ABSTRACT

Network Transparency
in Wide Area Collaborations

by
Jiaying Zhang

Chair: Peter Honeyman

The advent of wide-area high-speed networks provides the framework for deploying large scale applications. Concurrently, recent years have seen an increasing demand for global collaborations in scientific studies, spanning disciplines from high-energy physics, to climatology, to genomics. Applications in these fields demand intensive use of computational resources far beyond the scope of a single organization, and require access to massive amounts of data. This introduces the need for scalable, efficient, and reliable data access and management schemes.

To meet availability, performance, and scalability requirements, distributed services naturally turn to replication and file service is no exception. While the concept of file replication is not new, existing solutions either forsake read/write replication totally or weaken consistency guarantees. They fail to satisfy the requirements of global scientific collaborations, in which users want to use widely distributed computation and storage resources as though they were using them locally.

The rapid evolution of wide-area collaborations calls for a mutable replicated file system that supports consistent data access. However, when designing such a system, we must also consider the tradeoffs among consistency, performance, and availability. Most scien-

tific applications are read dominant, so a file system cannot become widely deployed if it supports mutable replication at the cost of impairing normal read performance. Similarly, support for strong consistency guarantees should not slow down applications for which weaker consistency semantics suffice, otherwise, those applications will not choose to employ the system.

This dissertation describes a replicated file system designed to meet the needs of global collaborations. We build a naming scheme that supports a global name space and location independent naming, which facilitates data sharing, distribution, and management. We develop a replication protocol that supports mutable replication with strong consistency guarantees. We implement our design in NFSv4, the new Internet distributed file system protocol, and evaluate the performance of the system with scientific applications. These studies support my thesis that a replicated file system framework addressing the data access and storage requirements of the emerging global collaborations is feasible and practical.

Network Transparency in Wide Area Collaborations

by

Jiaying Zhang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2007

Doctoral Committee:

Adjunct Professor Peter Honeyman, Chair
Associate Professor Brian D. Noble
Assistant Professor Zhuoqing Mao
Research Professor Thomas A. Finholt

to my family

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Peter Honeyman, who has guided my research over the past six years. I knew little of how to do research when I came to the graduate school. Peter was really patient with my growth and gave me enormous help and support. He spent a lot of time improving my writing. He gave me many freedoms throughout my graduate study. His enthusiasm, trust, and high expectations helped me through the hard time of my research.

I thank Professors Brian Noble, Thomas Finholt, and Morley Mao for serving on my committee. Professor Noble gave me many perceptive suggestions during my proposal and my defense. Professor Finholt made valuable comments on the prospects of Grid computing and the deployment of my work. Professor Mao has given me a lot of support and encouragement during my study.

I thank Dr. Shawn McKee and Tiesheng Dai for their help in setting up Atlas applications. They spent time from their tight schedules and led me through the work with patience. I also learned a lot from discussions with them on the data access needs in scientific computing. I also thank Dr. Daniel Nurmi for his advice on analyzing the failure distribution data.

I would like to thank several people at Center for Information Technology Integration (CITI). Trond Myklebust, Bruce Fields, and Andy Adamson gave me much help on NFS version 4 and the implementation work of my thesis. Olga Kornievskaia and Dean Hildebrand shared with me many experiences and thoughts throughout my graduate life. Chuck Lever, Charles Antonelli, Kevin Coffman, and Jim Rees kindly gave me help whenever I had any hardware or software problems.

I thank all of the friends I met at the University of Michigan, especially Wenjie Wang, Zhiheng Wang, Xiang Sun, and Huan Guo who helped me much during my first years; Jian Wu, Xin Zhao, Hai Huang, and Guangyu Cao for their collaborations on our class projects;

and Ping Yu and Congxian Jia, my roommates and friends, for spending time with me outside work.

Finally, I thank my family for their unconditional love and support. I am deeply indebted to my parents, Shuchi Wang and Yongcai Zhang. They taught me the importance of education since I was a child. They gave me a lot of support when I decided to come to the U.S.. They have suffered so much for my sister and me to pursue our careers. I am also very lucky to have a sister, Xianan Zhang, with whom I can talk about so many things, from personal life to research. As an elder sister, she set a standard for me to learn from.

This work has been partially supported by the Department of Energy under Award Numbers DE-FC02-06ER25766 and B548853, Lawrence Livermore National Laboratory under contract B523296, National Science Foundation award SCI-0438298, and by grants from Network Appliance and IBM.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1

Introduction

The advent of wide-area high-speed networks provides the framework for deploying large scale applications. Concurrently, recent years have seen an increasing demand for global collaborations in scientific studies. This dissertation describes a replicated file system designed to meet the needs of the emerging wide-area collaborations.

By way of introduction, this chapter reviews current network developments, the requirements of large-scale Internet applications, the motivation of my research, and the outline of this dissertation.

1.1 Distributed File Systems and Network Transparency

Advances in network technologies impelled the generation and development of Distributed File Systems. The history of Distributed File Systems dates back to the 1980s, a decade marked by two related advances in computer technology: the development of powerful microprocessors and the invention of high-speed networks. From 1945, when the earliest computers were created, until the late 1970s, computers were large and expensive. Even minicomputers cost tens of thousands of dollars. Most organizations could afford at most a handful of computers, which operated independently from each other for lack of a way to connect them. That situation started changing in the early 1980s. With the development of microprocessor technology, the cost of computers decreased dramatically. As high-speed networks became available, data was moved around easily in a fast local area network (LAN). These changes made it feasible for multiple interconnected computers to

cooperate in a local area network, introducing the need to share data easily, reliably, and efficiently. This spurred the research on Distributed File Systems.

Before Distributed File Systems were developed, remote file transfer, such as FTP [27], was the prevalent mechanism for sharing data in distributed computing. In this paradigm, when an application needs to access remote data, the entire file is transferred across the network from a remote computer. Although conceptually simple and easy to implement, this approach has several deficiencies. First, it is inconvenient. Applications have to know which files are remote and which are local, and explicitly transfer a remote file before viewing or accessing it, then transfer it back if the file is modified. Furthermore, applications need to manage transferred files themselves: they need to find a disk with enough space to store the transferred file, and when the file is no longer needed, they must remember to remove it to prevent local disks from becoming cluttered. Second, it does not support consistent sharing for distributed applications. When a file is accessed by multiple applications on different machines and one modifies it, the modification is not observable by others. Furthermore, when a file is concurrently modified on different machines, the final result reflects only the modification of the last writer. Third, it is inefficient. A remote file is transferred in its entirety even if an application wants to access only a small piece of the file. Furthermore, users may be forced to wait for long periods before being able to access the first byte of their data.

All of these deficiencies can be summarized as insufficient *network transparency* [104]. Network transparency allows applications to access network resources without needing to know whether they are remote or local. A system that provides higher network transparency lets applications extend to distributed environment easily, as no special software development are needed for this purpose. The recognition of this requirement motivated the development of Distributed File Systems; the extent to which a distributed file system provides network transparency is an important measure of quality.

