A. D. MCCXXIV

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunità Europea
Fondo Sociale Europeo

STRATEGIES FOR ACHIEVING DEPENDABILITY IN PARALLEL FILE SYSTEMS

GENEROSO PAOLILLO

Tesi di Dottorato di Ricerca

Novembre 2006

Dipartimento di Informatica e Sistemistica

via Claudio, 21- I-80125 Napoli - ☎ *[#39]* (0)81 768 3813 - *[#39]* (0)81 768 3816

A. D. MCCXXIV

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunità Europea
Fondo Sociale Europeo

STRATEGIES FOR ACHIEVING DEPENDABILITY IN PARALLEL FILE SYSTEMS

GENEROSO PAOLILLO

Tesi di Dottorato di Ricerca

(XIX Ciclo)

Novembre 2006

Il Tutore
Prof. Stefano Russo

Il Coordinatore del Dottorato
Prof. Luigi P. Cordella

Dipartimento di Informatica e Sistemistica

✉ via Claudio, 21- I-80125 Napoli - ☎ *[#39]* (0)81 768 3813 - 🖷 *[#39]* (0)81 768 3816

"A few observation and much reasoning lead to error;
many observations and a little reasoning to truth"
Alexis Carrel

Table of Contents

List of Tables

List of Figures

Acknowledgements

First and foremost thanks go my advisor, Prof. Stefano Russo, gave me the freedom to pursue my own academic interests. I am very pleased to have had the opportunity to work with him. Special thanks go to the Prof. Domenico Cotroneo for the opportunity he gave me to start this amazing experience of knowledge. He has been the person who closer followed my research studies. He was who initially suggested the topics for both my Masters and Ph.D. thesis. This dissertation has grown out of long and fruitful discussions with Mario Lauria. It was his great knowledge of parallel file systems that guided my studies during the experience I had in his research group at the Ohio State University. Antonio Strano also deserves thanks for contributing to my thesis through constant discussion, review, and technical assistance but first of all through his friendship. In working with him I felt more enthusiasm than before. Thanks go as well to all the members of the Mobilab Research Groups at University of Naples Federico II for their continued interest, advice, feedback, and discussions as the work in this dissertation matured.

Finally, a special thanks goes to my family, in particular to my mother, Carla. Her support has been unconditional and unwavering, even if she does not know how much she has been important in the fulfillment of this goal. Last, but certainly not least, I thank Anna whose love gave me the strength and encouragement to follow the path that I felt was right, and whose patience and trust allowed me the freedom to complete the task.

Naples, Italy — Generoso Paolillo
November 30, 2006

Introduction

High performance computing has long been focused more on central processing units and primary memory than on peripherals such as secondary and even tertiary storage. Recent applications such as database management systems (DBMSs), multimedia applications, and scientific simulations can produce huge amount of data, thus requiring I/O hardware and software that can move data rapidly across network to and from storage devices. Furthermore, these needs vary widely between different types of applications. For instance, many database and multimedia applications focus on reading data, while many scientific applications focus on writing it. Some applications have I/O-intensive initialization phases where large input datasets are read prior to execution. These input datasets vary widely in size and format. For these reasons, computational applications rely on I/O subsystems increasingly. Indeed, nowadays it is clear for application designers that accessing data on external storage is essential to the program's performance.

While the capacity storage of I/O systems, processing power and network bandwidths have been increasing at a comparable rate, the same has not been happening for the bandwidth of I/O systems. Because of this technological trend, in the research community there is a growing interest in the specific area of performance improvement in I/O systems.

Parallel I/O systems have been developed to address the problem of performance. Parallel I/O systems are based on the combination of many individual components (e.g. disks, servers, network links) together into a coherent whole used to provide high aggregate I/O performance to parallel applications. While the main focus for these systems are the performance of I/O systems for high-performance computing (HPC), there are other issues that are equally important. For instance, increasing the availability and maintainability of these systems is critical since without these non-functional requirements there is the risk of creating I/O systems that cannot be widely deployed. Fundamental requirements for I/O system developers are to provide high-performance, reliable, scalable, and easy-to-use I/O solutions for computational science.

There have been a number of significant advances in I/O for computational science, but many challenges still remain, such as high performance with scalability, reliability/fault tolerance, flexible and efficient integration with parallel codes and portability [1] [2]. In this dissertation we will emphasize the need for more flexible I/O systems that can better meet the wide set of application dependent requirements. Specifically, not all the aforementioned requirements can be satisfied independently. The main parallel I/O technique, the disk striping, in principle, can increase the I/O throughput by simply increasing the number of disks. But the problem with the striping scheme is the reliability. If a file is striped over multiple disks, the loss of even one disk could make the whole file useless, and the chance of losing a disk rises roughly proportionally to the number of disks.

Therefore, being the application requirement very different from one application to another it would be advisable that the I/O system lets applications to pick out, in

a flexible way, the suitable trade off between performance and reliability. Indeed, it seems to be impossible to find a single appropriate interface, caching policy, file structure, or redundancy strategy for all parallel applications, so the versatility become more and more the key aspect to better fit the application needs [2]. To this aim, this dissertation describes a novel strategy to enhance existing Parallel File Systems in order to provide flexible mechanisms that lets parallel application requirements to be met. Measurements show the different performance costs associated with various resiliency requirements. The significant trade-offs associated with resiliency and storage mechanism choices underscore the importance of flexibility in storage infrastructures.

Chapter 1

Parallel File Systems

This chapter sheds some light on motivation behind the adoption of parallel file systems in high performance computers. Specifically it firstly introduces the context of high performance computing and then the need for fast I/O systems. The I/O system includes storage devices, interconnection networks, file systems, and one or more I/O programming library. In the Section 1.2, the main characteristics of parallel file systems in terms of operational features and architectural components are discussed. Finally, in the Section 1.3 some of the key challenges for parallel file systems are examined.

1.1 Introduction

The area of High Performance Computing (HPC) has been focused for years on FLOPS (floating point operations/second) and eagerly rate computers in gigaflops, teraflops, but too rarely on data management. The apex of this compute-centric view can be observed in particular during the Cold War, in which the whole research effort was focused on the realization of specialized supercomputers to deliver orders of magnitude greater floating point performance than contemporary mainframes. Subsequently, high end computing had been perceived as having become just too hard, too expensive, and of too narrow interest to justify industry investment in the development of these specialty-class supercomputer architectures. However, most of the science evolution will be governed in their rate of progress by the magnitude of computational performance they are able to engage. Medical science and molecular biology including genetic engineering, space exploitation and cosmology, climate and Earth science, materials including semiconductors and composites, machine intelligence and robotics, financial modeling and commerce are some of the field in which the high performance computing plays a strategic role for the scientific discovery. Fortunately, this conflict between the requirements for high performance and the availability of resources needed to provide it is being addressed through more pragmatic cost-constrained means: commodity clusters.

Commodity cluster is a widely-used term meaning independent computers combined into a unified local computing system through software and networking. A cluster is local in that all of its component subsystems are supervised within a single administrative domain. The constituent computer nodes are commercial-off-the-shelf (COTS) and they may incorporate a single microprocessor or multiple microprocessors. The interconnection network employs fast local area network (LAN) dedicated to the integration of the cluster compute nodes and is separate from the clusters external environment. The low cost computing capability is derived from the mass market COTS PC and local networking industries. High-performance clusters are implemented primarily to provide increased performance by splitting a computational task across many different nodes in the cluster, and are most commonly used in scientific computing. Beowulf Clusters is a special class of cluster based on commodity hardware with open source software (Linux) infrastructure. For the academic community this leads to interesting research and the exploration of new ideas, but also resulted in one-of-a-kind machines. The cost-effectiveness and Linux support for high performance networks for PC class machines has enabled the construction of balanced systems built entirely of COTS technology which has made generic architectures and programming models practical.

1.1.1 Research Trends

While the capacity of external storage devices has been increasing at a rate commensurate with increases in processing power and network bandwidths, the same growth has not been involving the data transfer rate of external storage systems. Specifically, the current data transfer rate delivered by individual disk drive is orders of magnitude slower than the required rate needed by modern parallel computers. The result of this technological trend is that the application performance is primarily limited by I/O transfer rate rather than processing power. Furthermore, many types of high performance applications frequently need to access large dataset. In particular, there are three main categories of I/O-intensive applications that demand good I/O performance: database management systems, multimedia applications, and scientific simulations.

A different characterization, from the access patterns point of view, has been presented by Miller and Katz [3]. They divided supercomputer application I/O in three categories: *required*, *checkpoint*, and *data staging*. *Required I/O* includes reading input data and writing final results. *Checkpoint I/O* is the data a program writes periodically as insurance against a hardware or software failure. If the program halts before completing its computation, it can resume by reading the checkpoint data and restart the computation near where it left off. *Staging I/O* supports applications whose data does not fit in memory. The application transfers subset of large data

structure between primary and secondary storage as the computation proceeds. This last application characterization seems to be too simplistic in that scientific applications often perform more complicated file access patters, such as strided access patterns [3]. Both these characterizations apply to sequential and parallel programs. In particular, parallel programs must also deal with the distribution of data among the processes. Parallel programs do not exhibit standard pattern but usually they perform sequences of small access requests form multiple processes, which standard sequential file systems handle poorly.

In this dissertation we will give more emphasis on issues related to I/O systems for parallel programs because they represent the common case in high performance computing.

1.1.2 The I/O subsystem

One of the major limitation of traditional I/O subsystems, such as RAID arrays, is their limited scalability caused by shared controllers. Emerging parallel I/O subsystems enhance scalability by eliminating the shared controllers and enable direct host access to potentially thousands of storage devices.

For most I/O intensive applications the main requirements are I/O speed and storage capacity even though data integrity and availability are important as well. The performance of those applications is often limited by the speed of I/O subsystems. The reasons can be found at different levels of a parallel system architecture. A typical

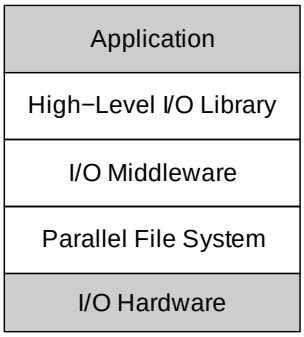

Figure 1.1: I/O software stack

software hierarchy of an I/O subsystem is shown in Figure 1.1. At the lowest level, I/O systems are composed of storage devices and interconnection networks to connect storage devices to the rest of the computer system. The design of storage devices has consequences on the performance of all the upper levels. One of the most important development in the design of storage devices for high performance computing has been the advent of RAID technology. Basic concepts of RAID technology are presented in the next subsection. As for the interconnection network, parallel I/O exploits high-bandwidth (multiple Gbps) and low latency (less than 10μs) of high performance interconnect. One of the most used type of interconnect for parallel computers is the internal message passing network such as Myrinet [4], Quadrics [5]. These networks are designed to allow many nodes to communicate with each other simultaneously. Finally, file system creates the necessary abstraction to make storage devices usable for applications either directly or through the I/O middleware and high-level IO library. File systems are also responsible for maintaining the data integrity. A more

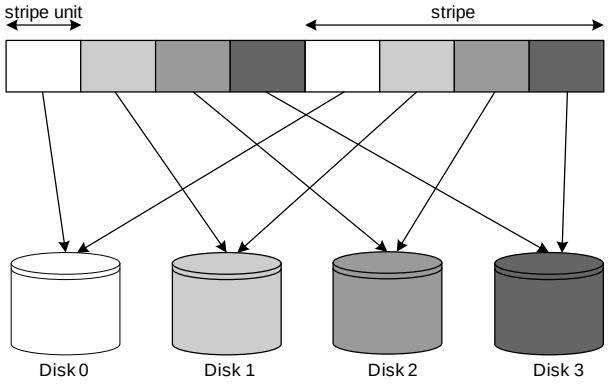

Figure 1.2: Disk striping

detailed description is presented in the Section 1.2.

Disk Striping

A natural solution to increase the I/O performance is to use parallel I/O, just as supercomputers have adopted parallel processing. Disk striping is the main technique to achieve greater aggregate bandwidth. It consists of splitting large amount of data into smaller pieces and spreading them simultaneously across a disk array. The data is generally divided into blocks, and blocks are distributed cyclically across the disks. The number of disk is called the *stripe factor*, and the size of blocks is the *stripe depth* or *stripe unit*. In Figure 1.2, a disk striping with stripe factor equal to four is shown. The stripe factor determines the degree of parallelism and therefore the maximum aggregate transfer rate. The choice of stripe unit size depends on how the disk array will be used. The problem with the striping scheme is the reliability because the risk of losing data is directly related to the number of disks. Since the end of 80's a

research group at the University of California, Berkeley, proposed Redundant Array of Inexpensive Disks (RAID) [6] as a possible solution to improve the reliability level of disk array. RAID today stands for redundant array of independent disks because of the reduced gap between high performance and cheap disks. In the work it has been presented three different techniques for storing redundant information: mirroring, Hamming code, and parity. Those techniques were applied to five strategies, called RAID level, with different performance and reliability characteristics.

1.2 Paralle File Systems

As mentioned before, the I/O subsystem of cluster architectures consists of many disks located in many different nodes. The software that organizes these disks into a coherent file system is called a "parallel file system".

Parallel file systems are similar to traditional network file systems in that they support access to shared files from multiple processes. However, network file systems traditionally do not provide support for striping data over multiple I/O nodes and they are not designed to efficiently support simultaneously accesses to individual files. For instance, in Network File System (NFS) [7], the logical parallelism provided by concurrent accesses to a shared file from several NFS clients running on compute nodes is not translated in a physical parallelism in that the NFS server running on an I/O node serializes all parallel requests as shown in Figure 1.3(a). In parallel file systems,

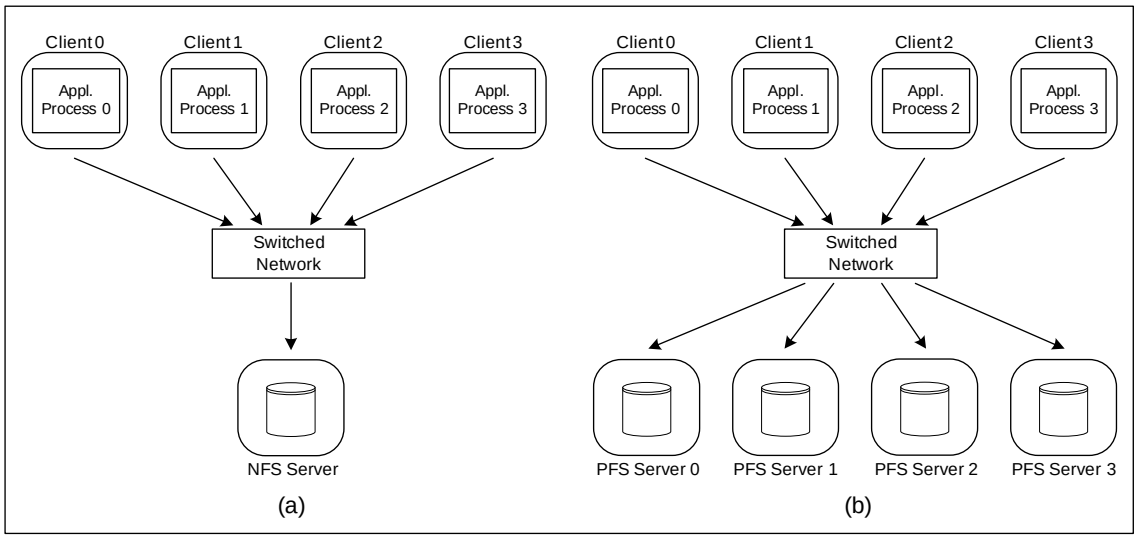

Figure 1.3: Network configuration: (a) NFS, (b) Parallel File System

PIOUS [8], PPFS [9], Vesta [10], Galley [11], PVFS [12], and GPFS [13], the files are typically striped over several I/O nodes as shown in Figure 1.3(b), managed by I/O servers. They are specifically designed to support concurrent accesses of parallel applications sharing data across many clients.

