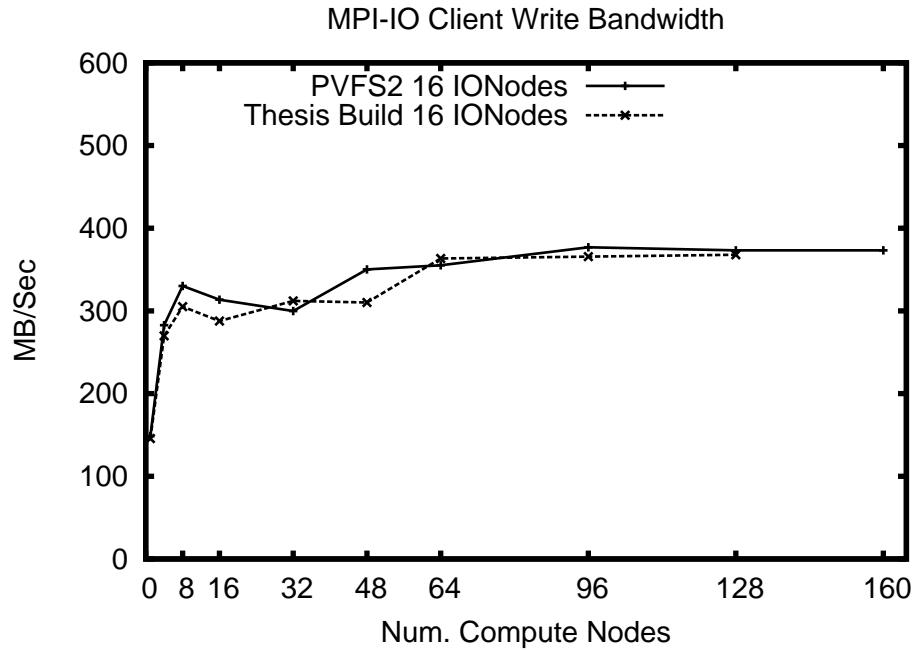


Figure 4.6: Comparison of aggregate read bandwidth with multi-threaded writes on Jazz w/ 16 IO Nodes

The diminishing slopes in figures 4.2 and 4.4 are due to the saturation of the network hardware. The Adenine cluster uses a 100Mbit Ethernet interconnect that is much slower than the operating system's ability to commit data to the file system (the operating system uses caching for asynchronous I/O primitives to make the disks seem much faster than their actual transfer rates).

Similarly, figures 4.6 and 4.7 illustrate that the addition of multi-threaded writes does not impact the file system performance on a cluster using a high speed interconnection network. The Jazz cluster uses a low latency, relatively high bandwidth interconnect called Myrinet. The much higher aggregate bandwidth scores are entirely due to the improved network, as the individual Jazz nodes do not have significantly faster disks, larger memory subsystems, or faster processors than Adenine. The large disparity between the aggregate bandwidth scores for reading and writing illustrates the effectiveness of the block cache used by the Linux operating system. In our tests we did not flush the block cache before reads so that the highest possible read performance could be measured and any pos-