Network transparency is preferable for any distributed systems. However, the degree of network transparency that a distributed system can provide is limited by network conditions. For example, in a slow network, it is hard to hide access latency when an application is accessing a remote file. As another example, in a network with a high rate of data loss, it is difficult to mask access failures. Furthermore, different applications have different characteristics and requirements. Applications that update data frequently require high net-

work transparency for write operations, but this is less of a concern for applications that are read dominant. Some applications can tolerate inconsistent data access, but others have strict requirements for the freshness of the accessed data. The design of any distributed file system must address the network conditions of its time and balance the requirements of diverse applications. As network technology develops further and new applications appear, distributed file systems face new conditions and requirements. Correspondingly, old designs need to be re-evaluated and, if necessary, replaced by new solutions.

In the 1980s, distributed computing was largely limited to local area networks. Most distributed file systems developed at that time assumed a LAN environment characterized by low latency and low data loss. Early NFS implementations [102], for example, used UDP [96] as the underlying transport protocol, and allowed no more than 8 KB block transfers.

The 1990s saw further advances in network technology. Most significantly, the Internet flourished as backbone speed improved. Millions of computers were suddenly networked together. Users and applications of distributed file systems were now widely disseminated. To cope with these changes, new versions of distributed file systems were developed, with extended features that support wide area data access. For example, version 3 of NFS [91], released in 1994, added support for reliable Internet communication using TCP, and allowed server and client to negotiate block transfer size for better performance over heterogeneous networks.

The advance of network technologies has not stopped. The nascent 21st century has seen further advances in wide area network (WAN). Driven by the widespread adoption of Gigabit Ethernet, WAN bandwidth capacity has increased rapidly. The pioneering high performance network Internet2 joins over two hundred universities and corporate partners through 10 Gbps connections, facilitating the development and deployment of advanced network applications and accelerating the creation of tomorrow's Internet. These advances provide the framework for employing large scale applications.

On the other hand, WAN delay remains an issue. Limited by the speed of electrons traveling on a wire and overhead in packet routing, network delay is not likely to drop dramatically any time soon. Minimizing the impact of remote data access is still key to improving the performance of a distributed system deployed over WANs.

Furthermore, the Internet is unreliable. Internet services depend on the proper functioning of many components, such as wide-area routing, access links, the domain name system, and servers themselves. Failure in any of them can interrupt end-to-end communications. Although various techniques have been developed to cope with link and router failures, there remain a number of more vexing faults, such as software and hardware bugs, improper configuration, or inability of current routing systems to cope with persistent congestion. The situation is exacerbated by the increasing number of attacks launched on the Internet. For example, Feb 2000 saw a massive denial of service attack on major web sites, including Yahoo, Amazon, and eBay [115]. In Jan 2003, the SQL Slammer worm took down 5 of the 13 DNS root servers along with tens of thousands of other servers, and affected a multitude of systems ranging from bank ATM to air traffic control to 911 emergency [80]. With the increasing damage caused by network failures and attacks, reliability and availability become key issues for distributed systems.

1.2 Global Scientific Collaborations

Stimulated by the Internet advances, the scientific community is seeing an increasing number of global collaborations, spanning disciplines from high-energy physics, to climatology, to genomics [5–7]. Applications in these fields demand intensive use of computational resources far beyond the scope of a single organization, and require access to massive amounts of data. This imposes new challenges in data access, processing, and distribution.

Driven by the needs of scientific collaborations, the emerging Grid infrastructure [30, 45] aims to connect globally distributed resources to form a shared virtual computing and storage system, offering a model for solving large-scale computation problems. Sharing in Grid computing is not merely file exchange but also entails direct access to computers, software, data, and other resources, as is required by a range of collaborative scientific problem-solving patterns. To make such sharing simple and effective demands data access schemes that are scalable, reliable, and efficient.

To meet availability, performance, and scalability requirements, distributed services naturally turn to replication, and file service is no exception. While the concept of file system replication is not new, existing solutions either forsake read/write replication totally

[11, 105, 118] or weaken consistency guarantees [60, 95, 106]. These compromises fail to satisfy the requirements of global scientific collaborations.

As an example, consider the Atlas experiment [1], which searches for new discoveries in high-energy proton collisions. Protons will be accelerated in the Large Hadron Collider accelerator, currently under construction at the European Laboratory for Particle Physics (CERN) near Geneva [5]. After the accelerator starts running, Atlas is expected to collect more than a petabyte of data per year, which then needs to be distributed to a collection of decentralized sites for analysis. Atlas is the largest collaborative effort ever attempted in the physical sciences. 1800 physicists from more than 150 universities and laboratories in 34 countries participate in this experiment. The wide distribution of the participating researchers and the massive amount data to be collected, distributed, stored, and processed demand scalable and efficient data access and storage schemes to allow physicists in all world regions to contribute effectively to the analysis and the physical results.

The workloads of Atlas have a mix of production and analysis activities. Physics analysis, for example, is an iterative, collaborative process. The stepwise refinement of analysis algorithms requires the use of multiple clusters to reduce development time. Although the workloads during this process are dominated by read, they also demand the underlying system to support write operations. Furthermore, strong consistency guarantees are often assumed. For example, an executable binary may incorporate user code that is finished only seconds before the submission of the command that requires using the code. To guarantee correctness, the underlying system needs to ensure that the modified data is successfully transferred to the remote machine where the code is running.

1.3 Thesis Statement and Organization

The rapid evolution of wide-area collaborations calls for a mutable (i.e., read/write) replicated file system that supports consistent data access. To facilitate Grid computing over wide area networks, we designed and implemented a replicated file system that provides users high performance data access with conventional file system semantics. The system supports a global name space and location independent naming, so applications on any client can access a file with a common name and without needing to know where the data physically resides. The system supports mutable replication with explicit consistency

guarantees, which lets users make data modification with ease, safety, and transparency. The system provides semantics compatible with POSIX [2], allowing easy deployment of unmodified scientific applications. We implemented this system by extending NFSv4, the emerging standard protocol for distributed filing [92]. These studies support my thesis:

A replicated file system framework that addresses the data access and storage requirements of the emerging global collaborations is feasible and practical.

The remainder of this dissertation is structured as follows. Chapter 2 reviews previous work on distributed file systems, replication and concurrency control in distributed database systems, and recent developments in Grid middleware and scientific collaborations. Chapter 3 discusses the design considerations for consistency, failure resiliency, performance, and name space architecture. Chapter 4 presents the design, implementation, and evaluation of a replication protocol that supports mutable replication with strong consistency guarantees and high failure resilience. Chapter 5 proposes a hierarchical replication control protocol that relaxes update distribution requirements to improve performance for synchronous updates, and amortizes the cost of synchronization by allowing a replication server to assert control at granularities coarser than a single file or directory. Chapter 6 explores the tradeoff between performance and failure resilience when using different update distribution policies in the presented replicated file system. Chapter 7 summarizes my contributions and outlines future work.