Parallel applications show two main type of I/O accesses: pure sequential access and multiple file access. With the first one, program sends all accesses through a single task while with multiple access each task write its own file. Even though the sequential access allow to easily manage all the data in one file and is likely to produce contiguous file accesses that storage devices and file system can manage efficiently, it presents some important drawbacks. One of these is that a single process may not have enough memory to hold all data that parallel job needs to read or write. Another and more important drawback is that the total transfer rate is limited to what a single node can

support. Multiple file access is an excellent choice especially when each process writes data to a separate file. On the other hand, the time to collect data from separate files into one large file can easily wipe out the good performance of multiple file access. However, parallel file system should combine the high performance and scalability of multiple file access with the convenience of collecting data in a single file. To this aim, it is necessary to allow multiple tasks to access the same file at the same time, even though, generally, different tasks are interested to different locations in a given file. Parallel file systems stripes file across different I/O nodes so it needs to manage the data mapping between multiple compute nodes and the shared file.

The data distribution among the compute nodes may have significant influence on accesses efficiency [14]. Parallel applications have a wide range of I/O access patterns which differ from the file physical layout on the I/O nodes. These different views of the file can cause performance loss due to fragmentation of data on the disks of the I/O nodes and complex index computations to access files. Furthermore, the fragmentation results in sending lots of small messages over the network and this cause contention of related processes at I/O nodes. Usually, the physical distribution of file is determinated when the file is first created. The file distribution is described by the striping parameters:

- the number of I/O servers on which data will be stored;
- the stripe size, the size of contiguous chunks stored on I/O servers;

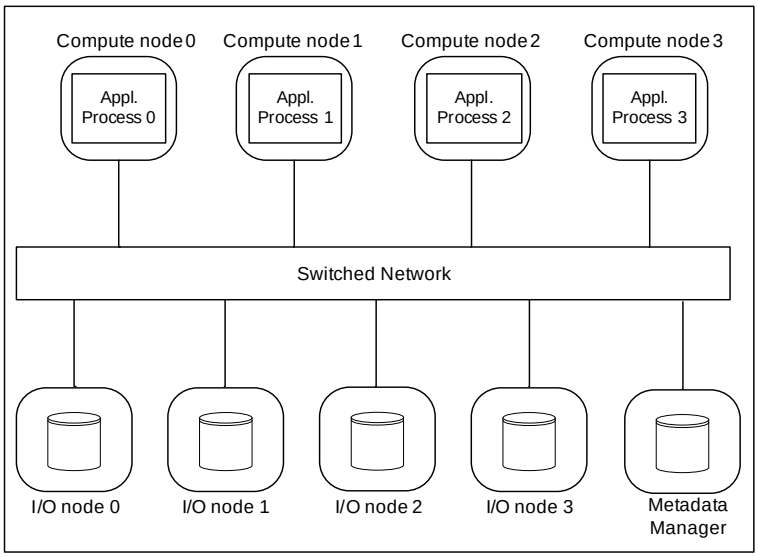

Figure 1.4: General parallel file system architecture

- the first I/O server to be used.

1.2.1 Parallel File Systems Architecture

Parallel file systems architecture have converged toward a general configuration shown in Figure 1.4. The nodes in a cluster are divided into three main components, which may or not overlapped: the compute nodes, the I/O nodes, and the metadata manager. Files are typically striped over the I/O nodes which store single datafile or subfile concerning that file. Applications run on the compute nodes. Each node of the cluster can play the role of a compute node, I/O server, or metadata manager. Usually there is just one metadata manager running on a parallel file system, sometime it is distributed across different nodes. Metadata manager holds information about how data files are distributed. In the next subsection two open source parallel

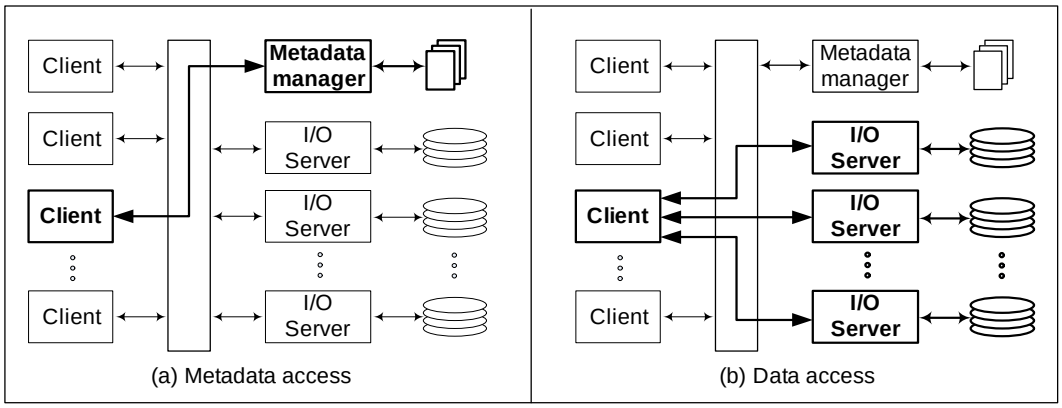

Figure 1.5: PVFS architecture: client/server interactions

file system architectures are presented.

PVFS2 Architecture

Parallel Virtual File System (PVFS) [12] is an open source parallel file system for Linux-based clusters. It has been jointly developed by the Parallel Architecture Research Laboratory at Clemson University and the Mathematics and Computer Science Division at Argonne National Laboratory. Currently, the PVFS version under development is the second one. The PVFS2 architecture reflects the general architecture presented in Figure 1.4. In PVFS2 the node serving as metadata manager runs a demon which manages the metadata information concerning the file, such as the file structure (the partitioning of the file in subfiles, the I/O servers on which the file is written), file size, creation and modification time. I/O nodes run a daemon which stores and retrieves files on local disks of the I/O nodes. When an application on a compute node (client) opens, close or send any request that involves file metadata, it

contacts the metadata manager and obtains a description of the file's layout on the I/O nodes (I/O servers), as depicted in Figure 1.5.a. the data transfer occurs directly between the client and the I/O servers storing the relevant portions of the file, as showed in Figure 1.5.b. Each I/O server stores its portion of a PVFS file as a file on its local file system.

PVFS offers two low-level I/O interfaces that clients commonly use to access parallel file systems. The first of these is the UNIX API as presented by the client operating system. The second is the MPI-IO [15] interface. Parallel applications, by leveraging the ROMIO MPI-IO implementation [16] for PVFS2, can link directly to a low-level PVFS2 API.

Lustre Architecture

Lustre [17] is an open source parallel file system for Linux clusters originated from research done in the Coda project at Carnegie Mellon and currently it is developed by Cluster File Systems, Inc. Three are the major types of software subsystems (see Figure 1.6): the Client FileSystem (CFS), the Object Storage Targets (OST), and Meta-Data Server (MDS) systems. Lustre clients run the Lustre file system and interact with the MDSs to access a file to determine which objects on particular storage controllers store which part of the file and to determine the striping pattern. After obtaining information about the location of data on the OSTs, clients establish direct connections to the OSTs that contain sections of the desired file, and then

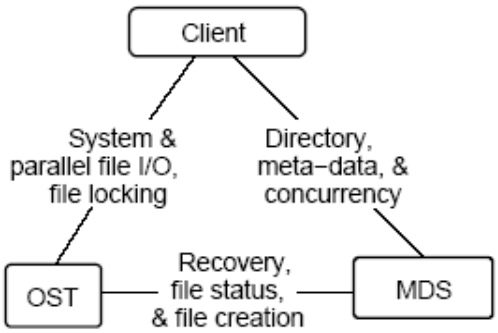

Figure 1.6: Lustre architecture: interactions between systems

perform logical reads and writes to the OSTs which directly interface with object-based disks (OBDs). The OSD specification handles objects or node-level data instead of byte-level data. The Lustre distribution comes with Linux device drivers that emulate OSD in case the hardware vendors do not support OSD. Currently, Lustre provides OBD device drivers that support Lustre data storage within journaling Linux file systems such as ext3, journaled file system (JFS), ReiserFS, and XFS. As part of handling I/O to the physical storage, OSTs manage locking, which allows concurrent access to the objects. File locking is distributed across the OSTs that constitute the file system, and each OST handles the locks for the objects that it stores. The overall architecture of Lustre is shown in Figure 1.7.

1.3 Key challenges for PFS

Modern PFSs scale to very large numbers of clients, handling many concurrent and seemingly independent operations. They present a rich interface on which efficient

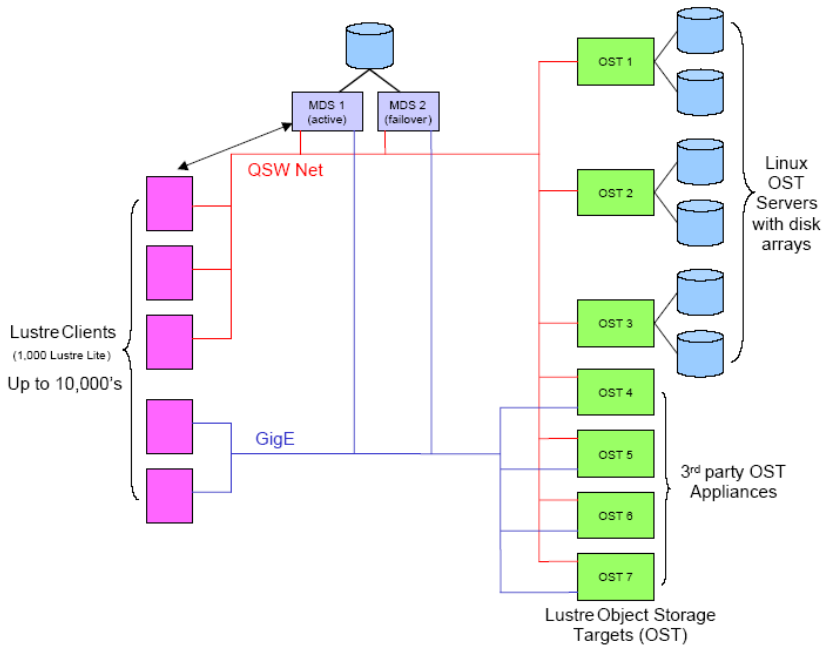

Figure 1.7: Lustre architecture

implementations of higher-level I/O components may be built. While many improvements have had a significant impact on the performance and usability of I/O systems for computational science, many challenges still remain that must be addressed [18] [1]. We describe some of these issues to common PFSs.

1.3.1 PFS Semantics

The POSIX I/O API [19] is the most widely used API for access to file systems, both by applications and by I/O libraries. The POSIX API was originally designed for sequential applications and has many features that limit the performance of parallel applications. For example, POSIX has a strong consistency and atomicity model,

which states that a write from any process is visible to all other processes as soon as the write function returns. Furthermore, if concurrent writes from two processes are directed to overlapped regions of the same file, the result is the data written by either one process or the other, and nothing in between (*sequential consistency*). In a parallel file system, sequential consistency cannot be supported unless a concurrency control mechanism is employed, due to striping of file data. In fact, such a semantic requires communication to coordinate access, for instance by acquiring exclusive access to the range of bytes it accesses. Figure 1.8 depicts the case in which two processes simultaneously access to the same block of data striped on two independent storage devices. One process read the data block while another concurrently write it. This example represents a violation of sequential consistency in that the reading process should retrieve either the previous data before the write or the data completely updated by the write and nothing else. However, most parallel scien-

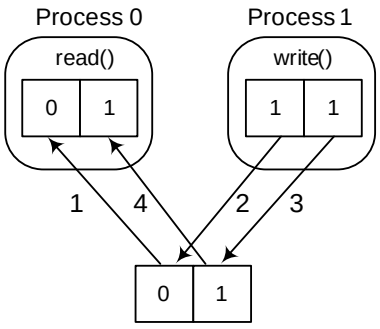

Figure 1.8: Violation of sequential consistency

tific applications do not write concurrently to overlapping regions of the file. If an

optional weak consistency mode were available, such applications could select it for higher performance.

For instance, PVFS2 does not support POSIX semantics. The design intent was to give users a great degree of flexibility in terms of the coherency of the view of file data. PVFS2 provides read and write semantics that better fit with the access and performance requirements of scientific applications. PVFS2 breaks POSIX consistency semantics, which require sequential consistency of file system operations, and replaces them with *non-conflicting writes* semantics. This semantic states that all I/O operations that do not access the same bytes (in other words, are non conflicting) will be sequentially consistent. Write operations that conflict will result in some undefined combination of the bytes being written to storage, while read operations that conflict with write operations will see some undefined combination of original data and write data.

Many parallel applications need to access noncontiguous data regions in the file because of the difference between the data ordering in the file and data distribution among the processes of a parallel program. The POSIX read and write functions allow users to read and write only a single contiguous piece of data at a time. Even though POSIX does have a function that allows the user to specify a list of reads and writes, each read or write is treated as a separate request and for this it would not allow the adoption of techniques to optimize the noncontiguous requests such as data

sieving [20].

An effort to define extensions to the POSIX interface specifically for HPC, to provide more efficient mechanisms for accessing parallel file systems and to improve building-block interfaces for I/O middleware [16] is underway by members of the I/O community from laboratories, universities, and industry. Many of the proof of concepts in this area are being prototyped in the PVFS2 because it is an open, community-driven effort to build a parallel file system specifically for HPC and serves not only as a production PFS option but also as a vehicle for research in parallel I/O concepts.

1.3.2 PFS Management and Reliability

I/O systems are now being used as resources for multiple clusters as well and thus are required to operate over multiple networks and in heterogeneous environments, complicating the implementation and configuration of such systems. Installation, configuration, and tuning of these systems verges on requiring an expert. To address these issues, more intelligence and flexibility is needed in the I/O system, particularly in the parallel file system.