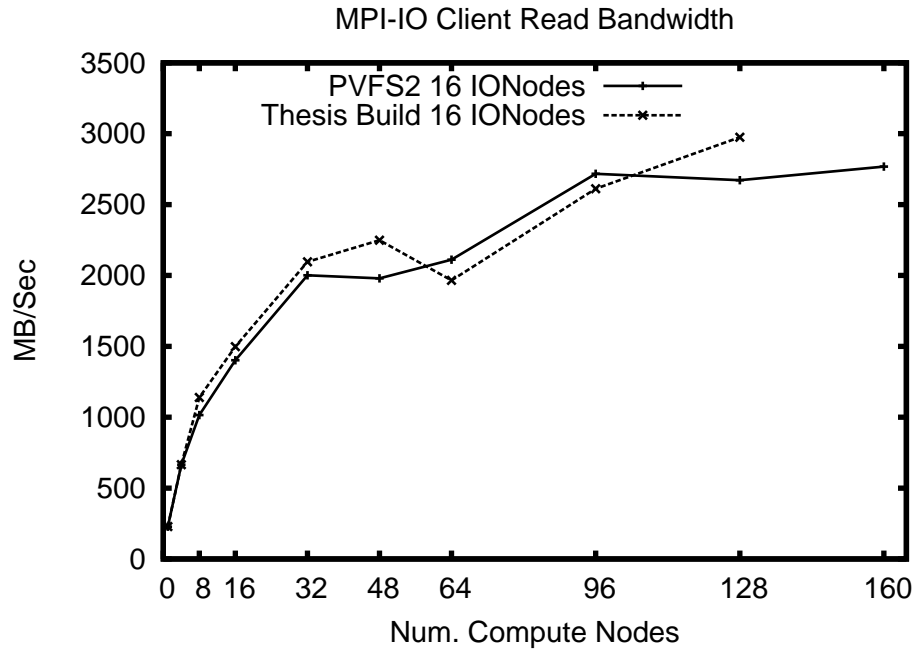


Figure 4.7: Comparison of aggregate write bandwidth with multi-threaded writes on Jazz w/ 16 IO Nodes

sible impacts or our modifications could be ascertained. In this case, where only the write mechanism is modified the matter is mostly academic. However, in later tests we will discuss possible effects of the block cache more thoroughly.

4.3 Write Performance

The performance impact of replication on write bandwidth is primarily governed by two constraints. On the one hand, we expect that the cost of writing twice as much data over the network and to disk will double the time to complete a given write. On the other hand, the use of pipelining may allow us to more effectively utilize the network and disk, thus mitigating some of the additional bandwidth requirements when the system is lightly loaded. Tables 4.1 and 4.2 show that as the number of clients is increased (i.e. the system load is increased) the write bandwidth with replication is reduced to 50% of the normal write bandwidth.

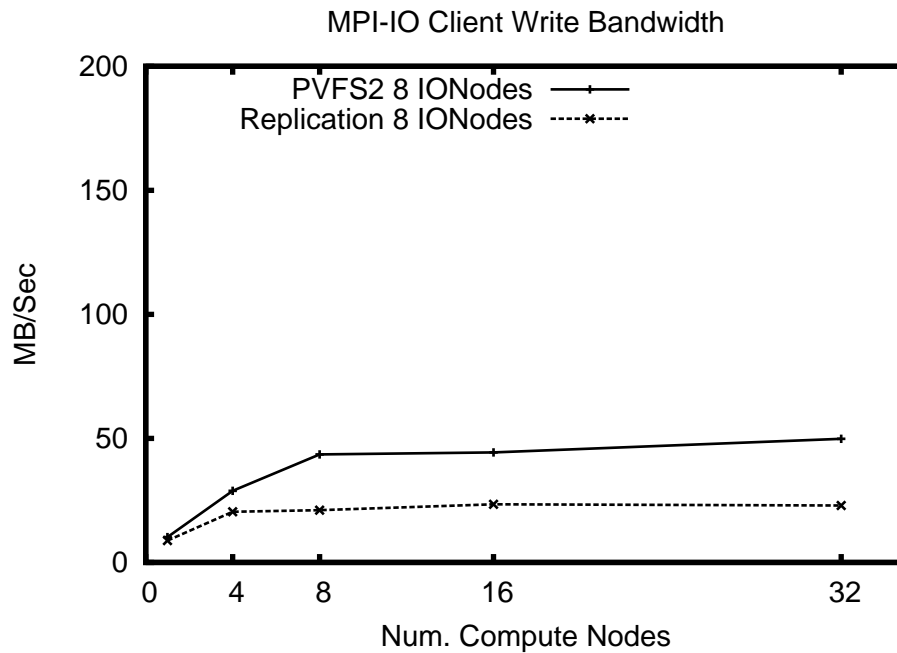


Figure 4.8: Performance on Adenine w/ 8 IO Nodes

Write Bandwidth (MB/Sec)			
Num Clients	Stock PVFS2	PVFS2 w/Replication	Percent Performance
1	10.3	8.8	85.4%
4	28.2	20.4	72.3%
8	43.5	21.1	48.5%
16	43.4	23.4	53.9%
32	50.1	22.9	45.7%