CHAPTER 2

Background and Related Work

This chapter provides the background for discussing replicated distributed file system and gives an overview of related work. Section 2.1 reviews milestones in the development of distributed file systems. The systems chosen are either in wide use or have influenced academic studies. Three popular distributed file systems: NFS, AFS, and Coda, are discussed in detail. Section 2.2 describes related work on concurrency control and replication in distributed database systems, another important type of storage systems. Section 2.3 discusses recent work on Grid computing and characterization of scientific workloads. Section 2.4 summarizes.

2.1 Distributed File Systems

2.1.1 NFS

The first distributed file system to realize widespread acceptance, Sun Microsystem's Network File System (**NFS**) [76, 102], was announced in 1984. NFS was originally developed at the University of California and Sun Microsystems for use on UNIX-based workstations, but NFS designers paid special attentions to portability and heterogeneity. An early and vocal proponent of open systems, Sun published the NFS protocol. Many implementations followed, and NFS became ubiquitous among networked workstations.

NFS employs a client/server architecture. The NFS protocol defines an RPC interface that allows servers and clients to communicate over a network. The protocol does not

specify how servers or clients should implement this interface. As a result, NFS runs seamlessly on a heterogeneous collection of computers.

In most operating systems, applications use system calls to access the file system. The native UNIX file system interface was replaced with an abstract Virtual File System (VFS) interface, which also became a de facto standard interface to different file systems [61]. VFS operations are either passed to a local file system, or passed to a separate component known as the NFS client, which takes responsibility for access to files stored on a remote server.

The NFS protocol was designed for simple error recovery. NFS servers are not required to maintain any contextual information about their clients. Each RPC request from a client contains all the information needed to satisfy the request. This stateless feature enables NFS to recover quickly from system and network failures. However, to some degree, functionality and consistency are sacrificed. In the modern version of NFS, NFS version 4 (NFSv4), this architectural shortcut is abandoned.

Version 3 of NFS (NFSv3) [77, 91], the most widely adopted version, was released in 1994. At that time, computers were less powerful than today's and networks were usually local area networks. The advent of wide area networks invalidated many assumptions inherent to the design of NFS and led to the development of NFSv4 [92, 109].

Like previous versions of NFS, NFSv4 has a straightforward design, robust error recovery, and independence of transport protocols and operating systems for file access in heterogeneous networks. NFSv4 also introduces some new features intended to improve Internet access and performance, such as compound RPC that groups multiple related operations into a single RPC request, delegation capabilities to enhance client performance for narrow data sharing applications, integration of file locking that can support different operating system semantics and error recovery, mandatory strong security via an extensible authentication architecture built on GSS-API [38], and the facilities to support file system migration and read-only replication. However, the published NFSv4 protocol does not provide mechanisms to support a global name space, transparent file system replication and migration, and mutable replication. These oversights impact the network transparency that NFSv4 file systems provide in wide area networks.

The absence of a global name space hampers collaborative work and sharing of data because users lack a common frame of reference. Lack of support for transparent file system

replication and migration complicates data distribution and management. The restriction to read-only replication sacrifices network transparency for write operations. To overcome these shortcomings, we develop a naming and replication scheme that provides the missing features, as described in detail in the following chapters.

2.1.2 AFS

Another widely used distributed file system, the Andrew File System (**AFS**), originated at Carnegie Mellon University in 1983 [81, 105]. The principal goal of AFS is to present a homogeneous, location-transparent name space to personal and time-sharing computers on a campus-wide network. However, AFS also pays special attention to scalability, administerability, and availability.

AFS clients aggressively cache files and directories on local disk. Servers record the files clients are caching, then execute callback routines to notify clients when cached data has changed. The AFS consistency model guarantees that a client opening a file sees the data stored when the most recent writer closed the file. However, this guarantee is hard to honor in a partitioned network when the callback operation can not be performed. The recent practice of caching partial "chunks" of a file further complicates matters.

To enhance availability and to distribute server load evenly, AFS employs read-only replication on data that is frequently read but rarely modified. Subtrees that contain replicated data may have read-only replicas at multiple servers, but there is only one read-write replica and all updates are directed to it. Propagation of changes to the read-only replicas is done by an explicit operational procedure.

To enrich file system administration, AFS organizes data into *volumes* [110]. A volume is a collection of files forming a partial subtree of the file system hierarchy. Independent of the physical configuration of the system, volumes provide a degree of transparency in addressing, accessing, and storing files. They also facilitate data movement and optimization of existing storage. However, it is a relatively heavyweight operation to configure a machine as an AFS server. This is in contrast to NFS, where any machine can easily export a subset of its local file system.

To support consistent file naming on a global scale, AFS employs a standard naming convention adhered to by cooperating but administratively autonomous sites, called *cells*

[124]. Each cell has its own servers, clients, system administrators, and users. The design allows transparent access to any file in the global name space from any AFS client, but does not require a site to relinquish administrative control over its own system. In particular, access to a site's files is still mediated by its own authentication system. Thus, globally centralized authentication services are avoided.

2.1.3 Coda

Coda [60, 106], a cousin of AFS, is designed for high data availability. It achieves this with two complementary mechanisms: server replication and disconnected operation.

In Coda, a volume is replicated at a set of servers, called a *volume storage group (VSG)*. For each volume from which it has cached data, a client keeps track of the subset of the VSG that is currently accessible. This subset is referred to as the *accessible volume storage group (AVSG)*, and is identical to the VSG in the absence of failures. A server is deleted from the AVSG whenever an operation on it times out. It is added back to the AVSG when the client is able to reestablish communication with the server.

Coda supports mutable replication with a variant read-one-write-all strategy. When servicing a cache miss, a client contacts all servers in its AVSG to make sure that all accessible replicas are synchronized. After that, the client obtains data from one server that has the latest copy in the AVSG. Once data is cached at the client, a callback channel is established with the selected server. Upon file close, updates are propagated to all available replicas.

If no servers can be contacted, a client resorts to disconnected operation, a mode of execution in which the client relies solely on cached data. The modified data are transferred to replication servers upon reconnection.

The strategy that allows an object to be be read and modified as long as one of its copies is accessible provides high availability. However, data consistency can not be guaranteed upon partition failures, as data copies might be updated concurrently in two or more partitions. Although the Coda group has investigated automated file and directory conflict detection and resolution mechanisms [63, 64], not all conflicts can be resolved. In many cases, user involvement is needed to get the desired version of data.

2.1.4 Other Distributed File Systems

Each of the three systems described above is important either because it is widely used in practice or because it occupies a prominent position in the space of distributed file system designs. There are many other distributed file systems. This subsection briefly describes some distinctive systems.

Sprite is a network operating system developed at the University of California at Berkeley [87]. In a Sprite network, most workstations are diskless, so large main-memory block caches are used to achieve high performance in the Sprite file system [84]. An important feature of Sprite is the exact emulation of Unix file system semantics. It achieves this by using file servers as centralized control points for cache consistency. Whenever a client opens or closes a file for reading or writing, it notifies the server that stores the file. A Sprite client usually caches pages of a file. However, when a file is open on multiple clients and at least one of them has it open for writing, Sprite disables client caching for the file. This strategy enables Sprite to provide consistency at the granularity of individual read and write operations.