The term used to express concepts such as self-managing, self-tuning in this field is what is known as *autonomic storage*, which represents one example of autonomic computing [21] . The required intelligence to integrate these capabilities can be placed on the servers so that they can exchange information about their health and their

resource utilization. By using this information, the servers could work together toward specific goals, such as balancing data load or maximizing access performance for frequently accessed files. In parallel file systems, integrating autonomic concepts is made more complicated by the need for predictable performance. The additional communication and data movement inherent in autonomic storage could cause performance to vary widely even for identical access patterns. This type of behavior is unacceptable in the tightly scheduled HPC systems of today.

Finally, more effort is necessary to address fault tolerance in HPC I/O systems. The sheer number of I/O devices used in todays I/O subsystems leads to many opportunities for failures to occur. Techniques such as RAID [6] can be used at each I/O server node to hide disk failures; however, as the nodes are complete and independent computer systems, an I/O server 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. To address this problem, in the 1990, Stonebraker and Schloss [22] proposed the use of software RAID (also called distributed RAID) across the I/O servers. Since that idea was presented a number of redundancy techniques have been applied to PFS.

Server failure is not the only issue for a PFS. Client failures are still a significant issue, especially when clients cache data and metadata. When clients fail, many

systems must drop into a recovery mode to determine what was lost before continuing, perturbing performance for the system as a whole. Lustre and GPFS are two example of PFS with this problem. An alternative approach is based on the use of stateless clients. Following this approach, client failures do not cause the loss of important file system data and may be ignored. The PVFS2 system takes a similar approach. Adoption of this approach, or further research into client caching, is necessary to reduce the impact of this last type of failure on the system as a whole. However, in order to develop an appropriate fault tolerance mechanism for parallel file systems, it is critical to analyze data resiliency, file consistency, scalability and performance for any candidate approach. 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. File consistency refers to the significant difficulty in maintaining consistency because of the concurrent accesses from many clients to regions of the same file. While, scalability in this context means that the system is able to deliver scalable performance as the number of storage nodes grows. In the next chapters those criteria are used as the basis for judging the effectiveness of the proposed fault tolerance strategy.

Chapter 2

Dependable Parallel File Systems

This chapter lays the motivation for the thesis and surveys the state of the art in fault tolerant PFSs. A field failure data analysis in high-performance computing systems is provided in Section 2.1.1 with the aim to better understand failures that involve PFS. In fact, the design of highly dependable systems requires a good understanding of failure characteristics. Section 2.2 presents well known techniques applied in common PFS and discusses about arguments for and against of each one. In Section 2.2 a related work study is presented with reference to different research directions. Section 2.3 concludes the chapter by describing the contribution of this dissertation with reference to the related work.

2.1 The Need for Dependability

Parallel file systems are composed by a large number of components combined in a single system through software. Even though the mean time to failure (MTTF) of each component may be very high, a PFS composed of thousands of reliable components will inevitably exhibit frequent failures.

For a better understanding of this simple but important concept, we report an example taken from [23]. Let's assume node failures follow exponential distributions and let R be a single nodes one-hour reliability. Furthermore, suppose the system stops functioning if one node fails. In this scenario, an n-node system's MTTF is approximately $\frac{1}{(1-R^n)}$. Figure 2.1 plots the MTTF for different n's and R's.

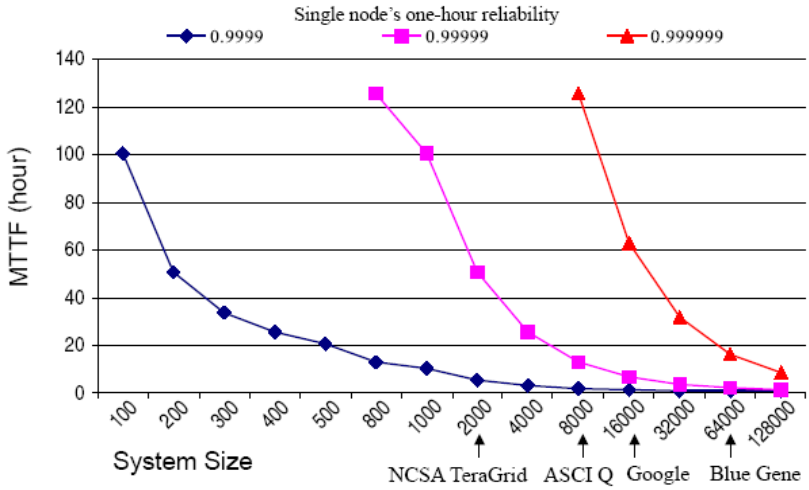

Figure 2.1: Reliability projection for large systems

In addition to hardware failures, PFSs could experience also software failure both in their software component and in the operating system. 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 parallel scientific applications have run times of several days, so to deal with machine unreliability they make use of periodic checkpointing, in which all processes coordinate to dump memory to stable storage simultaneously. The checkpointing technique rely on the assumption that the file system is correctly working so the focus is again on file system reliability.

In order to quantify the impact of failures in modern cluster environments and thus in PFSs, in the Section 2.1.1 some of the results available in the literature are presented. This analysis is necessary to design a strategy with the aim of improving the data resilience of PFS as well as putting some flexibility into the configuration. Data resiliency means the ability to tolerate errors or failures without all of the PFS being unavailable.

2.1.1 Failure Data Analysis

The design of dependable systems relies in many ways on a deep study of what failures in real systems look like. For instance, in order to improve cluster availability with effective resource allocation or to pick up the best parameters for checkpoint strategies [24] the knowledge of system failure characteristics is crucial. Creating realistic dependability benchmark also requires a clear understanding of real failure characteristics. Unfortunately, little field failure data on of modern high-performance computing systems is publicly available, due to the confidential nature of this data.

Existing studies on failures data analysis are often either based on short period [25] or are dated [26] so that it is no more representative of modern computer systems. Moreover, none of them make the raw data available for the research community.

Recently, the Los Alamos National Laboratory (LANL) has decide to public the own raw data on failure [27] . This data has been collected over the past 9 years and it covers 22 high-performance computing (HPC) systems, including a total of 4750 machines and 24101 processors. A first work [28] has been published on the statistical properties of the collected data. According to the analysis performed in [28] on raw data, the hardware is most common root cause of failure, with the actual percentage ranging from 30% to more than 60%. Software is the second largest contributor, with percentages ranging from 5% to 24%. For all 22 systems, memory was the single most common low-level root cause. The most common software failure is strongly dependent on the type of system, for some of them it is related to the the parallel file system, whereas for others it is related to the scheduler software or the operating system. Other interesting outcomes of the analysis are relating to the failure rate and repair rate. Even thought the analyzed cluster systems vary widely in size, a normalized analysis has proved that failure rates do not grow significantly faster than linearly with system size. Rather than with system size, failure rates vary significantly depending on the workloads running on every nodes as observed in previous works [29]. The average number of failure per year normalized by number of processors in the

system is around 0.3. Considering that each node in those systems has at least two processors the total number of failure (independent of root cause) per node is more than 0.6 per year. As concern, the mean repair time across all failures (independent of root cause) it is close to 6 hours. This value is dominated by hardware and software failures which are the most frequent types of failures even thought the repair time vary significantly depending on the root cause of the failure.

In a report concerning operation of the ASCI Q system at LANL, Morrison stated that file system is main source of failure for the ASCI Q system [30]. The troubles include loss of data on local scratch disks, considerable impact of local disk failures on the whole system (many hung services require whole machine reboot, which takes 4-8 hours), and occasional unavailability of files.

With the mentioned failure characteristics, and considering that these systems are designed to efficiently support complex scientific applications with long execution, multiple failures should be taken in serious consideration. These observations point out the need for versatile fault tolerant strategies to cope with different simultaneous component failures in the system. In this dissertation the emphasis is placed on the needs for high availability of the parallel file system that is also essential as a basis for effective implementations of further fault tolerant strategies such as checkpointing.

2.2 Related Work

In this section we will present the main fault tolerant mechanisms adopted in parallel file systems and how they differ from the strategy described in this dissertation. The related works are organized according to the different approaches they have adopted to accomplish the increasing need for high availability in current and future parallel file systems. Although parallel file systems are devoted to the performance, more and more attention has been placed on the system availability and specifically on scalability of the fault tolerant strategies as substantiated by the high number of works. In the rest of the dissertation we will refer indifferently to resiliency and dependability associated to a file configuration as the ability of PFS to tolerate simultaneous failures without file corruption.

2.2.1 Fault Tolerant PFSs

Parallel file systems for their own characteristic of striping data over different I/O nodes in the cluster are prone to I/O node failures. For this reason, most of the PFSs adopts a fault tolerant strategy to cope at least with this type of failure. Distributed RAID has been the mostly used technique to improve reliability in storage systems. Examples of fault tolerant file systems are Google file system [31], Panasas' ActiveScale Storage Cluster [32], Lustre [17], Petal [33], xFS [34], CEFT-PVFS [35]. Each of these storage systems, commercial systems or prototypes, hard-codes most

choices about the manner in which objects are stored. For example, Petal replicates data for fault tolerance, tolerates only I/O node crashes (i.e., fail-stop nodes), and uses chained declustering to disperse data and load across nodes in the cluster; these choices apply to all data. xFS also uses one choice for the entire system: parity-based fault tolerance for server crashes and data striping for dispersing load. CEFT-PVFS, a parallel file system that extends the original PVFS, combines striping with mirroring by first striping among the primary group of storage nodes and then duplicating all the data in the primary group to its backup group to provide fault tolerance for server crashes.

HP AutoRAID [36], CSAR [37] [38] and RepStore [39] are example of versatile storage systems. RepStore offers two erasure codes (replication or erasure coding) rather than just one and uses AutoRAID-like algorithms to select which to use for which data in an adaptive manner. AutoRAID automates versatile storage in a monolithic disk array controller. In fact, it provide solutions meant to be used in centralized storage controllers and for this reason it cannot be used directly in a cluster storage system. CSAR is a prototype based on PVFS that dynamically switches between RAID1 and RAID5 redundancy based on write size. Even thought CSAR presents a form of adaptation, it is not meant to be versatile with respect to the application requirements.

Differently by those fault tolerant PFSs the proposed strategy allows the application to select the suitable level of fault tolerance for each file it creates and manages through the use of erasure codes. We claim that this flexibility is necessary for PFSs in order to accomplish the different needs that characterize each single application running on such a shared environment.

Furthermore, the improvement on data residency obtained with erasure codes can lengthen the time between system maintenance. In fact, in current storage systems, management operations are either done manually after taking the system off-line, use a centralized implementation [36], or assume a simple redundancy scheme [33]. The paramount importance of storage system throughput and availability leads to the employment of adhoc management techniques, contributing to annual storage management costs that are 6-12 times the purchase cost of storage [40].

2.2.2 Client-based vs Server-based Redundancy Management

Another prospective to review the related work on availability improvements of PFSs is the different approach adopted to manage the redundancy. More specifically, related work can be divided according to two choice to place the responsibility of redundancy management on client side or on server side. With the client-based approach, clients during a write instead of sending just the data to I/O servers, they have to send also redundancy data to the I/O nodes that should hold that redundancy. One problem with this approach is that the client processes are now responsible for ensuring data

consistency both during concurrent updates and in failure scenarios.

GPFS [13], the general parallel file system, adopts a client-based approach to manage the redundancy. It provides support for software-based data replication and online fail over. In GPFS, all write requests first obtain a *lock token*, and then the client sends the data to two separate storage nodes. GPFS uses a sophisticated distributed byte range locking mechanism with support for two-phase locks and lock timeouts to achieve sequential consistency. The global lock manager coordinates locks between local lock managers by handing out *lock tokens*. Repeated accesses to the same disk object from the same node only require a single message to obtain the right to acquire a lock on the object (the lock token). Only when an operation on another node requires a conflicting lock on the same object are additional messages necessary to revoke the lock token from the first node so it can be granted to the other node. In conjunction with the locking subsystem, GPFS uses a heartbeat system to detect storage node failures. Heartbeat systems exhibit considerable complexity in determining the timeout threshold for heartbeat messages.

The other possible approach that has the advantage of not being dependent on the client to manage redundancy is the server-based approach. The client send only the request to the I/O servers and then the servers communicate with each other to rearrange the redundancy. Currently, a server-based approach with replication scheme is under study at the Clemson University [41]. A combined solution is represented by

CEFT-PVFS [35], a PFS that provide resiliency trough replication as shown in Figure 2.2. It provides four duplication protocols, two of them with server-based approach and other two with client-based approach. CEFTPVFS is similar to GPFS in that it employ the two-phase locking to ensure that writes are properly serialized and an heartbeat monitoring system to provide consistency and resiliency.

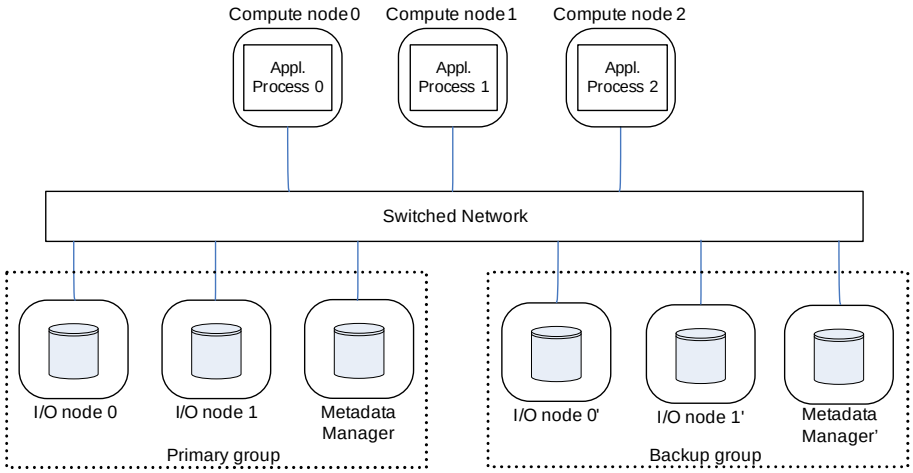

Figure 2.2: CEFT-PVFS architecture

Even though server based approach lightens client nodes it complicates a lot the design of server nodes by overwhelm them with additional computation to obtain the redundancy and communication to distribute it among the servers. This last consideration combined with the choice to adopt the erasure codes as redundancy scheme has motivated the adoption of a client based approach for the design of our dependable PFS. This also follows the the well known principle of shifting work from servers to clients to increase scalability [42].

In comparison with the GPFS, we do not provide sequential consistency but only

non-conflicting consistency, so the mechanism we use to provide data consistency during concurrent writes is much simpler and does not require a centralized lock server. We use a *device-served locking* mechanism [43] only for write operations that reduces the cost of a scalable serializable storage array by eliminating the need for dedicated lock server. Further details of adopted locking mechanism are presented in the Chapter.