Table 4.1: Relative replication performance on Adenine with 8 I/O servers

Write Bandwidth (MB/Sec)			
Num Clients	Stock PVFS2	PVFS2 w/Replication	Percent Performance
1	10.9	9.8	89.9%
4	33.3	28.7	86.2%
8	54.8	39.8	72.6%
16	83.3	40.3	48.4%
32	77.5	38.2	49.3%

Table 4.2: Relative replication performance on Adenine with 16 I/O servers

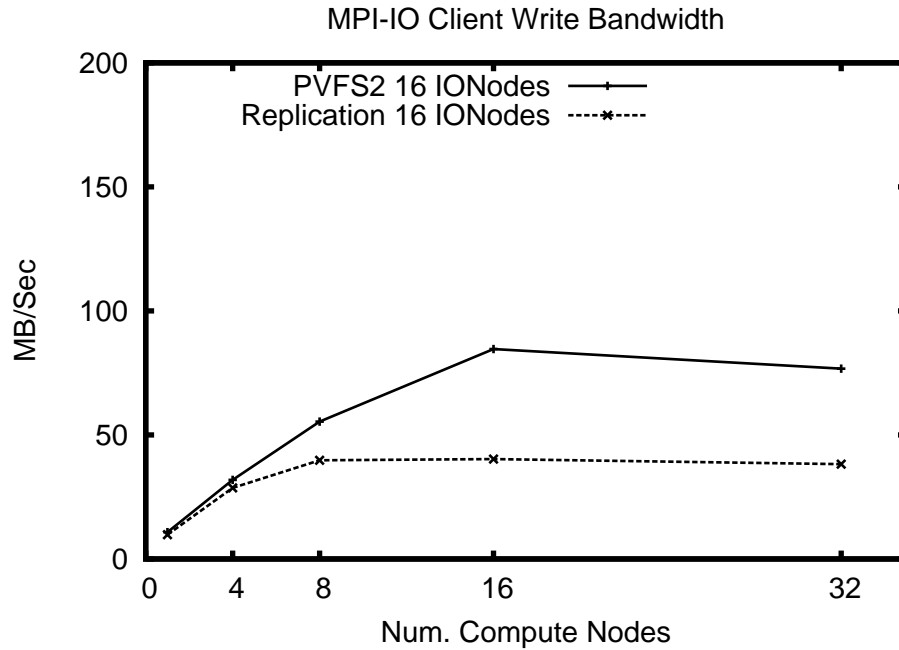


Figure 4.9: Performance on Adenine w/ 16 IO Nodes

Write Bandwidth (MB/Sec)			
Num Clients	PVFS2 8 I/O nodes	PVFS2 w/Rep. 16 I/O nodes	Percent Performance
1	10.3	9.8	95.1%
4	28.2	28.7	101.8%
8	43.5	39.8	91.5%
16	43.4	40.3	92.9%
32	50.1	38.2	76.2%

Table 4.3: Performance of a replicated and non-replicated file system on Adenine

These results confirm our intuition of how replication would affect write bandwidth (as seen by the clients). Figures 4.8 and 4.9 depict the write bandwidth curves for 8 and 16 I/O servers respectively. As you can see, the general shape of the bandwidth curves is very similar; however, the replicated file system reaches its maximum performance with a smaller number of clients than the non-replicating file system.