Ficus [95] is a peer-to-peer file replication system developed at the University of California at Los Angeles. It uses a single-copy availability update policy with a reconciliation algorithm to detect concurrent updates and periodically restore the consistency of replicas.

Echo [23] and **Harp** [73] are two distributed file systems that use a primary copy scheme [13] to support mutable replication with variants of a view change protocol [8] for failure recovery. Both systems assume a local area network environment and use a single primary server to perform all updates and reads. Availability is enhanced by electing a new primary server if the original one fails. In some sense, performance can be improved by spreading the load so that different servers act as primaries for different partitions. However, a single primary server remains a potential bottleneck if it contains hot-spot partitions.

Recent years have seen a lot of work in peer-to-peer (P2P) file systems, including OceanStore [98], Ivy [82], and Pangaea [101]. These systems address the design of systems in untrusted, highly dynamic environments.

OceanStore [98] and **Ivy** [82] use a low-level lookup service for replica localization. The lookup service works on immutable objects only. To support update, OceanStore maps an object to a sequence of read-only versions and a new version of data is generated when an update request is received. Update messages are distributed among replicas along a

self-organizing multicast tree. Ivy places mutability in a per-client log-head. Every client writes its updates to its log. To satisfy read operations, a client accesses its own as well as other client's logs to construct the current state of the file system. Operation logs are also exchanged between replicas at that time.

Pangaea [101] replicates directories and files wherever and whenever they are accessed. Replicas are organized in a graph and each replica remembers its "nearby" peers. Pangaea provides optimistic data consistency. Updates can be made at any replica, later flooded to other replicas.

2.2 Concurrency Control and Replication in Distributed Database Systems

Distributed database systems and distributed file systems concern different objects and address different application requirements. In database systems, correctness in the presence of concurrency, distribution, replication, and failures is tied to the concept of transactions. It is normally a basic requirement that the currency control algorithms can support Atomic, Consistent, Isolation, and Durable (**ACID**) semantics [50]. Distributed file systems, on the other hand, usually allow coarser granularity of data sharing and are focused more on I/O performance. Furthermore, distributed file systems are typically designed for workloads in which read-write sharing is less common than those in distributed database workloads.

However, techniques for concurrency control and replication in distributed database systems provide insights for designing a replicated file system, as both systems need to address the distributed nature of the deployed environment. In fact, some techniques described below have been grafted to distributed file systems. For example, optimistic concurrency control has been borrowed by several distributed file systems, such as Coda and Ficus, to support mutable replication. Ivy and OceanStore use multiversioning techniques to solve concurrent write problem. In later chapters, we use the view change rules developed by El. Abbadi et al. [8] for the design of a failure handling scheme, and a hierarchical replication control protocol similar to hierarchical locking found in transactional systems.

Most concurrency control algorithms in distributed database use variants of three techniques: **locking**, **timestamp**, and **optimistic concurrency control**.

2.2.1 Locking

The fundamental locking algorithm is **two-phase locking (2PL)** [40]. It synchronizes reads and writes by explicitly detecting and preventing conflicts between concurrent operations. With 2PL, before reading a data item X, a transaction must own a readlock on X; similarly, before writing into X, it must own a writelock on X. The ownership of locks is governed by two rules: First, different transactions cannot simultaneously own conflicting locks [1]. Second, once a transaction surrenders the ownership of a lock, it may not obtain additional locks until it has released all locks.

The requirement of obtaining a lock before accessing any data item introduces performance overhead. To improve concurrency and performance of simultaneous transactions, many modern transactional systems use **hierarchical locking** [51] to support multiple granularities of locking unit.

In distributed database systems, two-phase locking is often used with a **primary copy** scheme [13]. In this model, one copy of each logical data item is designated the primary copy; before accessing any copy of the logical data item, the appropriate lock must be obtained on the primary copy.

2.2.2 Timestamp

Timestamp ordering is a technique whereby a serialization order is selected a priori and transaction execution is forced to obey this order [97]. A basic timestamp based algorithm works as follows.

Each transaction is assigned a timestamp at startup. Each object is given two timestamps: a read timestamp and a write timestamp. A transaction is allowed to read an object if its timestamp is later than the object's write stamp; otherwise, the transaction is aborted. After a transaction reads an object, the object's read timestamp is set to the later of the object's read timestamp and the transaction's timestamp. A transaction is allowed to write an object if its timestamp is later than the object's read stamp; otherwise, the transaction is

[1] Here, the definition of conflicting locks depends on the type of synchronization being performed: for read-write synchronization, two locks conflict if both are locks on the same data item, one is a readlock, and the other is a writelock; for write-write synchronization, two locks conflict if both are writelocks on the same data item.

aborted. After a transaction writes an object, the object's write timestamp is set to the transaction's timestamp. With these rules, the system guarantees that the conflicting operations are processed in timestamp order.

An important variant of the timestamp algorithm is **multiversion** concurrency control [20]. In this model, the system maintains several versions for an object, each with a write timestamp; the system guarantees that a transaction reads the most recent version whose timestamp precedes the transaction's timestamp.

Multiversioning can be used with a **voting** algorithm [49] to synchronize transactions in a replicated database system. A transaction is required to write a majority of copies to modify an object and read at least enough copies to make sure that one of the copies is current. Although interesting research, this model is not attractive in most situations because reading an object requires reading multiple copies.

Instead, a special type of voting is widely used, called **read-any/write-all**: to read an object, a transaction can read any copy, but to write an object, it must write all copies. An example that uses this technique is the **view change** protocol proposed by El. Abbadi et al. [8]. In that protocol, each replica maintains a view consisting of the replicas that it believes it can communicate with. Upon failure, a new view is formed among the replicas in the majority partition. Within each view, the "read-any/write-all" rule is used. The protocol is superior when reads are much more frequent than writes.

2.2.3 Optimistic Concurrency Control

In **optimistic concurrency control**, transactions proceed in three phases: *Read*, *Validation*, and *Write* [65]. When a transaction executes in *Read* phase, it reads values from the database but writes values to a private workspace. When the transaction wants to commit, it enters the *Validation* phase, during which the database system checks whether the transaction could possibly have conflicted with any other concurrently executing transactions. If there are possible conflicts, the transaction is aborted. Otherwise, the transaction moves to the *Write* phase, in which it copies the written data in its private workspace into the database.

The basic premise in optimistic concurrency control is that most transactions do not conflict with other transactions. If there are many conflicts, the cost of repeatedly aborting transactions can significantly hurt performance.

2.3 Scientific Applications and Grid Computing

2.3.1 Data Access in Grid Computing

Grid-based scientific collaborations are characterized by geographically distributed institutions sharing computing, storage, and instruments in dynamic virtual organizations [30, 45]. By aggregating globally distributed resources, Grid middleware provides an infrastructure for computations far beyond the scope of a single organization.