There is also another classification of the redundancy strategies that is orthogonal to the above one. This classification refers to the schedule of the redundancy update. Two are the main strategies:

- *lazy redundancy* (redundancy written on file close)
- *commit redundancy* (redundancy written after write completion)

With respect to this further classification the proposed strategy complies with the *commit redundancy* in that for each file modification the client updates the redundancy. With the lazy redundancy a server failures between the open and close operations on a file can lead to the loss of file updates because the redundancy is not consistent with data until the closing. The decrease in reliability associated with this approach depends upon the application behavior in that the longer is the time to close a file the greater is the chance that a failure occurs before it. This drawback can be obviated with the *commit redundancy* but with an overhead due to frequent computations of new redundancy. This overhead is proportional to the amount of data to

write (i.e. application workload) and it depends strongly on the type of redundancy adopted (i.e. replication, parity, erasure codes). We choice to adopt the *commit redundancy* approach because it provides higher reliability than *lazy redundancy* and we believe that this last one is more suitable to be integrated in higher levels of the I/O software stack (see Figure 1.1) rather than in the PFS layer as proposed in [44].

2.2.3 Concurrency control

In the context of storage system for cluster three are the main source of problems for consistent access:

- access concurrency;
- servers failures;
- client failures (resulting in partial or corrupt updates).

Many protocols have been proposed in literature to address various mixes of these problems. Lamport in the '86 proposed atomic register [45] to permit multiple readers and multiple writers access. In most storage systems access concurrency is addressed via leases [46], optimistic concurrency control [47], or serialization through a primary [33]. Partial writes by clients that fail can be addressed by two-phase commit [48] or by post-hoc repair (in systems using replication). A further consideration is necessary as concern the context of parallel applications. In fact, while in

distributed storage systems concurrency is uncommon [49] [50] in parallel file systems usually more processes try to access shared regions of the same file. For PFSs that do not make use of redundancy and that adopt relaxing semantic such as the *nonconflicting writes* semantic used in PFS2 (see section 1.3.1) concurrent overlapping writes do not represent an issues for the PFS consistency. When the redundancy is added, concurrent writes can lead to inconsistency between data and redundant information. In the context of shared redundant storage, consistency means that redundant storage blocks contain data that correctly encodes the corresponding data block values Figure 2.3 depicts a case where two clients are writing on contiguous regions of the same file stored with a RAID-5 fashion. Each client is writing to a separate region,

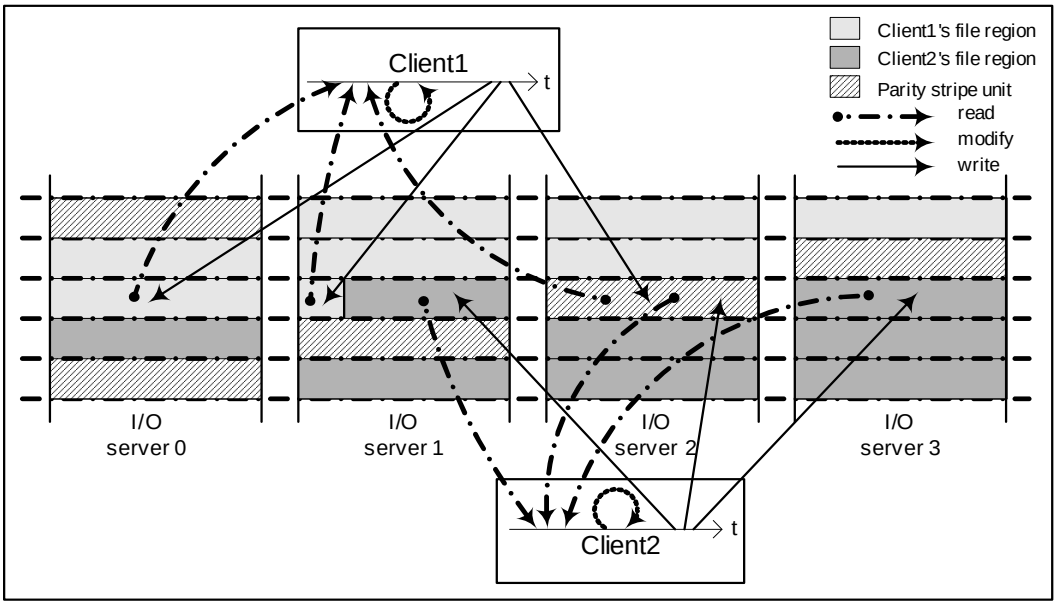

Figure 2.3: Concurrent writes with a shared stripe

but part of those regions happen to be in the same RAID stripe, thereby sharing the

same parity stripe unit. Both clients pre-read the same parity stripe unit and use it to compute the new parity. Later, both clients write data to their independent regions but overwrite the parity stripe unit such that it reflects only one clients update. The final state of the parity stripe unit is, therefore, not the cumulative XOR of the data unit in the stripe. A subsequent failure of a data disk, say I/O server 0, will lead to reconstruction that does not reflect the last data value written to that server. In general, races can occur between concurrent client accesses, or between concurrent accesses and management operations such as data reconstruction. The same consistency problem occurs with the other redundancy schemes. In general, a write on a partial stripe is composed by three operations that should be executed atomically with respect to the other accesses in order to prevent consistency issues:

- a read of the missing data on the partial stripe;
- compute the redundancy information;
- a write of the data and redundancy information.

Another source of inconsistency is the client failure during a write. Although client failures during write operation are not very common, a PFS should, however, guarantee the correct semantic in these cases.

To provide sequential consistency in PIOUS [8] a parallel file system developed at the Emory University, a transaction-based mechanism has been adopted. It has been

called *volatile transaction* and derives from the standard transaction mechanism but with reduced functionality to be optimized for concurrency control in PFS. A volatile transaction, differently by standard transaction mechanism, does not guarantee a consistent state in presence of failures. The implementation of this mechanism requires that the I/O daemon running on I/O nodes be augmented with a scheduler to prevent access conflicts. The scheduler adopted in PIOUS is based on *strict two-phase locking* (S-2PL). The concurrency control proposed in this dissertation has a similar approach but with substantial differences. The first one is the purpose of its adoption: here we do not want to provide a sequential consistency but simply a *non-conflicting consistency* guaranteeing the correctness of redundancy. The second difference states in the management of locks, the client during a write operation acquires locks on each I/O nodes and after that it starts to transmit its data to the I/O nodes. Further details about our concurrency control are discussed in the Chapter3.

In the storage cluster called Sorrento [51] a further concurrency control scheme has been adopted: a *version-based data consistency model*. It uses the standard two-phase commitment (2PC) to ensure the atomicity and a customizable replication on the file basis to fit the reliability requirement of different applications. Sorrento can detect server failure trough heartbeat message and support addition and departure of server nodes. Unfortunately this approach works well only under the assumption that the application exhibit low write-sharing patterns. In a recent work [52] has

been adopted an approach very similar to Sorrento but with the differences that it employs both replication and erasure codes and the write-lock lease to synchronize update operations among simultaneous commitments. In comparison with our solution it does not adapt resources to meet dependability goals in a multi-application environment. However, it would be interesting, in a future work, to compare these two storage systems with our prototype for different access patterns.

2.3 Open Challenges

The costs of data unavailability depends on many factors, some of them are: application requirements, type of file, time to repair, cost of management. Choosing a proper data protection techniques for designing dependable storage solutions for multiple applications in shared environments is a difficult task. There are numerous approaches for protecting data and allocating resources but there is no one that works well for all the type of applications. Storage system designers typically use ad hoc techniques that often are expensive or under-provisioned.

In contrast, in this dissertation we explore the possibility to integrate erasure codes in PFSs with the aim of providing both higher availability and flexibility to better meet dependability goals in a multi-application environment. Even though erasure codes represent a well known redundancy technique especially in the field of communication systems, recently they have been extensively used for distributed file

systems [53] [54]. To the best of our knowledge there is no work on the evaluation of those schemes in parallel environments. In this dissertation we outline the need for versatile PFS in which the resiliency degree associated to each files it manages is postponed up to the file creation time. In this way, each application can select the specific performance and resiliency trade-off to be associated for each file it creates.

We discuss design and implementation choices associated with the redundancy management. Furthermore, we evaluate the implemented prototype with different system configuration (i.e., both different resources and number of tolerated faults) and under different system status (i.e., number of failed server nodes).

Chapter 3

Analysis and Design of the Proposed Strategy

Typical faults that can affect a file system data include hardware and software faults on server nodes or compute nodes, network faults, unplanned power outages. The design of a data redundancy mechanism for parallel file systems should provide data resiliency in the face of these faults. Furthermore, the number of simultaneous faults that can occur before data loss is an important factor to take into consideration especially for large systems. To this end, we propose a flexible redundancy scheme by using the erasure codes that gives versatility to parallel file system allowing the application to specify the own level of data resiliency for each file it creates on the basis of available resources.

Another critical factor in the design of a data redundancy mechanism is the degree of data consistency it provides and the associated management overhead it involves. Obviously, for those PFSs the data consistency should be preserved both during concurrent updates on shared files and after system faults. The concurrency control scheme we adopted is a form of two phase locking that prevents conflicts from concurrent writes. Additionally, a heartbeat mechanism has been used to detect and recover from both client failures during a write and I/O node failures.

Finally, a data redundancy strategy for PFSs must entail acceptable overhead on system performance under common access patterns. Moreover, the additional mechanisms introduced in the PFS to manage the redundancy should be designed in a way that the resulting system still provides scalable performance. Scalable performance is the well known notion that as computing resources are added, system performance improves.

This chapter provides a description of the proposed strategy with reference to the two main objective of this dissertation: PFS dependability improvement in terms of file resiliency (expressed as number of I/O node failures before data loss) and the adoption of flexible strategies in order to use available resources more efficiently. The flexibility characteristic for a PFS plays an important role in supporting specific application requirements such as file availability. For example, some application might be more interested in availability for a particular file rather than in performance

whereas for another file the same application might desire high performance access to the detriment of file availability.

This strategy has been implemented in a prototype, namely DePFS, in order to evaluate performance cost and scalability, achieved by this approach. The DePFS prototype has been developed on the basic architecture of PVFS2. PVFS2 is a fully functional parallel file system that represents an excellent platform to use for adding data redundancy extensions by means of its modular design. The adoption of redundancy in PFS arises three categories of issues in the management of accesses: access concurrency, servers failures, and client failures. Even if access concurrency does not represent a treat for PFS that does not support sequential consistency, it can lead to data inconsistency for redundant PFS. As regards client and server failures, while server failure needs to be carefully managed in order to assure the service continuity, client failure can result in partial or corrupt updates.

3.1 DePFS Architecture

DePFS is an modified version of PVFS in which several mechanism have been introduced to provide a flexible management of redundancy. In Figure 3.1 the DePFS architecture is depicted with reference to its system components and to the software layers. In PVFS2 there are two user level-interfaces available: a kernel interface and an MPI interface. The kernel interface is realized by a kernel-module integrating

PVFS into the kernel Virtual Filesystem Switch (VFS) and a user-space daemon which does communicate with the servers. PVFS2 is specially designed to provide an efficient integration into MPI-2, which is an interface standard for high performance computing. An ADIO device for ROMIO (an MPI-IO implementation for PVFS2) links directly to a low-level PVFS2 API for access.

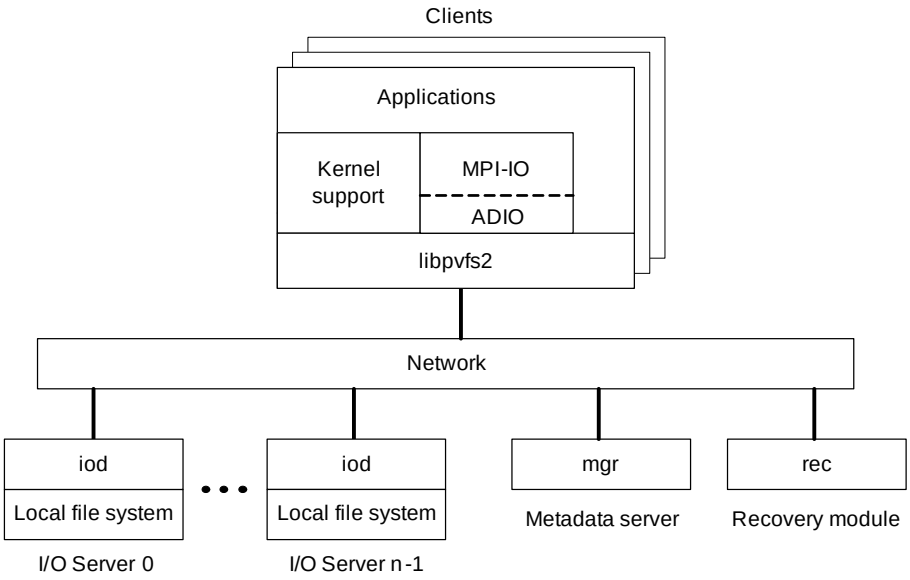

Figure 3.1: DePFS architecture

In DePFS we exploit the tight coupling of PVFS2 with MPI-IO to allow MPI parallel application to specify the desired level of resiliency for each file. Specifically, we modified the ADIO module to support the management of new labels in which are specified the configuration parameters that identify the resiliency level (for further details see Section 3.2.2). Compared with PVFS2, DePVFS presents an additional component called *Recovery module*. This component is responsible for detection and

recovery from some of the system faults. In order to justify all the implementation choices adopted in the DePFS design, first of all we present the assumed fault model. Subsequently, we present all the mechanisms introduced in the original PVFS2 architecture. These mechanisms allow to tolerate system faults ensuring file system consistency and availability. PVFS2 does not mask server failure, it simply continue to serve requests by using the surviving servers. However, any application that was reading data from that server will not be able to retrieve data and an error will be raised. In DePFS this situation will be transparently managed by the system and the application will continue to properly use the file system.

Even though in PVFS2 a client failure does not represent a treat, in DePFS, data inconsistency could be a problem since data redundancy is introduced. For this reason, client failure handling has been considered in the DePFS design.

3.1.1 Fault Model

The DePFS data access protocol has been designed to provide consistency under the assumption described by the fault model summarized in Table 3.1.

	Specification
Timing model	Synchronous
Server failure	Crash failure
Client failure	Crash failure

Table 3.1: Fault model

The timing assumption we used in our protocol are based on the synchronous

timing model, in which known bounds are assumed on message transmission delays between correct clients and servers, as well as the execution rates of clients and servers. This model introduces a strong assumption on the system behavior because it imposes a specific upper bounds in the client-server interaction, irrespective of the system workload. However, cluster environment are characterized by dedicated high performance interconnect such as Myrinet and Quadrics that guarantee high-bandwidth (multiple Gbps) and low latency (less than 10s). As server and client failure model we assume crash failures, in which some nodes (clients or servers) crash and never respond to request again. In the synchronous timing model, crash failures can be reduced to fail stop failures, in which servers detectably crash [55]. For a specific file stored on DePFS with a *k-of-n* code (more details about codes are drawn in the next section), *n-k* server crash failures can be tolerated without data loss, whereas the number of tolerated client failures is not bounded. As regards network failures, they are not modeled directly because most failures can be mapped to a server failure model. A further assumption has been taken on the system. It concerns the Metadata Server that holds all the information about the files stored in a PVFS2 file system (for example, creation time, owner, group, size, etc.). In our fault model we do not take into account Metadata server failure but we concentrate only on the failure mentioned above. The reason for this choice is illustrated in the next section.