Table 4.3 compares the performance of an 8 I/O node non-replicated file system with the performance of a 16 I/O node fault tolerant file system. At all configurations less than 32 clients, the file systems provide very similar write bandwidths. At 32 clients the

Write Bandwidth (MB/Sec)			
Num Clients	Stock PVFS2	PVFS2 w/Replication	Percent Performance
1	145.7	107.6	73.9%
4	269.8	154.6	57.3%
8	305.2	173.7	56.9%
16	287.7	137.1	47.7%
32	312.2	183.7	58.8%
64	363.9	157.1	43.2%
128	367.4	180.1	49.0%

Table 4.4: Relative replication performance on Jazz with 16 I/O servers

performance of the replicated file system falls to only 72.6% of the non-fault tolerant file system.

Evaluating the performance of replication on the Adenine cluster can be difficult because the network bandwidth quickly becomes the bottleneck even while using a relatively small number of clients. The Jazz computational cluster uses a high performance interconnection network based on cut through switching that alleviates the network bottleneck during writes. Instead, the bottleneck becomes how fast the operating system can commit data to the file system, which is much faster than data can be written to disk. Due to the much higher network throughput available (and the large number of nodes available), we are able to benchmark with a much higher system load than was available for the Adenine cluster.

Table 4.4 and Figure 4.10 show a 16 I/O node file system configuration with up to 128 clients. In general, as the load is increases the replicated file systems available client bandwidth is only half the available non-replicated bandwidth. Unfortunately, there is significant noise in the data making it difficult to draw exact conclusions. Although the interconnection network used on Jazz is much faster than the 100Mbit Ethernet used on Adenine, the network is also hierarchical, meaning that communication between any two nodes may not be identical due to congestion and additional routing time between some destinations. Unfortunately, it is difficult to control for this behavior, particularly on jobs larger than 16 nodes that are guaranteed to traverse the network switch hierarchy.

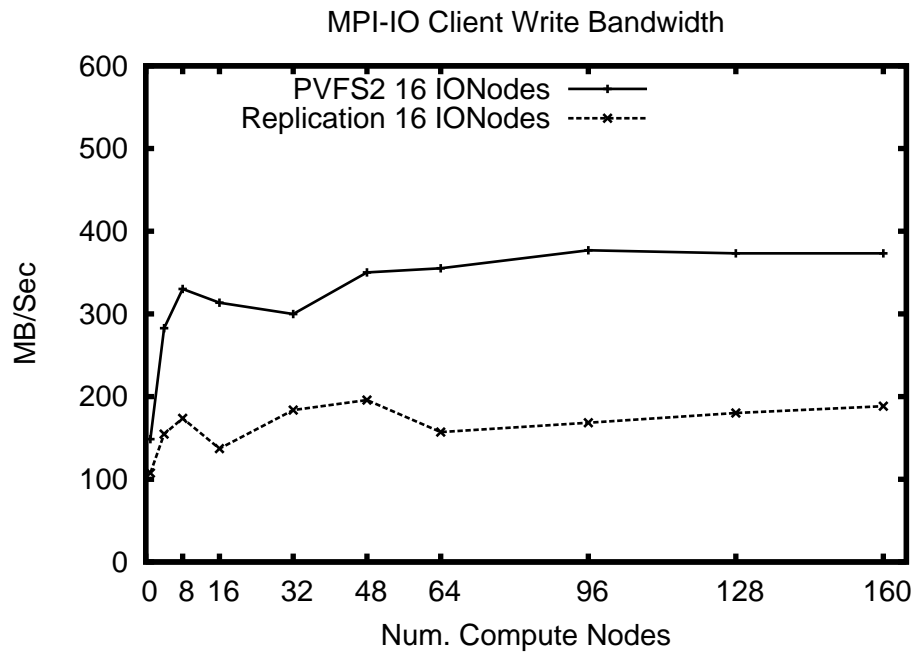


Figure 4.10: Performance on Jazz w/ 16 IO Nodes

Still, even with the data variance we can see that the knee of the curve has been reached on client sizes larger than 32 nodes. The bandwidth curves are effectively parallel at this point with a replicated file system providing 50% of the available bandwidth in a non-fault tolerant file system.