Grid computations feature high performance computing clusters connected with long fat pipes, a significant departure from the traditional high-end setting of a collection of nodes sited at one location connected by a fast local area network. This difference introduces new challenges in storage management, job scheduling, security provision, etc., stimulating growing research in these areas. In particular, the need for flexible and coordinated resource sharing among geographically distributed organizations demands efficient, reliable, and convenient data access and management schemes to ease users' efforts for using Grid data.

At present, the primary data access method used for Grid computing is **GridFTP** [12]. Engineered with Grid applications in mind, GridFTP has many advantages: automatic negotiation of TCP options to fill the pipe, parallel data transfer, integrated Grid security, and partial transfers that can be resumed. In addition, as an application, GridFTP is easy to install and support across a broad range of platforms.

However, just as FTP is inadequate for transparent distributed sharing, the limited semantics that GridFTP offers are not capable of providing the sharing functionality that many Grid applications require.

For example, in a common use case of the Grid, a scientist wants to run simulations on high performance computing systems and view results on a visualization system. With the Grid technologies available today, the scientist submits the job to a Grid scheduler, such as Condor-G [47]. The Grid scheduler determines where the job will be run, pre-stages

the input data to the running machine, monitors the progress of the running job and when the job is complete, transfers the output data to the visualization system through GridFTP. The output data is reconstructed with visualization tools in the visualization site. Finally, results are returned to the scientist after reconstruction.

The scenario has the advantage of enabling a scientist access to more computing resources and speeding up his simulation. However, the whole process is performed in batch mode. The scientist cannot view intermediate results before the entire scheduled job completes. Scientific simulations are often iterative and interactive processes, so the need to wait for hours or days to examine experimental results is very inconvenient. To overcome this disadvantage, the Grid infrastructure requires more flexible data distribution and sharing in its middleware.

Figure 2.1 shows a different picture for this kind of scenario: By employing a replicated file system, the intermediate outputs of simulation are automatically distributed to the visualization system and the scientist's computer. The scientist can view intermediate results and determine if parameters or algorithms need to be adjusted. If so, he can update them from his local computer and restart the simulation on the remote computing site.

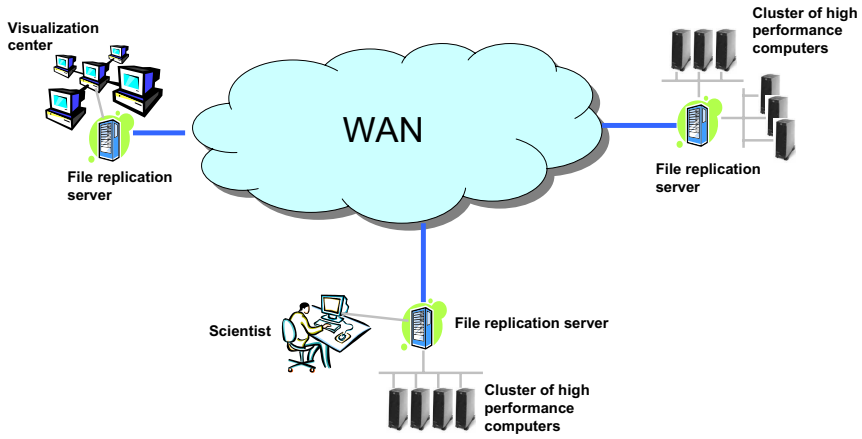


Figure 2.1: A Grid use case example. By employing a replicated file system, the scientist can view the intermediate results of his simulation and adjust the experiment in real time.

Various middleware systems have been developed with the goal of facilitating data access on the Grid. For example: **Storage Resource Broker (SRB)** [16] introduces a metadata catalog service to allow location-transparent access for heterogeneous data sets. **NeST** [17], a user-level local storage software system, provides best-effort storage space

guarantees, mechanisms for resource and data discovery, user authentication, quality of service, and multiple transport protocol support, with the goal of bringing appliance technology to the Grid. The **Chimera** system [46] provides a virtual data catalog that can be used by application environments to describe a set of application programs, then tracks all the data files produced by executing those applications. Chimera is motivated by the observation that much scientific data is obtained not from measurements but rather derived from other data by the application of computational procedures, which implies the need for a flexible data sharing and access system.

A common feature missing among these middleware solutions is the lack of fine-grained data sharing semantics. Furthermore, most of these systems provide extended features by defining their own APIs. In order to use them, an application has to be coded and linked for their libraries. Consequently, scientific researchers are generally hesitant to install and use these Grid software systems.

The **GridNFS** [4] project developed at CITI aims to provide file system semantics and high performance data access for Grid applications. The mutable replication support presented in this dissertation is a major part of GridNFS. My design fits the above picture by providing the following features. First, fine-grained mutable replication allows users to perform data modifications easily. Second, my design allows a client to access data from a nearby server. In common cases, the overhead for supporting mutable replication is negligible. Third, it provides a canonical global name space. Applications access data through logical names, without needing to know the physical location of data. Fouth, it complies with the POSIX API, allowing easy deployment and providing the possibility to employ unmodified application software on the Grid. Throughout this dissertation, we will discuss in detail how the system provides these features.

2.3.2 Scientific Workload Studies

The growing need for large-scale scientific collaborations has stimulated a proliferation of research in Grid computing and scientific workload studies. For the purpose of brevity, we summarize only two recent works that are directly related to our study.

Thain et al. studied the workload characteristics of six batch-pipelined scientific applications [113]. The applications they characterize are chosen from different scientific

disciplines. The studied workloads demonstrate three common characteristic behaviors. First, small initial inputs are usually expanded by early stages into large intermediate results, which are often reduced by later stages to small results. Second, although users tend to identify large data collections needed by an application, in a given execution, applications usually select a small working set. Third, significant data sharing is observed for users who often submit large numbers of very similar jobs that access similar working sets.

Holtman et al. investigate the data processing requirements of CMS experiments [3] and the workload characteristics expected once the Large Hadron Collider accelerator starts running [56]. Regarding file access, they point out that the CMS workloads will be dominated by reading. The stepwise refinement of algorithms will lead to a workload where series of jobs are run over the same input data, with each job containing the refined code or parameters. Because of the large data reduction factor in several important types of CMS analysis, the sparseness of the input datasets increases as an analysis progresses. CMS expects to access files through regular POSIX I/O calls so that in the future, CMS can use commercial software components that do not require re-linking with special "Grid version" libraries. CMS has a strategy of moving code to data. However, sometimes data may also have to be moved to a location where code runs. In such cases, not all data used by a CMS executable is necessarily available locally when the execution starts. Sometimes, CMS applications need random access to data from data sets that are too large to stage to every machine in a site. Such use cases require accessing files on a large file system local to the site but not local to the machine.

2.4 Summary

This chapter surveys the data access requirements of the emerging Grid computing and the related work on distributed file systems and distributed database systems.

Grid computation is characterized by high performance computing clusters distributed over wide area networks. However, most existing scientific applications were developed for local running environments. They encounter performance and reliability challenges when their accessed data is widely distributed. Currently, the primary Grid data access method is GridFTP. As a remote file transfer protocol, GridFTP does not support consistent data sharing, which hinders the re-use of the existing software in Grid computing. Ideally, we want

to allow users and applications to access Grid data as simply as they access them locally. This introduces the need for a large-scale replicated file system that allows applications to access remote data efficiently and through the conventional file system semantics.