Additional Assumptions

In the design and analysis of DePFS we make some assumptions on two components of the architecture: the Recovery Module, presented later in the Section 3.2.1, and the Metadata Manager. The Recovery Module is a stateless component because it does not hold information concerning the state of the system, and thus it can be easily replicated. As concern the Metadata Manager, it is possible to use well known techniques such as passive replication, also called primary-backup approach [56], in order to replicate it in a consistent way. For those reasons we concentrate our attention on fault treatment of the other system components.

3.1.2 Redundancy scheme: Erasure Codes

Erasure code-based schemes, also called Erasure Codes (EC) or *k-of-n* code, divide a file into k data fragments (equal size) and produce n encoded fragments (of same size as before) usually called a stripe. The key property of EC is that the original file can be recovered from any k out of the n fragments. The storage overhead in this case is the ratio of total number of fragments to the original number of fragments n/k. Systematic maximum distance separable (MDS) erasure code is a particular type of code with the peculiarity that the initial k data fragments are the same also after the encoding process, and thus only the n-k fragments comes out from the encoding process (systematic property). In the later sections we will always refer to this last

type of EC as simply erasure codes. Systematic property will give us the possibility to significantly improve performance in PFS's read operations. Whole file replication and parity-based code such as RAID-5 may be considered trivial *1-of-n* and *(n-1)-of-n* erasure correcting schemes. The availability analyses computed on a model for component reliability [57] show that file availability can be many orders of magnitude higher than replication with similar storage overhead.

With reference to the PFS scenario, by striping files, according to the above scheme, across n separate storage nodes, the file system can survive to n-k simultaneous node failures and any k file fragments can recover the original k files. The basic idea consists of choosing for each file the value k in order to vary the level of resiliency associated with it. An example is shown in Figure 3.2, where three different configurations are selected for each one of the three files (i.e., file *FA* and *FB* encoded with erasure codes respectively *4-of-6* and *3-of-6* and file *FC* with simple round robin striping). Each of them has the same size (twelve times a stripe unit size) but each one has different storage overhead and resiliency associated with. Even though the stripe unit size is the same for the three configurations, they impose different relationships among the data stripe units. For instance, for the file FB the encoded stripe units CB0, CB1, and CB2 depend on the data stripe unit FB0, FB1, and FB2 through the encoding process. The choice of adopting the erasure codes to store the redundancy is mainly due to four reasons:

- the excellent ratio between space used and fault tolerance;
- the data is stored explicitly (e.g. systematic property);
- relief from failed storage device identity;
- fine tunable level of data protection.

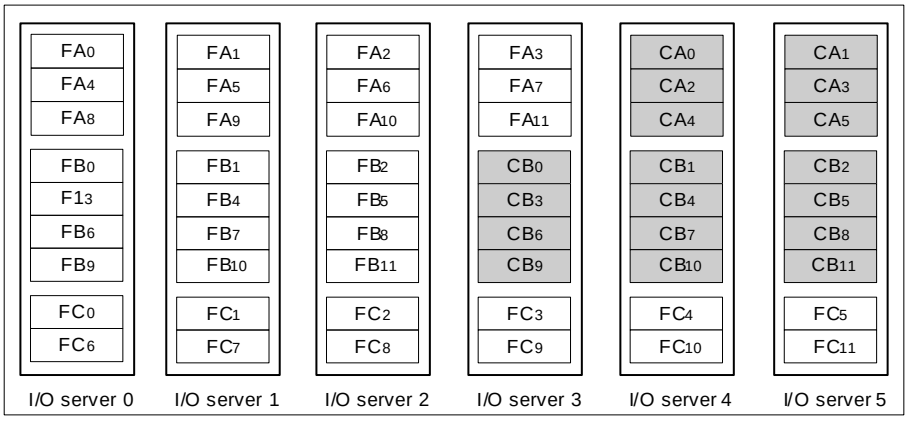

Figure 3.2: File striping with erasure codes: file FA, FB, and FC with code respectively 4-of-6, 3-of-6, and 6-of-6 (no redundancy)

The first property allows to use the available storage space of the cluster in a more efficient way respect to the well known RAID-like techniques. The second property enables to optimize the reading of data in the normal operation mode (e.g. without failures there is no need to decode data). The third property is useful during a node failure, in fact in this case, we can reconstruct the missing data from the ones stored on k of the remaining nodes. The last property refers to the opportunity provided by the erasure codes to select arbitrarily values for k under the condition $k \leq n$ in order to choose the suitable level of fault tolerance. It is worth noting that the improvement in file resiliency that can be obtained with a configuration rather than another is paid in terms of a reduced I/O parallelism because of the use of more storage nodes for encoded data in spite of explicit data. However, even though the reduced parallelism decreases the I/O throughput (as happen in almost all the redundancy schemes, see for example Figure 2.2), the main drawback that arises from the introduction of erasure codes is the computational overhead due to the data encoding and decoding process during respectively write and read operations. This last factor depends strongly on the specific erasure code adopted.

Reed-Solomon (RS) codes [58] are considered quite expensive because of the computational cost of both encode and decode operations which grow rather quickly with k and n. Moreover, encoded information is generated through operations on a Galois field [59]. The computational cost of this process is related to the size of the field, where typical RS implementations operate in a field of size 2^8, or one with 255 elements. For this code, a linear increase in the depth of the field results in an exponential increase in computational cost. For these reasons they are usually deemed appropriate only for limited values of k and n. One approach to get around this limit is to segment the data into a number of mini-blocks and separately apply RS coding to each mini-block. While high throughput can be achieved for encoding and decoding, this is less space-efficient than other alternatives. In contrast to RS codes, *irregular codes* such as *Low-Density Parity-Check* (LDPC) codes [58] (including *Tornado* codes [58]) provide probabilistic erasure correction in the sense that the amount of data that must be acquired to perform decoding is larger than with RS. This class of codes provides a significant reduction in computational cost for encoding and decoding, but with a reduction of storage efficiency and the loss of the deterministic behavior.

For our prototype we choice to adopt an efficient implementation of RS [60] by preferring the strong properties they guarantee instead of lighter codes such as irregular ones that exhibit characteristics not suitable for integration in PFSs. Currently we are investigating the opportunity to further improve the implementation of those codes to reduce its computational cost.

Data Distribution

PFSs exploit data striping across several I/O nodes to achieve greater aggregate bandwidth. This allows PFS's clients to access all the I/O nodes in parallel, reducing the

amount of time required to perform reads and writes operations. The data distribution (i.e., mapping between the logical file and the physical storage) and the adopted redundancy scheme might impact the throughput on the client side directly. For PFS data redundancy can be used just to protect data and not to improve performance. Differently by what happen in distributed storage systems, where encoded fragments can be used in spite of explicit data to speed up the reconstruction of the whole data, in PFS network behavior is much more predictable and thus explicit data are always preferable to encoded data in fault-free scenarios. From the previous consideration, it is easy to figure out that the read performance in the fault-free scenario decreases when the number of I/O nodes dedicated to encoded data increases due to reduction in parallelism. The same consideration can be done for the write operation. The only difference between read and write operations regards the computation associated with them. While for the read, in a fault-free scenario, there is no need of decode process because the data are stored in explicit way on data I/O nodes (systematic property of erasure codes). For the write, the encode process is always necessary in order to update redundancy information on the encoded I/O nodes. This combination of data distribution and redundancy scheme allows to achieve read performance that is independent by the decode process. While write performance is still dependent by the overhead associated with encoding process.

3.1.3 Concurrency Control Mechanism

As stated above, PVFS2 does not provide data resiliency, so if an I/O server nodes experiences a failure all the file striped on that node will be unavailable or definitively lost. We build upon the PVFS2 architecture to develop robust protocol for transmitting data and redundancy among PVFS clients and I/O server nodes. For the redundancy management we used a client-based approach in which the client processes are responsible for ensuring that the redundant data is kept consistent in

spite of both concurrent accesses and I/O server failures. However, with this approach a client becomes now itself a potential source of data inconsistency. In fact, if a client fails during a write operation it can leave the data not consistent with its redundant information. The detection of this error is left to a new system component we introduced in DePFS called *Recovery Module*.

DePFS has been designed to exhibit the same semantic of PVFS2 (*non-conflicting writes* semantic) but with the additional task of keeping redundancy consistent. Due to the performance demands placed upon parallel file systems, PVFS2 read and write requests can proceed simultaneously. Although this allows PVFS2 to provide the maximum parallelism it does pose severe difficulties for a file system with redundancy. For instance, simultaneous overlapping writes are not serialized so that only each individual byte of the overlapping region will contain data from one of the competing writes. Consequently, without guarantee on which data will be written to a given stripe unit for two concurrent and overlapping writes the redundancy will not be consistent with data. For this reason, we modified the PVFS2 write protocol in order to prevent conflicts from concurrent writes. It is worth noting that these consistency issues can occur even if the application processes running on the client nodes are participating in an application-level concurrency control protocol because the stripe organization of PFS imposes an hidden relationships among the data stripes that should be preserved (see Figure 3.2).

Data Consistency on Writes

The concurrency control scheme we employed to ensure data consistency in DePFS is based on two phase locking protocol. This locking protocol does not make use of centralized server but it exploits the I/O servers (i.e., storage device) to serves locks with a granularity of stripe unit and for this reason is called *device-served locking* [43]. While widely-parallel locking like this can introduce many lock and unlock messages,

this one mitigates this somewhat by piggy-backing these messages on I/O requests as summarized in Figure 3.3. The physical layout of a generic write operation on

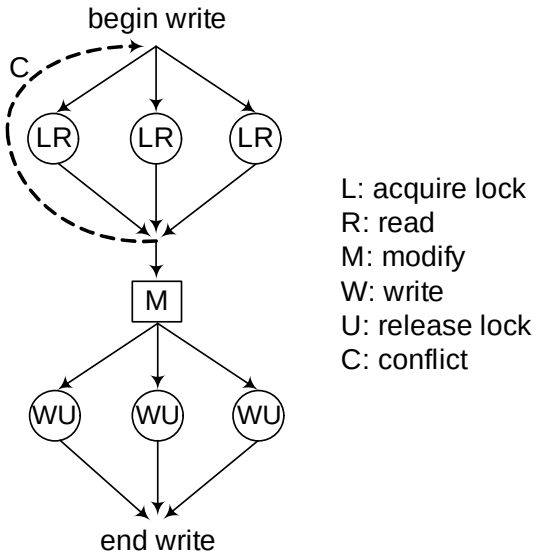

Figure 3.3: Concurrency control scheme: device-served locking and the piggy-backing optimization

the PFS can be represented as composed by an initial partial stripe followed by an integer number of full stripes and finally with another partial stripe. Only in rare cases ($write\_size = k * stripe\_unit\_size * i$, and $write\_offset = stripe\_unit\_size * j$, where i,j are two integers) it is composed by only full stripes. Every times in the write there is at least a partial stripe, a *pre-read* operation is needed to complete the partial stripes and compute the encoded stripe units (*read-modify-write* operation). From this point on, the client has the new encoded data to complete the write, transferring them to the I/O nodes. Seeing that the *pre-read* requests are almost always necessary in the write, we optimize the lock acquisition phase by piggy-backing the lock requests onto the *pre-read* requests as depicted in Figure 3.3. Furthermore, all the locks are released during the second I/O phase by piggy-backing the unlock messages onto the write requests. This write protocol has no latency overhead associated with locking.

Only concurrent conflicting write may incur additional phases of communication.

The protocol requires that after a client sends all the lock/read requests to all the servers it waits until all data has been retrieved. A negative acknowledgment is sent back to the client if another client has already locked the requested stripes. In this case, the lock acquisition phase must be restarted by releasing eventually the already acquired locks for those stripe in order to prevent deadlock. Otherwise, if all the servers reply with data to the lock/read requests it means the lock acquisition phase has been completed successfully and the data encoding can take place. Each read/lock request is processed by server in order to verify possible conflict with previous locked regions. Each acquired lock is stored on a table locally to each server. The entries of this table contain offset and number of locked stripes , and timestamp representing the actual time of the request as shown in Table 3.2.

Offset	**number of stripes**	**timestamp**
...	...	...

Table 3.2: Fields for each entry of the table held by I/O server

As soon as the entry has been stored, the server replies to the client with requested data. When all the data is gathered, the client starts the second phase by issuing write/unlock requests to the servers. Each server commits data on the disk, releases the lock and replies with an acknowledgment to the client. It is worth noting that even though a lock mechanism can severely limit concurrency, a key factor to reduce its impact on the system performance is represented by the lock granularity. In our implementation, clients lock only the strictly number of stripes necessary to cover the file region they are going to write and immediately after the write they release the locks.

Besides the locking infrastructure, it is fundamental to design a proper scheme for failure handling. The failure detection scheme is based on an heartbeat protocol.

A special module is responsible for monitoring the server status. On the basis of the information stored on the I/O server nodes, this module is in charge to detect both orphan locks due to interrupted write operations and I/O server failures.

3.2 System Components

The DePFS prototype is based fundamentally on the PVFS2 architecture but it incorporates additional mechanisms and components with the aim to furnish flexibility in the selection of file resiliency. Some of the new mechanisms we introduced in DePFS have been presented in the Section 3.1.2 and Section 3.1.3 whereas in the Section 3.2.2 the mechanism to allow applications to select their desired file resiliency trade off will be presented. As regards additional components, in DePFS we have introduced a special client called Recovery Module that is in charge of detecting and recovering from client and server failures.

3.2.1 The Recovery Module

DePFS introduces a special client, namely the Recovery Module (RM), which is in charge of detecting server node failure and client failure during write operation and to trigger the recovery action for these system faults. The RM uses the basic mechanism of the heartbeat messages to check the status of I/O server nodes. On the basis of analysis of messages received by I/O servers containing the current state of operations in progress, the RM is informed of any client failure that has happened in the middle of a write operation. This particular event can be detected by means of elapsed timeouts in the two phase write operations. In fact, the client failure represents a treat for the PFS only if it fails either by interrupting a write operation (during the second I/O phase) or by leaving some orphan lock on servers (during the first phase of locks acquisition). In the first case, the server will detect the client failure by

experiencing an interruption in the communication. In the second case there will be some lock with an old timestamp associated with. In both case, the recovery module by polling the I/O server will be aware of these situation and will trigger the recovery operations. The recovery action performed by the RM for the client failure consist of two operation: to fix the redundancy information of the file region involved in the failed write and to release the lock associated with the failed write.

As concern for the I/O server failure, when an I/O server does not respond to the heartbeat message within a timeout, the RM updates an internal structure in which it store the health state of the system. When the failed I/O server will be restored, the RM will be responsible of data reconstruction necessary to align the server with the other ones. The reconstruction exploits information stored on surviving servers in which there is trace of all files modified during the downtime period. This information is stored on each I/O server when at least one I/O server is recognized failed by the client and will help to speed up the reconstruction by fixing only the bare minimum of files. Only when the reconstruction procedure is completed the I/O server is brought back up into production.