4.4 Read Performance

Read performance in a replicated file system should be very similar to the performance of a standard non-replicated file system. Figures 4.11 and 4.12 indeed confirm that for I/O jobs on Adenine, a network bound operation, read performance on a fault tolerant parallel file system is effectively identical to a parallel file system with no fault tolerance. This result is very encouraging. It implies that in a network bound environment, the additional overhead of consistency checks is inconsequential and the file system scalability is virtually the same.

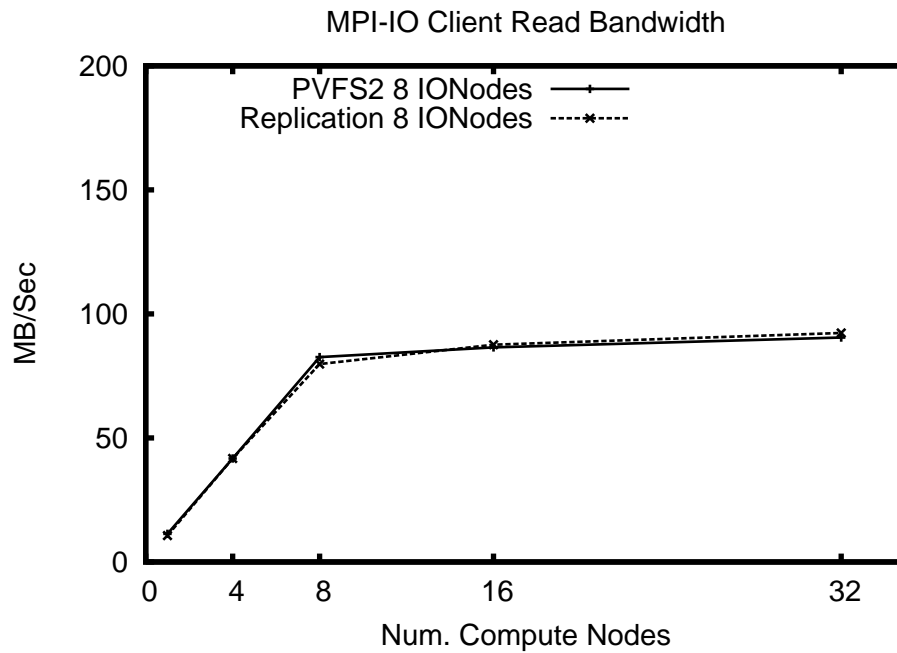


Figure 4.11: Performance on Adenine w/ 8 IO Nodes

In many ways, this result corresponds to our intuition. The cost of sending a small message between servers to ensure consistency should be much smaller than the cost of an I/O transfer. Of course in a highly congested Ethernet network the probability of sending the messages successfully may drop substantially, but on client sizes of 32 nodes or less, it is clear that the overhead is quite low.

Because Jazz uses a cut-through switching network (i.e. a deterministic network rather than a network with random backoff-based congestion control), we are not able to determine if high network loads can have a disproportionately large negative impact on the consistency overhead. However, we are able to see that despite some degree of noise in the data, read performance is mostly identical for a fault tolerant and non-fault tolerant parallel file system.