File system replication has been studied for a long time. However, existing solutions either forsake read/write replication totally or weaken consistency guarantees. In contrast, our system supports mutable replication with strong consistency guarantees. This allows applications to access remote data reliably and easily. A primary concern of such a system is the performance overhead it might add to applications. The main contribution of my thesis is the design and development of a replicated file system that provides consistent mutable replication in a large-scale file system yet adds little overhead to application performance.

CHAPTER 3

Network Transparency and Data Access in Grid Computing

As we have seen, existing data access methods that work well in local area networks or for applications that require limited data sharing fail to satisfy the requirements of global scientific collaborations. To facilitate distributed computing over wide area networks and to improve network transparency of data access in global scientific collaborations, we developed a replicated file system that provides users high performance data access with conventional file system semantics.

Network transparency allows applications to access remote resources through network without needing to know whether they are remote or local. In a distributed file system, network transparency involves three aspects: naming, performance, and failure resiliency.

Naming plays an important role in distributed file systems. A file system is said to provide **name transparency** if it satisfies the following three requirements. First, any file is accessible from any location. Second, the same name is used at every location. And third, file location is not reflected in the name. Section 3.1 discusses in detail a naming scheme that achieves name transparency by supporting a single shared global name space and location independent naming.

Good performance is always a critical goal. A distributed file system provides **performance transparency** if it hides access latency for remote resources.

In a distributed file system, a client's access can be interrupted by any number of machine or network failures. **Failure transparency** requires that the system hides a failure and recovery of resources.

Replication is a common technique for improving performance transparency and failure transparency in distributed systems. There are two primary reasons for replicating data. First, replication improves performance by allowing access to distributed data from nearby or lightly loaded servers. Second, replicating data improves availability in the face of failure by allowing users and applications to switch from a failed server to a working replica. Throughout the latter chapters of this dissertation, we describe a replicated file system with special attentions to performance evaluation and mechanisms for handling failures.

In distributed systems, the main problem introduced by replication is maintaining consistent state. Whenever a replica is updated, that replica becomes different from the others. To keep replicas consistent, we need to propagate updates in such a way that temporary inconsistency is not observable. More challenging, we need to maintain consistency without penalizing performance, even in large-scale distributed systems.

In Chapters 4 to 6, we present a read/write (i.e., mutable) replicated file system that balances the tradeoff among consistency, performance, and failure resilience by offering applications stringent yet flexible consistency guarantees. To build a foundation for the detailed discussion, we clarify our design principles in terms of consistency guarantees, failure model, security considerations, and performance tradeoff in Section 3.2.

3.1 Global Naming

A file system provides access to named files. Naming therefore plays an important role in distributed file systems. In wide area networking, several principles guide the design of a naming scheme.

First, a global name space for all files in the system is desirable. By providing a common frame of reference, a global name space encourages collaborative work and sharing of data. Users on any client, anywhere in the world, can then use an identical rooted path to refer to a file or directory.

Second, location independent naming facilitates transparent migration and replication. A distributed system provides transparent migration and replication by using abstraction layers to hide the fact that a resource is migrated or replicated, and thus present users the appearance of a single unified system. With location independent naming, files are not bound by name to individual servers, so they can be transparently migrated or replicated.

Third, name space operations should scale well in the face of wide or massive distribution. A scalable distributed system is one that can easily cope with the addition of users and sites and whose growth involves minimal expense, performance degradation, and administrative complexity. These requirements suggest that a naming scheme should adopt a hierarchical architecture and allow delegation of administration.

Fourth, a naming scheme should be easy to apply in practice so that it can be used in reality. Many of our design decisions are motivated by this principle. E.g., we avoid employing a new directory service to support a global name space and location independent naming, but utilize the existing Domain Name Service (DNS) [78] that has been pervasively deployed and used for years.

In the rest of this section, we describe the design and implementation of a naming scheme that provides these features.

3.1.1 Design

Following the principles described above, we develop a naming scheme that allows organizations to export their data autonomously in a single shared global name space, and supports transparent migration and replication through location independent naming.

Global Name Space and File System Migration & Replication

We extend the Linux NFSv4 implementation to provide NFSv4 clients a single shared global name space. By convention, a special directory /nfs is the global root of all NFSv4 file systems. To an NFSv4 client, /nfs resembles a directory that holds recently accessed file system mount points.

Entries under /nfs are mounted on demand. Initially, /nfs is empty. The first time a user or an application accesses an NFSv4 file system, the referenced name is forwarded to a daemon that queries DNS to map the given name to one or more file server locations, selects a file server, and mounts it at the point of reference.

The format of reference names under /nfs follows Domain Name System conventions. We use SRV Resource Record (RR) [79] for server location information. A RR in DNS can map a reference name to multiple targets, in this case replicated file servers holding the same data, which is a necessary element in transparent file system migration and repli-

cation. When a file system is replicated to a new server, the administrator updates the DNS entry, adding a mapping from the reference name of the file system to the new server location. Similarly, when a file system is migrated to another server, the old mapping is updated to point to the new server. Once the migration completes, the old server returns NFS4ERR_MOVED, a special NFSv4 protocol value intended for migration support, for subsequent client requests. Upon receiving this error, the client queries DNS for the new file system location and retries the request with the specified server.

Here is an example of a pathname in the global name space:
`/nfs/umich.edu/lib/file1`, where `umich.edu` refers to the NFS file system provided by the University of Michigan and `/lib/file1` is the path under that file system.

File System Name Space and Directory Migration & Replication

The file system name space provided by an NFSv4 server is rooted in a *pseudo file system*. A pseudo file system glues all of the directory hierarchies exported by an NFSv4 server into a single tree rooted at `/`. Portions of the server name space that are not exported are bridged so that an NFSv4 client can browse seamlessly from one export to another. Exported file system structures are controlled by servers. Thus, a server can dictate a common view of the file system that it exports. This feature is essential for the support of a single global name space, and reflects the intention of the NFSv4 protocol designers to provide that support.

We implement directory migration and replication by exporting a directory with an attached reference string that includes information on how to get directory replica locations, such as replica lookup methods and lookup key. To allow an organization to maintain replica location information flexibly, we support four kinds of replica lookup methods: LDAP, DNS, Configuration File, and Server Redirect. The comparison and analysis of these methods are presented in Table 3.1.

When a replicated directory is accessed by a client for the first time, the server resolves the replica locations of that directory with the attached reference string. It then sends this information to the client through the FS_LOCATIONS attribute, as the NFSv4 protocol specifies. Upon receiving the replica locations, the client selects a nearby replication server, mounts it at the place of reference, and continues its access.

Query Method	Advantage	Disadvantage
DNS RR	widely deployed, no need for extra installation, hierarchical structure, efficient for simple queries	poor support for frequent updates, remote updates, complex queries, and security; more burdens on DNS
LDAP	good support for frequent updates, remote updates, complex queries, and security	less efficient for simple queries; extra requirement to maintain LDAP service
Configuration File	easy to use	does not scale
Server Redirect	no lookup cost	does not scale

Table 3.1: Comparison of different lookup methods.