3.2.2 File Resiliency Selection

In order to satisfy different application requirements, MPI-IO layer together with PVFS should provide a mechanism to tune the capacity efficiency, reliability, and performance of the file system. The basic idea is to choose for each file the value k in order to set the resiliency associated with it. To this aim, we have modified the label management used in MPI-IO interface through the ROMIO [20] implementation. This modification allows to resolve new file system aliases in order to extend the PVFS2 label. The new label type we introduced is composed by the old PVFS2 label followed by two values separated by the symbol '@'. For instance, the string $DePFS@n@k : /filename$ means that the application wants to use the file system

PVFS2 with *(n,k)* erasure code to store the file called 'filename'. Figure 3.4 shows the path of the label from the application level up to the PVFS2 client library passing through ADIO, an internal abstract I/O device layer of MPI-IO that enables portability of its implementation. The application selects different label to associate specific

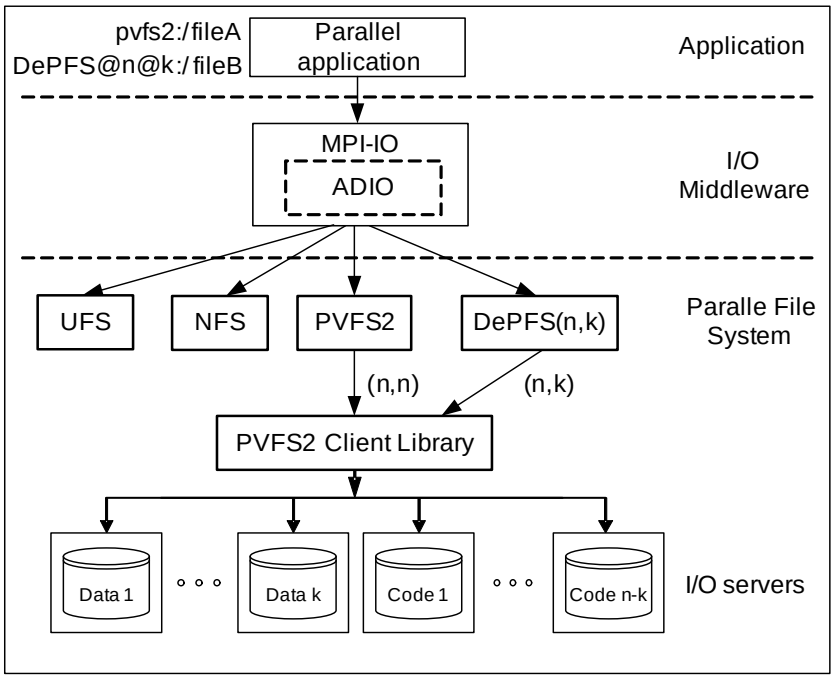

Figure 3.4: DePFS label management

characteristics to the file it creates. The label mechanism is the same used to identify the file system in a cluster environment. Each configuration supported by DePFS, for the application looks like a different file system. When a MPI application issues an operation on a file, ADIO recognizes the target file system and propagates the request. For DePFS labels, ADIO passes additional information to DePFS concerning the two parameters that specify the adopted erasure code. If the operation is a file creation then the client library of DePFS takes care to store the two parameters in special PVFS2 data distribution structure associated with the file on the Metadata server. In subsequent accesses from other clients this structure will be used to operate

correctly on the file. To this aim, we extended the data distribution structure to hold also the coding parameters. This structure is responsible for the mapping between logical and physical file layout. We also created a new type of data distribution to support our new data file distribution.

3.3 Failure Scenarios

According to the fault model presented in the Section 3.1.1, DePFS is able to tolerate failures of both the server and the client node. As stated above, client failures represent a risk for the PFS only during a write operation whereas server failures should be managed carefully during all the system life. The two following sections discuss separately the handling of these two system faults.

3.3.1 Client failure

Client failure during a write operation, as discussed in the Section 3.2.1, can occur in one of the two phases of write: pre-read/lock-acquisition, write/release-lock (see Figure 3.3). A client failure during the acquisition of the locks can leave orphan locks on some of servers because the second phase does not take place. This fault is detected by means of the timeout associated with each lock. The RM periodically checks the status of locks on each I/O server and trigger the correspondent recovery. Instead, a client failure during the second phase can leave the system in a non consistent state because some of the writes has been committed on the disk and some other have not been completely written. In this scenario, two are the possible cases. First, the client fails during the data transmission and thus the server detects the abnormal interruption. Second, the client fails between a completed write on a server and the beginning of the write on the next server. In this second case, the client failure is

detected by means of lock timeouts.

3.3.2 Server failure

I/O server crash is easily detectable through timeout mechanism. The detection of this fault is performed by both clients and RM. Since clients interact directly with I/O servers for each file access, when a request timeout happens the client retry to issue the request for just one time. If also the second request is not accepted within a timeout, the client updates an internal vector status in which there is the status for each server. On the basis of the status vector, the client will issue the future access requests by avoiding to waste time waiting for replies from crashed servers. The vector status has a associated timestamp that elapses in order to enable restored server to be back into production.

As regards the RM, the I/O server crash is detected through the timeout associated to the heartbeat mechanism. Once the fault has been detected, the RM starts collecting on the surviving servers the information about all the write executed after the server crash. These information regard the file written since the crash failure. They will be useful for the data reconstruction when the server will be replaced/restored.

3.4 Recovery Procedure

The RM is the responsible for all the recovery actions executed on the DePFS. These actions are performed by the RM to preserve the file system consistency in spite of system faults. Two are the main type of recoveries: recovery from client failure; recovery from server failure. The first one is triggered immediately after the client failure detection. It consist of fixing the consistency among servers for the stripes involved in the failed write operation. The detection time t_D depends on the heartbeat

period t_{HB} and on the timeout used by the client to assume a server's failure t_T. The relationship among these times are expressed in the following equation:

$$t_D \leq t_{HB} + 2 * t_T \tag{3.4.1}$$

Instead, the time to recover is not deterministic and depends on the number of stripes involved in the failed write. Although there is no upper bound for the time to recover, the orphan locks will protect that region from subsequent write accesses until the recovery has been accomplished and the locks have been released. To this aim, it is worth emphasizing that the RM is designed to be a special client because it is the only one to be able to access file system regardless from locks.

As regards the recovery from server failure, the recovery is triggered automatically when the administrator restarts the I/O daemon on the restored I/O server. This daemon should be restarted with a special flag that allows RM to reconstruct missing data and to align it with other I/O servers while all the clients still consider it as failed. This phase is necessary to permit the server alignment before its definitive integration.

Chapter 4

System Evaluation

The previous chapter showed the DePVFS's architecture and all mechanisms introduced into the original PVFS2 project in order to provide flexible and cost-effective levels of performance and reliability. This chapter focuses on the evaluation of the presented system prototype with the aim to clarify the trade-offs between performance and reliability that it supports. This evaluation is carried out by means of performance analysis of the DePFS prototype.

4.1 Preliminary considerations

Historically, in order to evaluate system's quality (effectiveness) as opposed to its cost, performance measures such as throughput, response time, and resource utilization, have long been recognized important in the context of computer system design. The need to evaluate effects of incorrect behavior due to either design faults or operational faults (physical or human-made faults that subsequently occur during system operation) in this context has received more and more attention during the last two

decades. Both types of faults can obviously affect the system's ability to deliver the correct service. The transition from correct service (usually called also proper service) to incorrect service is called service failure or simply failure. The evaluation of such system's ability can be performed by using certain measures based on the generic concept of *dependability*. The dependability was originally defined as that property of a system which allows reliance to be justifiably placed on the the service it delivers [61]. Special attributes of dependability include reliability (continuity of failure-free service), availability (readiness for correct service), integrity (absence of improper system alterations), and maintainability (ability to undergo modifications and repairs) [62]. While dependability models have the purpose to analyze the reliability and/or availability of a system based on its structure and the failure and repair behavior of its components. The performance models enable to represent the probabilistic nature of the work the system is subject to, and predict its ability to carry out work assuming that the system (or its components) does not fail.

For *degradable systems*, which have the ability to survive the failure of one or more of their active components by degrading the service level, it is necessary to adopt a new modeling paradigm that can give combined performance and dependability measures, usually called *performability* measures [63]. Specifically, with respect to some designated aspect of system quality, a performability measure quantifies how well the system performs in the presence of faults over a specified period of time. Such measures can thus account for multiple degraded levels of performance which, according to failure criteria, remain satisfactory. Indeed, when system performance is modeled separately from dependability, the assumption is being made that system has only two performance levels: fully functional or failed. This assumption holds for systems that has no redundancy, or in which duplicated hardware or software is activated only in case of failure, so that the active system resources remain constant.

4.1.1 Resources Limiting the Performance

Before evaluating the performance of the DEPFS prototype, in this section we want to underline some aspect that should be taken into account in order to properly analyze the results. The performance is influenced by each of the system component: the I/O subsystem, network and CPU. These resources limit strongly the performance of a parallel file system. For instance, the operating system buffers a fair amount of an I/O operation. The buffer size depends on the server memory. A write operation can be buffered efficiently so it can complete before data is actually written to disk. A read operation can completely omit an I/O operation if the data is in memory. Otherwise, the operation has to wait. Another dependency there is with the network. Latency and bandwidth depend on the used network technology.

In comparison with network latency and storage subsystem's access time, the CPU is the fastest component. Even though CPU is not expected to limit operation throughput in the original PVFS2 implementation, with our prototype an expensive computation is performed on the client side at least for each write operation. This is because the adopted redundancy scheme is based on the erasure codes that exhibit a computational cost which grows rather quickly with k and n. However, the effect of CPU on the overall throughput will be noticeable only when the network and the disks throughput are not already saturated.

4.1.2 Scalability

Scalability in terms of a parallel file system's capabilities means:

- the aggregate performance grows proportional to the number of clients
- with a fixed number of clients, the aggregate performance increases proportional to the number of servers

Requests from additional clients can saturate the network or I/O system, on the other hand a server's CPU can be kept busy. In the next sections some of these scalability characteristics are shown.

4.2 Performance Evaluation

There are many benchmarks for file systems, which were approved by the community, for example bonnie++. However, for parallel file systems no such common benchmark suite exists. There are several programs which attempt to fill the gap. Two examples are the Effective I/O Bandwidth Benchmark (b_eff_io) [64] and the NASA Parallel Benchmarks [65]. These approaches are not suited to measure the whole range of interesting performance characteristics. Instead, the benchmarks make assumptions of access patterns used from the clients. Thus, a good result does not necessarily mean that a file system is a good choice for a given application. Often, a scientific application is used directly to measure the I/O system's efficiency. However, the overall performance of an application depends much from the cluster configuration. Therefore, we decided to use simple benchmarks suited for PVFS2 in order to measure points of special interests. These are both I/O requests using a small data-amount (block-size) per access and large sequential I/O requests.

4.2.1 Parallel I/O Benchmark: mpi_io_test

PVFS2 includes in its distribution a program namely *mpi_io_test* that allows to test MPI-I/O interface. The *mpi_io_test* benchmark represents a highly favorable workload for most parallel file systems. The program can be used for all ADIO modules, not only PVFS2. It measures 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 more that in the original request due to the redundancy. 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 data irrespectively of whether or not the data has been encoded. There are two important input parameters: the block size defines the amount of data which is accessed per system interface function call and the iteration count which is the number of subsequent I/O operations. Each client writes a fixed amount of data. The more clients running the benchmark, the larger the generated file. As a result, this benchmark frequently uncovers saturation points in either the storage system or networking infrastructure. For each data point in the experiments, *mpi-io-test* has been run with a iteration count equal to 10 in order to reduce the uncertainty of measurements.

4.2.2 Experiment Setup

All experiments are performed on the a large production cluster at the Ohio Supercomputer Center. It consists of 128 nodes with dual 900MHz Itanium II processors, 4GB of RAM and a single 80GB SCSI disk, and both a Myrinet 2000 interface card and a Gigabit Ethernet interface. In our experiments, the traffic between the client nodes and the I/O servers used Myrinet.

In all of our test configurations we assigned each node to be either a computation node or an I/O node and only one Metadata manager was running on a separate node. On a computer that acts as a client, a single client process of *mpi_io_test* benchmark

is run. As concern the DePFS configuration, for all the experiments we used a stripe unit size equal to 16KByte, instead the stripe size depends on the number of I/O nodes used in the specific test ($stripe\_size = stripe\_unit\_size * num\_IO\_nodes$).

4.3 Sensitivity to the I/O Request Size

In this section, the performance of DePFS is analyzed for the following two cases: small contiguous I/O requests, large contiguous I/O requests. We consider a request as a small request when it requires data that fits in a stripe. In these tests the number of available I/O servers has been fixed to 8 while the number of nodes used to hold redundancy has been varied between zero and four (i.e., configurations with codes: *8-of-8*, *7-of-8*, *6-of-8*, *5-of-8*, *4-of-8*).

It is worth noting that the configuration *8-of-8* corresponds to the original PVFS2 with 8 I/O servers. In this way, for each choice of the resiliency level the performance behaviors is drawn by varying the size of access and by keeping fixed the available resources. In these tests, the performance measurements represent the read and write throughput in Mbyte/s as seen by just one client in a fault-free scenario. The results for each file configuration has been obtained by simply repeating the benchmark with different label in the file name.

4.3.1 Small Contiguous I/O Requests

In this test, one client repeatedly accesses a file with different block sizes between 16 Kbyte and 128 Kbyte. It is expected that the performance improves when the block size increases. This behavior is common to both read and write. The read performance for different file configuration is almost the same, as shown in Figure 4.2, because the DePFS read in a fault-free scenario does not need to decode data. Furthermore, also

the different number of I/O nodes used for explicit data within different configurations does not influence the results because the access does not exploit all the available parallelism of the I/O nodes.

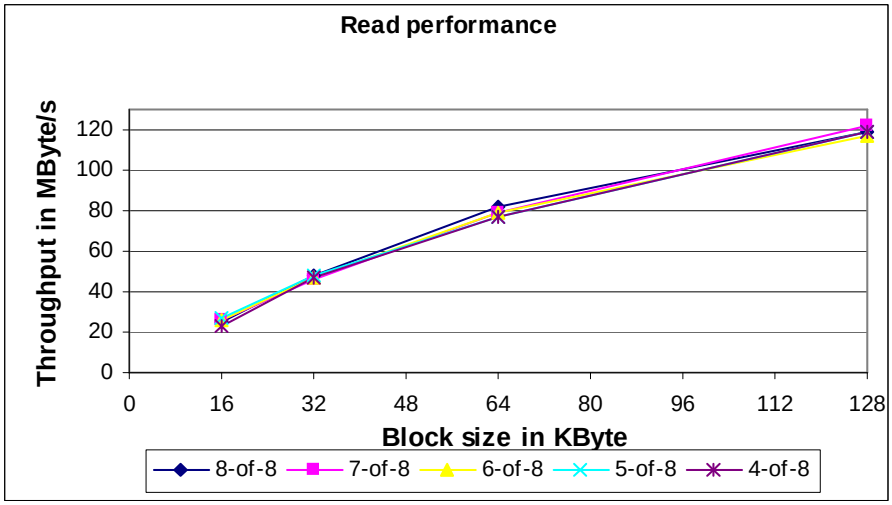

Figure 4.1: Read throughput using small block sizes

The same consideration does not apply to the DePFS write. In fact, also in a fault-free scenario, the write needs to encode data in order to update the redundancy. This glitch is evident in Figure 4.2, where the computational overhead associated with the erasure codes makes the difference in the performance for the considered configurations. Although the performance loss between PFVS2 and DePFS with 7-of-8 is prominent, the performance loss among the remaining configurations is not appreciable. These different performances are directly dictated by the characteristics of the employed erasure code. In fact, the decode time depends on the optimizations exploited in the implementation of the adopted RS decode algorithm.