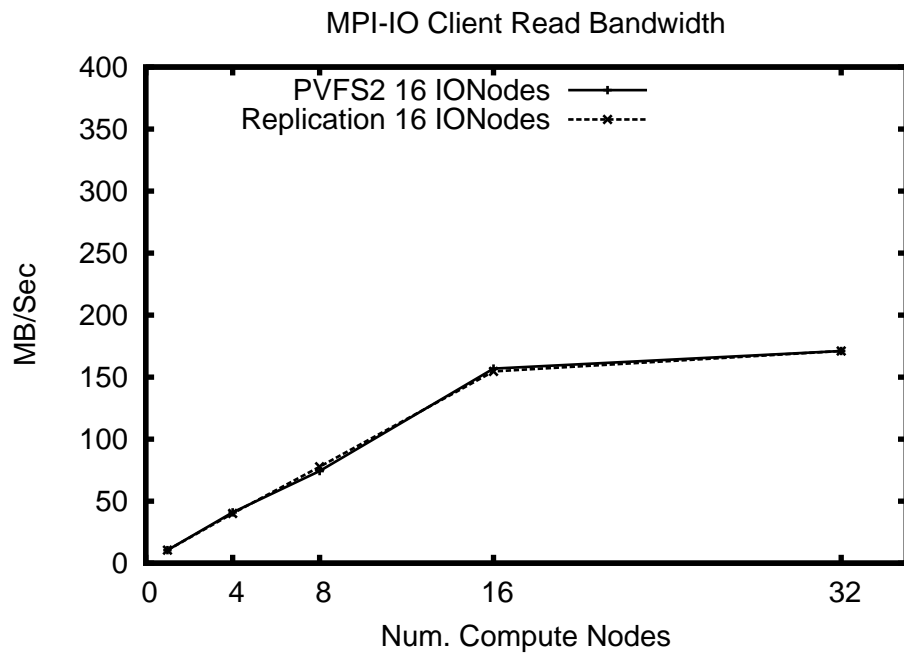


Figure 4.12: Performance on Adenine w/ 16 IO Nodes

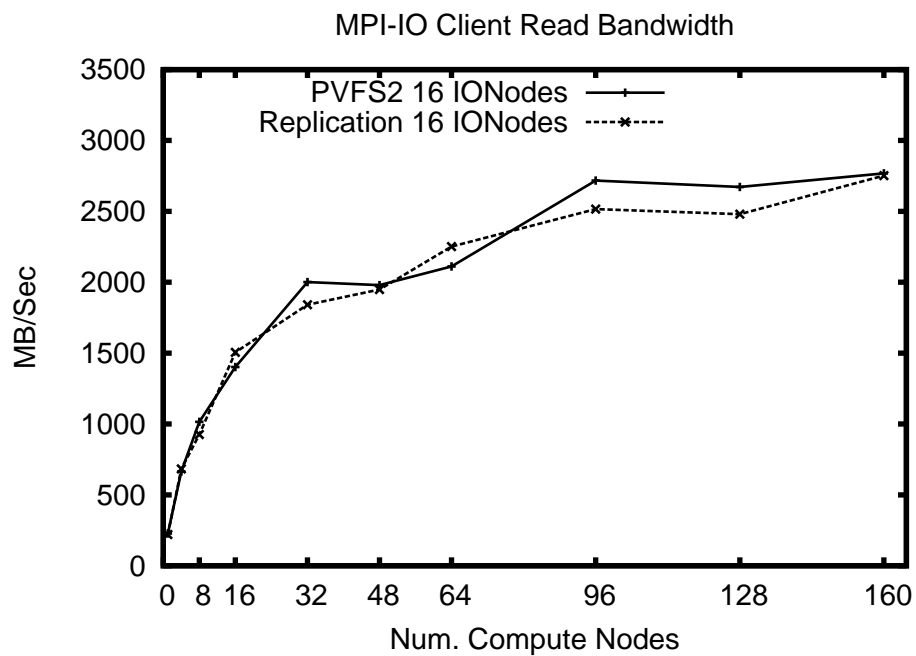


Figure 4.13: Performance on Jazz w/ 16 IO Nodes

CHAPTER 5

CONCLUSION

Experiments have shown that parallel file systems are an effective approach for achieving high performance I/O on large-scale cluster systems. However, as cluster I/O systems scale to larger and larger numbers of nodes, the need for fault tolerant I/O systems grows more urgent. We have identified resiliency, consistency, and performance as three critical aspects of determining a fault tolerance scheme's effectiveness for parallel file systems.