When a directory is migrated to another server, the old server returns NFS4ERR_MOVED error for subsequent directory requests. Upon receiving the error, the client obtains the new location of the migrated directory by examining the content of the FS_LOCATIONS attribute and connects to the specified server.

3.1.2 Implementation

This subsection reports the implementation details of the presented naming scheme. We first describe the mechanisms for supporting a global name space. We use a modified Automount daemon on the client side to intercept access requests, perform DNS lookups, and automatically mount and unmount NFSv4 servers. These extensions and our specified naming convention provide NFSv4 clients a unified global name space. Following that description, we discuss implementation details for the support of directory replication.

Implementation of Top-level Namespace

Automount and AutoFs are tools that allow users of one machine to mount a remote file system automatically at the instant that it is needed. Automount, often referred as AMD, is a daemon that installs AutoFs mount points and associates an automount map with each AutoFs mount point. AutoFs is a file system implemented in the kernel that monitors attempts to access a subdirectory within a designated directory and requests AMD to perform mounts or unmounts. Upon receiving a mount request from the kernel, the AMD uses the automount map to locate a file system, which it then mounts at the point of reference within the AutoFs file system. If the mounted file system is not accessed for a specified interval, AutoFs instructs the daemon to unmount it.

Automount supports various mapping methods. However, support for DNS mapping is not provided in the current implementation. We extend the Automount daemon program to support DNS mapping. The global root directory of NFS, `/nfs` by convention, is made an AutoFs mount point with DNS mapping as the associated automount mapping method.

Maintenance and Lookup of Directory Replica Locations

NFSv4 uses `exportfs` utilities on the server side to export a directory. In the Linux kernel, an `export` structure is maintained for each currently accessed directory export. We extend the `exportfs` interface so that the reference string of a replicated directory can be passed into kernel. After that, the reference string is maintained in the corresponding `export` structure.

In our implementation, a server caches replica location information in the kernel. When an NFS client encounters a directory export with an attached reference string, the server calls cache lookup with the reference string as the lookup key. If there is a cache hit, the cached value is returned. If a cache miss occurs, an upcall is made to a user-level handler, which performs the lookup and adds the queried data to the cache.

3.2 NFSv4 Replication

3.2.1 Consistency Guarantees

The great challenge when replicating files is keeping copies synchronized as updates occur. In a distributed storage system, synchronization semantics are expressed in terms of the **consistency guarantees** granted to applications sharing data. Essentially, a consistency guarantee is a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly.

In distributed file systems, various consistency guarantees have been introduced. The most stringent guarantee, **strict consistency**, assures that all clients see precisely the same data at all times. Although semantically ideal, strict consistency is detrimental to performance and availability in networks with high latency, many clients, and the potential for partition. On the other end of the spectrum, consistency guarantees are abandoned altogether in **optimistic replication**. Some examples include P2P systems that strive to max-

imize availability [82, 95, 101] and distributed file systems that use heuristics to address conflicts when inconsistency happens [64, 106]. To balance the benefit of replication with the cost of guaranteeing consistency, some distributed file systems provide **read-only** access to replicated files, side-stepping update consistency problems altogether [11,105,118].

We observe that although optimistic replication has been widely studied, few applications in reality can deal with the conflicts that might happen. Even if applications can provide such support, conflict resolution must be performed carefully; otherwise, the cost to reproduce data, if possible, can be considerable. The lack of consistency guarantees makes it infeasible for scientific collaborations that require reliable and coordinated data access.

For its superior read performance, read-only replication has been favored in the current Grid experimental platform [30]. With read-only replication, once a file is declared as shared by its creator, it cannot be modified. An immutable file has two important properties. First, its name may not be reused, and second, its contents may not be altered. While simple, read-only replication has several deficiencies. First, read-only replication fails to support complex sharing behavior, such as concurrent writes. Second, to guarantee uniqueness of file names, file creation and retrieval require a special API, which hinders using the software developed in the traditional computing environment for global collaborations.

These considerations argue that global scientific collaborations require consistent mutable replication. A desirable solution is not to bypass consistency problems, but to develop an advanced system that supports write operations and consistency guarantees without hurting ordinary case performance.

In considering the options for consistency guarantees, the “principle of least surprise” (which is a colloquial way of saying “transparency”) argues for the strictest possible semantics, yet this choice is rarely offered as the default. Even in local file system sharing, applications typically read and write through private, thus potentially stale, I/O buffers. It is tacitly understood that weaker guarantees, such as “ordered writes” or “close-to-open” semantics suffice for most applications, and that applications requiring more stringent guarantees usually provide for themselves with explicit synchronization mechanisms, such as lock or sync system calls.

Our design also follows this strategy. We provide users and applications stringent yet flexible consistency semantics. The replication system we develop supports three consis-

tency guarantees: **sequential consistency**, **close-to-open semantics**, and **synchronized access**.

In sequential consistency, distributed nodes do not *necessarily* see updates simultaneously, but they are guaranteed to see them in the same order [67]. We take advantage of this relaxed consistency requirement to design and build a system that imposes no penalty on reading files.

The conventional NFS consistency model, the so-called "close-to-open" semantics [109], guarantees that an application opening a file sees the data written by the last application that writes and closes the file. This strategy has proved to provide sufficient consistency for most applications and users [91]. We also consider close-to-open semantics to be important for many Grid applications and provide mechanisms for supporting this consistency model.

Certain applications, e.g., database, require synchronized access (i.e., strict consistency). That is, if two applications on different clients have the same file open concurrently, a read of one observes the written data by the other. To enable correct execution of such applications, we offer this consistency option as well.

By default, our replication system guarantees close-to-open semantics, i.e., the same consistency semantics that the standard NFSv4 protocol provides. To offer better performance and availability for applications that require ordered writes only, we allow users and applications to explicitly demand sequential consistency as a mount option. If a file system is mounted with this option, the server that the client writes to coordinates other replication servers to enforce sequential consistency, with a mechanism slightly different from the replication control protocol that is used for guaranteeing close-to-open semantics. Chapter 4 discusses this mechanism in more detail.

In NFSv4, a client's write request includes a special flag that declares whether the written data is to be synchronized. An NFSv4 client sets this flag on user's behalf if the user issues an fsync system call or the written file was opened with `O_SYNC` flag set. Upon receiving a synchronous write request from a client, our system guarantees synchronized access for the written file, which provides stronger consistency guarantee but less failure resilience.

In Chapter 4, we describe in detail the approaches for providing these consistency semantics.

3.2.2 Failure Model

One objective of our work is to increase data availability with a replication scheme that tolerates a large class of failures. There is a well-studied hierarchy of failures in distributed systems: omission failure, performance failure, and Byzantine failure [35].