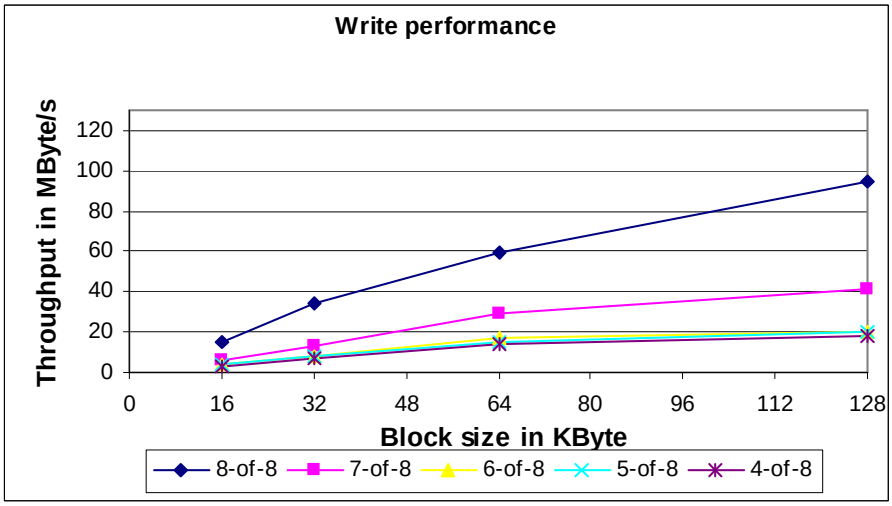

Figure 4.2: Write throughput using small block sizes

4.3.2 Large Contiguous I/O Requests

Similarly to the test above in this test, one client repeatedly accesses a file but with block sizes that vary between 1 Mbyte and 16 Mbyte. In this case all the accesses request data blocks that do not fit in one stripe since the stripe size is 128 KByte (stripe unit size = 16KByte and number of I/O nodes = 8) whereas the minimum access size for this test is 1MByte. The read performance for large requests, depicted in Figure 4.3, compared with the one with small requests remarks a slight difference among the configurations. This variance is due to the different degree of parallelism provided by the configurations. Even though the number of I/O nodes is fixed, the parallelism in the data access depends only on the number of I/O node with explicit data which changes for each configuration. Looking at the Figure 4.3, it is evident that the client has reached the maximum peak performance, in that a request size larger than 16MBytes does not improve the performance. This limit is dictated by the saturation of the client bandwidth, in fact as we will see later, the PFS is still able to increase its delivered bandwidth. This last observation is valid also for the write

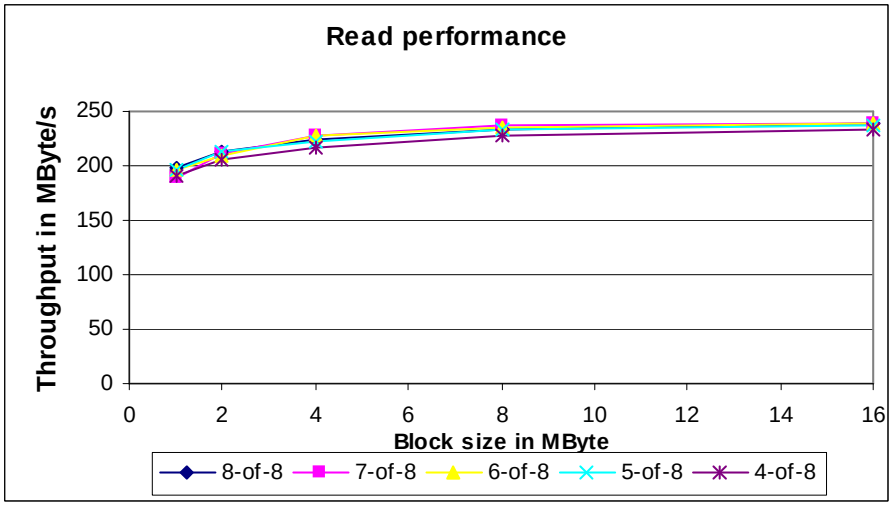

Figure 4.3: Read throughput using large block sizes

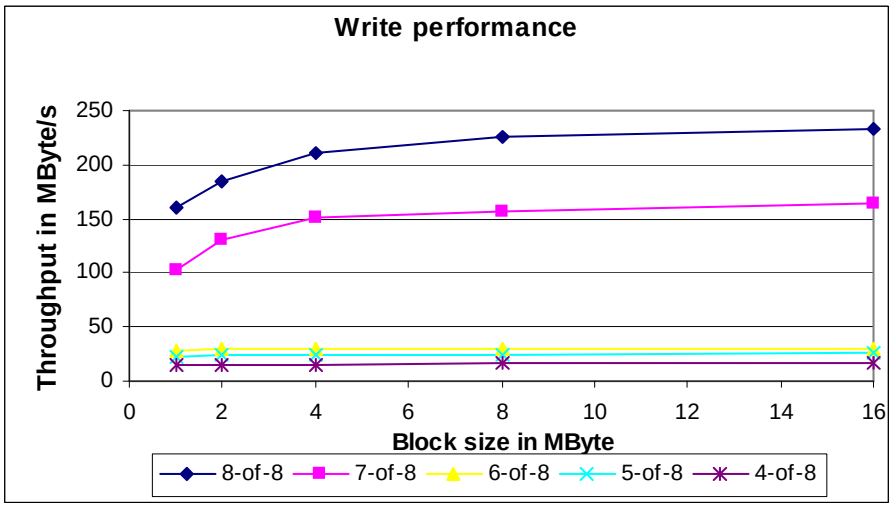

Figure 4.4: Write throughput using large block sizes

performance depicted in Figure 4.4. This figure is consistent with the one concerning the small access size because the same differences among the file configurations are emerged.

4.4 Scalability Evaluation

The scalability of DePFS has been studied by varying both the number of clients and the number of servers. When the number of clients increases we expect to measure a growth in the aggregate throughput as seen by the parallel application. This growth should be visible up to the complete saturation of either the network or I/O subsystems. Instead, when the number of clients is fixed and the number of I/O nodes increases it is possible to measure as a greater parallelism can improve the aggregate throughput.

4.4.1 Increasing the Number of Clients

For this test we used the same DePFS setup adopted previously. It consists of 8 I/O servers and one Metadata server. The number of clients has been varied by running the *mpi_io_test* benchmark with different values of the parameters. In order to avoid contention in the use of resources among the *mpi_io_test* processes, we run each process on a separate node. In the Figure 4.5 is depicted the DePFS read performance by varying the number of clients from 1 to 16. The measurements confirm the expected behavior of DePFS read. For a file with configuration *k-of-n*, DePFS read delivers the same throughput as PVFS2 read with k I/O servers. The DePFS write performance for number of total I/O servers n equal to 8 and resiliency levels n-k equal to 1,2,3,and 4 are depicted in Figure 4.6. While DePFS with configuration *7-of-8* gets comparable performance with PVFS2, the other three configurations exhibit

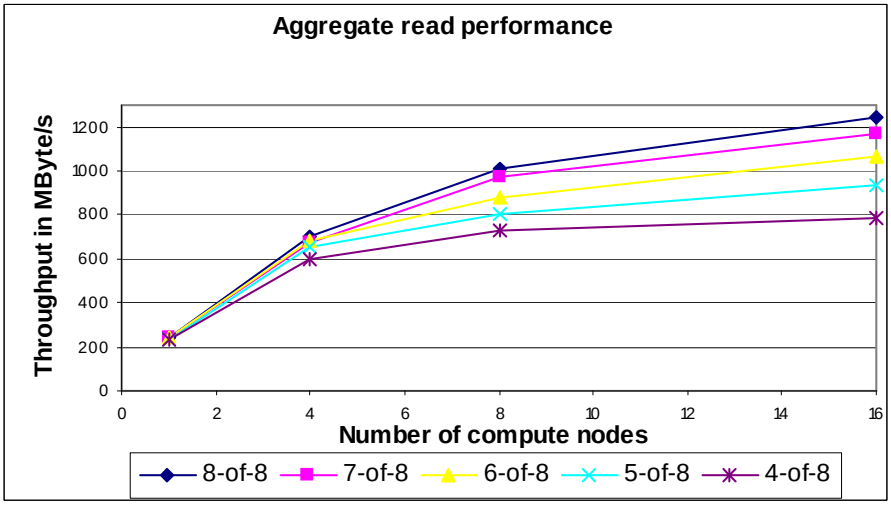

Figure 4.5: Aggregate read throughput by increasing number of clients

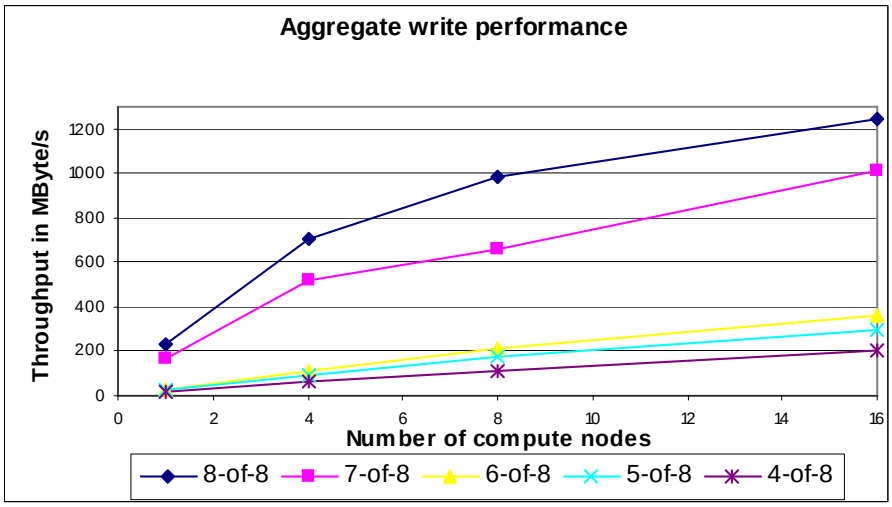

Figure 4.6: Aggregate write throughput by increasing number of clients

poor performances even though they show scalable performance when the number of clients increases.

4.4.2 Increasing the Number of Server

To complete the scalability evaluation of DePFS, we have performed a test with an increasing total number of servers. In order to evaluate the DePFS performance with different number of available servers we decide to keep fixed the degree of file resiliency n-k and changing only the number n of total I/O servers. In this test we have employed 8 clients and a number of servers that ranges from 4 to 32. The

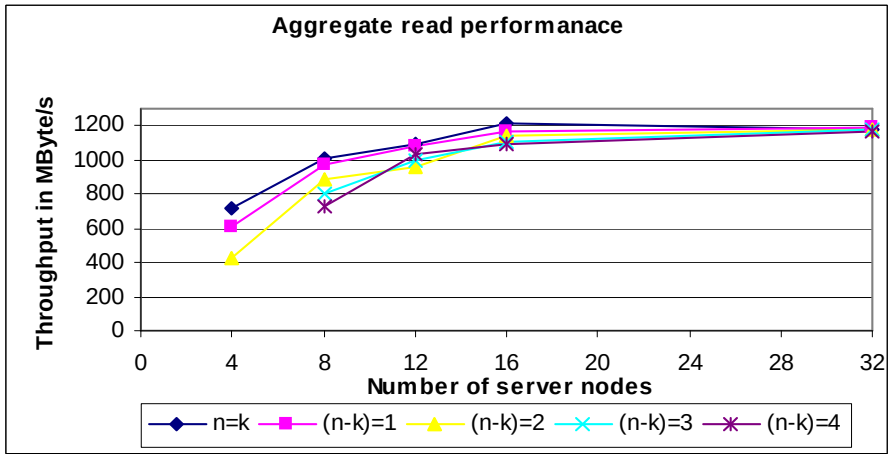

Figure 4.7: Aggregate read throughput by increasing number of total servers

read measurements depicted in Figure 4.7 show a saturation of clients bandwidth for all the 5 configurations. In fact, even if the number of I/O nodes grows from 16 to 32 the aggregate throughput remains steady around 1200 MByte/s. As regards the write performance, also in this test, the DePFS with file resiliency equal to one, with the only exception in correspondence of 12 I/O servers, is the only configuration that exhibits performance very close to the one of PVFS2 that instead exploits all the I/O node to store explicit data. As already seen with the read case, the number of clients

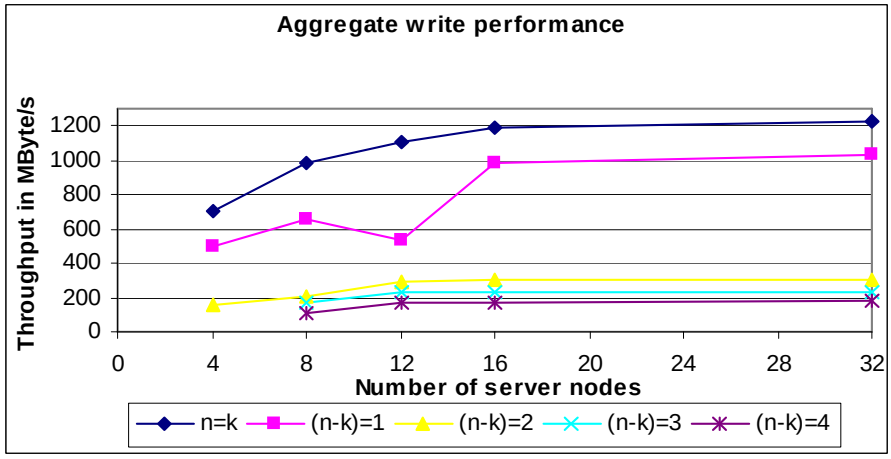

Figure 4.8: Aggregate write throughput by increasing number of total servers

is not sufficient to exploit the increasing potential throughput provided with these DePFS configurations.