Our analysis in Chapter 3 demonstrated that it is possible to provide a high degree of data resiliency by replicating data to an independent I/O node. Replication is able to tolerate up to $1/2$ the I/O nodes failing with an expectation that $1/4$ of the servers can fail before the file system can no longer reconstruct a file. In addition to providing a scheme for tolerating the failure of storage servers, we have also described a robust online failover scheme and a server recovery scheme.

In Chapter 3 we also described how consistency is maintained using additional state attributes stored with each data object. We also described how the consistency checking process can be performed concurrently with an I/O request and how a quorum system can be employed to ensure that data is consistent when only a single copy of the data is available.

Finally, in Chapter 4 we benchmarked the performance of our parallel file system implementation. The benchmarks demonstrated that our protocol added very little overhead to the client read performance despite performing additional consistency checks to ensure that the backup data is intact. In many ways, the read performance of the fault tolerant file system may be considered a success. In general, we expect scientific applications to read and perform data processing many times, whereas scientific data is usually only written once, at acquisition time.

The benchmarks also showed that on a lightly loaded parallel file system the performance impacts on write bandwidth may be much less than a 50% degradation. On a heavily loaded I/O system the available client write bandwidth is very close to 50% of the non-replicated file systems bandwidth. Unfortunately, the additional bandwidth cost may not be acceptable to some aggressive data collection environments. If several Terabytes of telescope data is to be collected every night, the additional performance costs of fault tolerance may be too great. However, even in such cases, our experiments demonstrate that the observatory could double the number of I/O servers and reasonably expect to achieve fault tolerant file system performance similar to the original non-redundant file system.

In whole, we have shown that a server-to-server based replication scheme is able to meet our requirements of a high degree of data resiliency, data consistency, and performance.

5.1 Contributions

This work has yielded the following contributions to the field of parallel I/O system fault tolerance:

- Identification of three critical factors for evaluating a parallel I/O system fault tolerance scheme: resiliency, consistency, and performance.
- A rationale and design for a robust replication protocol for parallel file systems that addresses all three factors identified.
- A recovery protocol for restoring a parallel file system with non-critical failures to normal operation.
- An implementation of the replication protocol to measure the performance impacts and overhead of replicating data to independent servers.

5.2 Future Work

The scalable replication mechanism described in this thesis considerably improves parallel file system fault tolerance. Even so, several improvements and future projects have become evident during this work. In this section we will outline several extensions and improvements to the replication mechanism we described earlier.

Replication to a single backup server can double the mean time to failure of a file system, but on a large-scale file system, an improvement of 1 day to failure to 2 days until failure may not provide enough time to perform adequate repair. In a situation where single replication cannot provide enough reliability it may be necessary to replicate data to multiple independent I/O nodes. Furthermore, in a large-scale I/O system, extensive replication may allow parts of the system to be shutdown for maintenance or may be used to change the degree of locality so that data is replicated across different segments of a hierarchical network architecture.

Another useful improvement to our replication proposal could be an improved system for detecting overlapping writes. In our current system, every incoming write request must be checked to ensure it does not overlap any current write request. It may be possible to introduce some degree of transactional semantics or ordering semantics to ensure that overlapping writes are committed in the same order on both the primary and backup data copies.

PVFS2 currently has no support for transactions or atomic writes. In general, the performance impacts are deemed to be too detrimental for such mechanisms. However, in an environment where data reliability is given precedence over absolute performance, modifications to PVFS2 to allow atomic writes could greatly improve the file system reliability. This change has already come to workstation and server file systems in the form of journalized file systems. If atomic writes are added to a parallel file system, replication mechanisms for the parallel file system may need extensive modifications.

In addition to these larger endeavors, there are several minor improvements that could be made to replication that are not straightforward to design or implement. One example is allowing the servers to change roles, that is allowing the primary and backup copy to change dynamically, with changes perhaps governed by some load balancing constraints. Other improvements could involve extending the recovery protocol to allow online recovery.

BIBLIOGRAPHY

- [1] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd international conference on Software engineering*, pages 562–570. IEEE Computer Society Press, 1976.
- [2] Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, and Michael K. Reiter. Fault detection for byzantine quorum systems. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):996–1007, 2001.
- [3] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster i/o with river: making the fast case common. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 10–22. ACM Press, 1999.
- [4] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 198–212, New York, NY, USA, 1991. ACM Press.
- [5] Philip H. Carns. Design and analysis of a network transfer layer for parallel file systems. Master's thesis, Clemson University, Clemson, SC, December 2001.
- [6] Philip H. Carns. *Achieving Scalability in Parallel File Systems*. PhD thesis, Clemson University, Clemson, SC, May 2005.
- [7] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.
- [8] International Business Machines Corporation. Blue Gene/L specification sheet. <http://www-1.ibm.com/servers/deepcomputing>.
- [9] Standard Performance Evaluation Corporation. SPEC CPU benchmark results. <http://www.spec.org>.
- [10] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.
- [11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43. ACM Press, 2003.
- [12] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 150–162, 1979.
- [13] Network Working Group. Network file system (NFS) version 4 protocol. <http://www.ietf.org/rfc/rfc3530.txt>.

- [14] M. Halem, F. Shaffer, N. Palm, E. Salmon, S. Raghavan, and L. Kempster. Can we avoid a data survivability crisis? *Science Information Systems Newsletter*, (51), 1999.
- [15] Hui-I Hsiao and David DeWitt. Chained Declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of 6th International Data Engineering Conference*, pages 456–465, 1990.
- [16] The IEEE and The Open Group. IEEE std 1003.1, 2004 edition. <http://www.opengroup.org/onlinepubs/009695399/toc.htm>.
- [17] Minwen Ji, Alistair Veitch, and John Wilkes. Seneca: Remote mirroring done write.
- [18] Paris C. Kanellakis and Christos H. Papadimitriou. Is distributed locking harder? In *PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 98–107, New York, NY, USA, 1982. ACM Press.
- [19] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, Washington, DC, November 1994. IEEE Computer Society Press.
- [20] Lamport, Shostak, and Pease. The byzantine generals problem. In *Advances in Ultra-Dependable Distributed Systems*, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), IEEE Computer Society Press. 1995.
- [21] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 226–38. Association for Computing Machinery SIGOPS, 1991.
- [22] Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Pete Wyckoff, and D. K. Panda. Performance comparison of MPI implementations over Infiniband, Myrinet, and Quadrics. In *ACM/IEEE Proceedings of SC2003: High Performance Networking and Computing*, 2003.
- [23] H.W. Meuer, E. Strohmaier, and J.J. Dongarra. TOP500 Supercomputer Sites, 23rd edition. In *ACM/IEEE Proceedings of the Supercomputing Conference (SC2003)*, 2003.
- [24] H.W. Meuer, E. Strohmaier, and J.J. Dongarra. TOP500 Supercomputer Sites, 24th edition. In *ACM/IEEE Proceedings of the Supercomputing Conference (SC2004)*, 2004.
- [25] Brian Noble and M. Satyanarayanan. An empirical study of a highly available file system. In *Measurement and Modeling of Computer Systems*, pages 138–149, 1994.
- [26] U.S. Department of Energy. The office of science data-management challenge, March-May 2004.

- [27] Parallel Virtual File System 2. <http://www.pvfs.org/pvfs2>.
- [28] Fabrizio Petrini, Kei Davis, and José Carlos Sancho. System-level fault-tolerance in large-scale parallel machines with buffered coscheduling. In *9th IEEE Workshop on Fault-Tolerant Parallel, Distributed and Network-Centric Systems (FTPDS04)*, Santa Fe, NM, April 2004.
- [29] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, January 2002.
- [30] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.
- [31] D. Feng Y. Zhu, H. Jiang and D. Swanson. CEFT-PVFS: A cost-effective, fault-tolerant parallel virtual file system. Technical Report TR02-10-03, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 2002.