4.5 Performance in presence of server failures

So far we have evaluated the DePFS read and write performance for different configurations. In this section we draw the performance that DePFS deliver in presence of server failures. DePFS is expressly designed to tolerate a certain number of server failures by exploiting the redundant information held on surviving servers. Thus, when a server failure occurs, the client continues to use properly the file system but with reduced performance due to continuous data reconstruction that is necessary to obtain data from surviving I/O servers. Actually, only when the failure occurs to I/O nodes that hold explicit data the performance become worse. If the failed server hold encoded data, the read will continue to access only servers with explicit data while the write will continue to write only on the surviving servers. For this reason, the test we presented in this section refers only to server with explicit data. The measurements

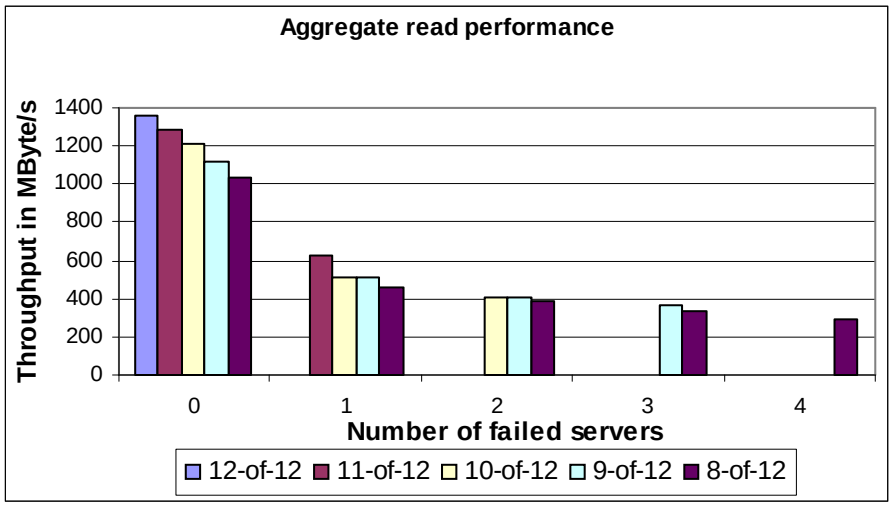

Figure 4.9: Aggregate read throughput in presence of server failures

shown in Figure 4.9 refer to a DePFS with 12 I/O servers and 5 file configurations with increasing level of associated resiliency. The number of clients used in this test is equal to 16 with a total of 29 nodes. The number of server failures equal to zero corresponds to the fault-free scenarios. The cost associated to the encode process becomes remarkable when the number of failure increases. It is worth noting that although the configurations with great resiliency do not exhibit great performance they can assure medium performance for longer period of time because they can survive to more server failures. The DePFS write performance, in presence of server failures, is completely different from the read case. In fact, while the read needs to decode data from surviving servers only when the explicit data are not available, the write has always the need of encode data independently by server failure occurrences. This consideration is confirmed by the Figure 4.9, in which the performance of write for all the configurations is constant irrespectively of the number of server failures.

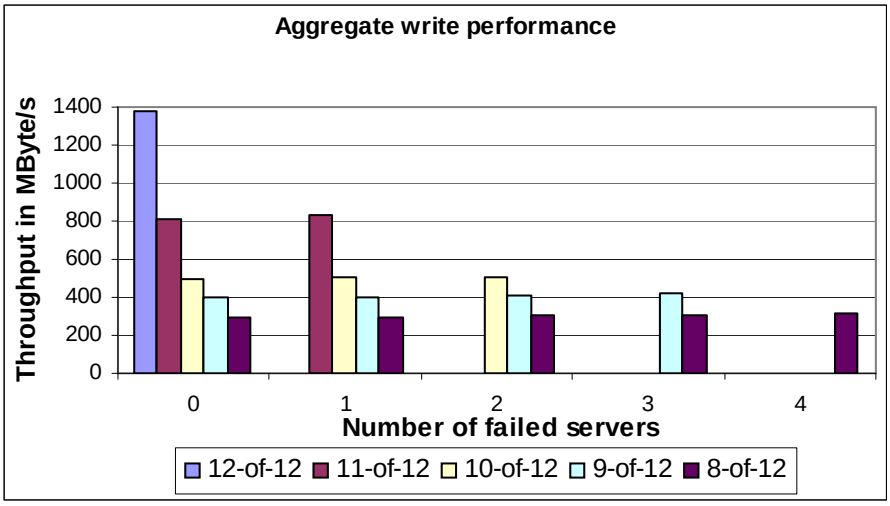

Figure 4.10: Aggregate write throughput in presence of server failures

Chapter 5

Final Remarks and Conclusions

The aim of this dissertation was to underline the need for versatile parallel file systems that can efficiently support parallel applications and meet their diverse requirements. PFS are specifically designed to work on shared environments, such as clusters, in which many parallel applications demand high performance and high availability. These two requirements cannot be satisfied independently but a trade off should be selected by each application for each file it creates.

In this dissertation, a redundancy strategy has been described in detail. It allows to flexibly achieve either high level of file availability or high performance access by using accordingly the available resources. Most of the mechanisms necessary to cope with common system faults such as client and server failure has been discussed. A prototype parallel file system, namely DePFS, has been built to verify the goodness of the proposed strategy. Throughput experiments with DePFS demonstrate its characteristics with respect to request size sensitivity, scalability, and performance in presence of server failures. The versatility of DePFS has been shown by comparing for each experiment at least 5 DePFS file configuration with the unique PVFS2

configuration on equal available resources. Furthermore, the performance evaluation have clearly shown with meaningful configurations the trade off that this strategy can offer to the parallel applications and even though more resilient file configurations do not achieve very high performance they can be very useful for specific file in which the availability is more critical than performance. For instance, all those files that are not accessed frequently but that hold final results.

5.1 Future work

The system evaluation section has underlined the need to go into more depth in erasure code studies. The computational cost associated with the encode and decode process becomes prominent for some system configuration, for instance when the degree of resiliency is greater than two. We are investigating the adoption of more sophisticated erasure codes that reduce the computational cost.

Another aspect of this dissertation that deserve more attention is the effect that the adopted locking mechanism introduce with respect to different workload and more sophisticated benchmarks. Even though *mpi_io_test* is very useful to understand the peak performance that a parallel file system and to uncover the limiting resources for each system configurations, it is not very representative of real application workload.

Bibliography

[1] Walter B. Ligon III. Research directions in parallel I/O for clusters. In *CLUSTER*, page 436. IEEE Computer Society, 2002.

[2] David Kotz and Nils Nieuwejaar. Flexibility and performance of parallel file systems. *SIGOPS Oper. Syst. Rev.*, 30(2):63–73, 1996.

[3] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of the Conference on Supercomputing*, pages 640–649, Los Alamitos, November 1994. IEEE Computer Society Press.

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

[5] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/February 2002.

[6] David Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). 17(3):109–116, June 1988.

[7] Ralph Sandberg et al. Design and implementation of the SUN network file system. In *Proceedings of the Summer USENIX Conference*, pages 119–130, Portland, Oregon, June 1985. Usenix Association.

[8] Steven A. Moyer and V. S. Sunderam. Characterizing concurrency control performance for the PIOUS parallel file system. *Journal of Parallel and Distributed Computing*, 38(1):81–91, October 1996.

[9] James V. Huber, Jr., Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 22, pages 330–343. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[10] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[11] Nils Nieuwejaar and David Kotz. The galley parallel file system. *Parallel Computing*, 23(4-5):447–476, 1997.

[12] Philip H. Cams, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. 2(1):317–327, October 2000.

[13] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the FAST '02 Conference on File and Storage Technologies (FAST-02)*, pages 231–244, Berkeley, CA, January 28–30 2002. USENIX Association.

[14] Florin Isaila and Walter F. Tichy. Clusterfile: a flexible physical layout parallel file system. *Concurrency and Computation: Practice and Experience*, 15(7-8):653–679, 2003.

[15] The MPI Forum. Mpi-2: Extensions to the message-passing interface, Jul 1997.

[16] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32, New York, NY, USA, 1999. ACM Press.

[17] Inc. Cluster File Systems. The lustre storage architecture. Technical report, 2003.

[18] Robert Ross, Rajeev Thakurand, and Alok Choudhary. Achievements and challenges for i/o in computational science. *Journal of Physics: Conference Series (SciDAC 2005)*, 16:501–509, 2005.

[19] International Organization for Standardization. Information technology — portable operating system interface (POSIX) — part 1: System application program interface (API) [C language]. ISO/IEC 9945-1, 1996.

[20] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, feb 1999.

[21] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[22] Michael Stonebraker and Gerhard A. Schloss. Distributed RAID - A new multiple copy algorithm. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. IEEE/Wiley Press, New York, 2001. chap. 6.

[23] CHARNG-DA LU. Scalable diskless checkpointing for large parallel systems, 2005.

[24] Yanyong Zhang, Mark S. Squillante, Anand Sivasubramaniam, and Ramendra K. Sahoo. Performance implications of failures in large-scale cluster scheduling. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *JSSPP*, volume 3277 of *Lecture Notes in Computer Science*, pages 233–252. Springer, 2004.

[25] Ramendra K. Sahoo, Anand Sivasubramaniam, Mark S. Squillante, and Yanyong Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *DSN*, page 772. IEEE Computer Society, 2004.

[26] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure data analysis of a LAN of windows NT based computers. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS 99)*, pages 178–189, Washington - Brussels - Tokyo, October 1999. IEEE.

[27] LANL. The raw data and more information is available at the following two urls:. http://www.pdl.cmu.edu/failuredata/ and http://www.lanl.gov/projects/computerscience/data/, 2006.

[28] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006)*, pages 249–258. IEEE Computer Society, 2006.

[29] R. K. Iyer and D. J. Rosetti. Effect of system workload on operating system reliability: A study on IBM 3081. *IEEE Transactions on Software Engineering*, 11(12):1438–1448, 1985.

[30] J. Morrison. The asci q system at los alamos (talk), 2003.

[31] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *In Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[32] David Nagle, Denis Serenyi, and Abbie Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53, Washington, DC, USA, 2004. IEEE Computer Society.

[33] E. K. Lee and C. A. Thekkath. Petal: distibuted virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA USA, 1996.

[34] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe Matthews, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 24, pages 364–385. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[35] Y. Zhu et al. Improved read performance in a cost efficient, fault-tolerant parallel virtual file system (CEFT-PVFS). In *Parallel I/O in Cluster Computing and Computational Grids: IEEE/ACM International Symposium on Cluster Computing and the Grid 2003 (IEEE/ACM CCGRID 2003)*, Tokyo, Japan, May 2003.

[36] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[37] Manoj Pillai and Mario Lauria. CSAR: Cluster storage with adaptive redundancy. In *ICPP*, pages 223–230. IEEE Computer Society, 2003.

[38] Domenico Cotroneo, Generoso Paolillo, Stefano Russo, and Mario Lauria. CSAR-2: A case study of parallel file system dependability analysis. In *High Performance Computing and Communications, First International Conference, HPCC*, volume 3726 of *Lecture Notes in Computer Science*, pages 180–189. Springer, September 2005.

[39] Zheng Zhang, Shi-Ding Lin, Qiao Lian, and Chao Jin. Repstore: A self-managing and self-tuning storage backend with smart bricks. In *ICAC*, pages 122–129. IEEE Computer Society, 2004.

[40] R. Golding, E. Shriver, T. Sullivan, and J. Wilkes. Attribute-managed storage. In *Wkshp. on modeling and specification of I/O*, 1995.

[41] Bradley W. Settlemyer. A mechanism for scalable redundancy in parallel file systems, May 2006.

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

[43] Khalil Amiri, Garth A. Gibson, and Richard A. Golding. Highly concurrent shared storage. In *Proceedings of the Int. Conf. on Distributed Computing Systems (ICDCS2000)*, pages 298–307, April 2000.

[44] William D. Gropp, Robert B. Ross, and Neill Miller. Providing efficient I/O redundancy in MPI environments. In Dieter Kranzlmúller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*, volume 3241 of *Lecture Notes in Computer Science*, pages 77–86. Springer, 2004.

[45] L. Lamport. On interprocess communication. *Distributed Computing*, 1:77–101, 1986.

[46] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 23(5):202–210, December 1989.

[47] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2), June 1981.

[48] J. N. Gray. Notes on data base operating systems. In Bayer, Graham, and Seegmuller, editors, *Operating Systems, an Advanced Course*, volume 60. Springer Verlag, Heidelberg, FRG and NewYork NY, USA, lncs edition, 1978.

[49] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proc. Thirteenth ACM Symp. on Operating System Principles*, page 198, Pacific Grove, CA, October 1991. Published as Proc. Thirteenth ACM Symp. on Operating System Principles, volume 25, number 5.

[50] Brian Noble and Mahadev Satyanarayanan. An empirical study of a highly available file system. In *SIGMETRICS*, pages 138–149, 1994.

[51] Hong Tang, Aziz Gulbeden, Jingyu Zhou, William Strathearn, Tao Yang, and Lingkun Chu. A self-organizing storage cluster for parallel data-intensive applications. In *In Proceedings of the 2004 ACM/IEEE conference on Supercomputing,SC 04*, page 52, Washington, DC, USA, 2004. IEEE Computer Society.

[52] Changsheng Xie and Bin Cai. A decentralized storage cluster with high reliability and flexibily. In *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP06)*, 2006.

[53] Aguilera M. K., Janakiraman R., and Xu L. Using erasure codes effciently for storage in a distributed system. In *Proceedings. International Conference on Dependable Systems and Networks, DSN 2005*, pages 336 – 345, June 2005.

[54] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the oceanstore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies, FAST '03*, pages 1–14, March 2003.

[55] Schlichting R. D. and Schneider F. B. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 3(1):222–238, 1983.

[56] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, , and Sam Toueg. The primary-backup approach. pages 199–216. ACM Press - Addison Wesley, December 1993.

[57] Weatherspoon and Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems (IPTPS), LNCS*, volume 1, 2002.

[58] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on reed-solomon coding. *Software Practice & Experience*, 2(35):189–194, February 2005.

[59] Peterson W. W. Error correcting codes, 1961.

[60] J. S. Plank. Gflib - c procedures for galois field arithmetic and reed-solomon coding, 2003.

[61] J.-C. Laprie. *Dependability: Basic Concepts and Terminology*, volume 5 of Dependable Computing and Fault-Tolerant Systems. Springer-Verlag, 1992.

[62] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Ccomputing*, 1(1):11–33, March 2004.

[63] Robin A. Sahner, Kishor S. Trivedi, and Antonio Puliafito. *Performance and reliability analysis of computer systems*. Kluwer Academic Publishers, 2002.

[64] Rolf Rabenseifner, Alice E. Koniges, Jean-Pierre Prost, and Richard Hedges. The parallel effective I/O bandwidth benchmark: b_eff_io. In Christophe Cerin and Hai Jin, editors, *Parallel I/O for Cluster Computing*, chapter 4, pages 107–132. Kogan Page Ltd., February 2004.

[65] D. Bailey and et. al. The NAS parallel benchmarks. Technical Report Report RNR-91-002 revision 2, NASA Ames Research Center, 1991.