An **omission failure** occurs when a component fails to respond to a service request. Server crash and network partition are typical omission failures. A **performance failure** occurs when a system component fails to respond to a service request within a specified time limit. Occasional message delays caused by overloaded servers or network congestion are examples of performance faults. In **Byzantine failure**, components act in arbitrary, even malicious ways [71]. Compromised security can lead to Byzantine failure.

Although security breach is increasingly common in the Internet, Byzantine failure is beyond the scope of the work presented in this dissertation. Rather, we rely on the use of secure RPC channels, mandatory in NFSv4, for server-to-server communication.

When considering omission failure and performance failure, we expect that the latter happens more frequently as a system scales to wide area networks. To develop a replication system that performs well in the face of typical Internet conditions, it is important that the performance of our system is insensitive to temporary message delays. We achieve this by allowing a client write request to be acknowledged either immediately or as soon as a specified number of replication servers have acknowledged the update. Consequently, even though occasional message delays can cause a handful of replication servers to respond sluggishly, the penalty to application performance is small. Furthermore, to tolerate long-term omission failures, we develop a failure recovery scheme that allows the system to operate continuously as long as a majority of replication servers are in working order.

3.2.3 Security Considerations

As previous versions of NFS, NFSv4 relies on the underlying security model of RPC for its communication. However, NFSv4 mandates the use of strong RPC security flavors to improve security. NFSv4 requires support for Kerberos Version 5 [62] and LIPKEY [39], and allows other mechanisms to be specified and used as well. To enable in-band security negotiation, NFSv4 adds a new operation that allows a client to query a server about which

security mechanisms must be used for accessing the server's file system. With this support, a client can securely match the security policies specified at both the client and server.

Our replication extension is a server-to-server communication protocol and does not affect the NFSv4 security model between a server and a client. Furthermore, our system is implemented on top of RPC. This allows us to use the security mechanisms provided by RPC for server-to-server authentication, integrity, and privacy. As mentioned in Section 3.2.2, we assume no Byzantine failures. That is, a replication server is trusted by other peers as long as it can authenticate itself with the specified security policies on other servers.

3.2.4 Performance Tradeoff

Good performance is a critical goal for all file systems. Our design follows a simple but fundamental rule: make common accesses fast. Based on the insights from workload analysis of real file systems [15, 24, 25, 99] and the recent workload studies of the future global-scale scientific applications [56, 113], the following cases are considered, ordered by their expected frequencies:

- **Exclusive read:** most common. Support for replication should add negligible overhead to unshared reads.
- **Shared read:** common. Blaze observes that files used by multiple workstations make up a substantial proportion of read traffic [24, 25]. E.g., files read by more than one user make up more than 60% of read traffic, and files read by more than any ten users make up more than 30% of read traffic. This motivates us to avoid any cost penalty for shared reads.
- **Exclusive write:** less common. File system workload studies show that writes are less common than reads [15, 24, 25, 99]. When we consider access for data that needs to be replicated over wide area networks, this difference can become even larger. This allows us to design a replicated file system with good average performance at the cost of slightly more expensive data updates.
- **Write with concurrent access:** infrequent. A longer delay due to enforcing consistency can be justified when a client accesses a file or a directory that is being updated by another client.

- **Server failure and network partition:** rare. Special failure recovery procedures can be used when a server crashes or a network partitions. During the time of failure, write accesses might even be blocked without doing much damage to overall performance, if stringent consistency must be guaranteed.

To guarantee data consistency without penalizing exclusive reads and shared reads, our replication system uses a variant *primary-copy scheme* with operation forwarding to coordinate concurrent access during file modifications. Under a conventional primary copy approach, a primary server is statically assigned for each mount point during configuration so all write requests under a single mount point go to the same primary server. In contrast, in our system the server to which a client sends the first write request is elected as the primary server for the file or the directory to be modified. With no writers, a client's read requests are served on a nearby server, as in a read-only replication system. Furthermore, in our system, failure detection and recovery are driven by client accesses. No heartbeat messages are needed. When a primary server fails, recovery mechanisms generate a replacement server that then synchronizes other replication servers to a consistent state. With these approaches, the five cases discussed above introduce overheads inversely proportional to their expected frequencies, roughly speaking.

3.2.5 Summary

Following the principles discussed above, we have designed a mutable replicated file system targeted to facilitate the data access and management in the emerging wide-area collaborations. The system is well suited for read-dominated applications, as mutable replication adds no cost to exclusive reads or shared reads. It supports file modifications by using a variant primary copy scheme with operation forwarding to guarantee consistency. Three consistency models are supported: sequential consistency, close-to-open semantics, and synchronized access. Thus, the overhead to enforce stringent consistency is induced only when users demand it. The system tolerates a large class of component omission and performance failures, even when these lead to network partition.

We have implemented our replication scheme as an extension to NFSv4, the emerging Internet standard for distributed filing. Our implementation utilizes several existing features provided by NFSv4, such as client side failure recovery, mechanisms to support

read-only replication, and compound RPC. We pay special attentions to keeping the protocol simple so that it is potential to standardize the extensions as an IETF minor version of NFSv4. Furthermore, the replication scheme requires minuscule extensions on client-side implementation, which makes it easy and practical to deploy. In the rest of this dissertation, we refer to the developed replicated file system as **NFSv4.r** for short.

CHAPTER 4

A Fine-grained Replication Control Protocol that Supports Consistent Mutable Replication

To meet the rigorous demands of large-scale data sharing in global collaborations, we developed a replication scheme for NFSv4 that supports mutable replication without sacrificing strong consistency guarantees. In this chapter, we present the design, implementation, and evaluation of a mutable replication protocol that guarantees stringent yet flexible consistency semantics by electing a primary server upon client updates at the granularity of a single file or directory. We refer to it as *fine-grained replication control protocol* in the following discussion.

The rest of the chapter is organized as follows. Section 4.1 describes a replication scheme that guarantees sequential consistency. Based on that, Section 4.2 discusses the approaches to further enforce close-to-open semantics and synchronized access. After that, Section 4.3 reports the experimental results, and Section 4.4 summarizes.

4.1 Sequential Consistency

To support read/write replication, we need mechanisms to distribute updates when a replica is modified and to control concurrent accesses when writes occur. Since mutable replication must not affect exclusive or shared read performance, we adopt an extended primary copy scheme with operation forwarding to coordinate concurrent writes. Under the conventional primary copy approach, a primary server is statically assigned for each mount point during configuration so all write requests under a single mount point go to the

same primary server. In contrast, in our system, the server to which a client sends the first write request is elected as the primary server for the file or the directory to be modified. This strategy provides the following advantages.

First, the overhead to support mutable replication is induced only when there are writers present. With no writers, the system resembles a read-only replication system, i.e., a client accesses data from any nearby replication server.

Second, a primary server is selected at the granularity of a single file or directory, which allows fine-grained load balancing.

Third, in our scheme, a primary server is dynamically chosen upon file or directory modification. Hence, in the most common case (exclusive write), a client's write requests are served by a nearby primary server. The solution is well suited to the Grid computing environment because a replica can be dynamically created and an optimal primary server can be chosen for a file or directory in flight.

The remaining of this section proceeds as follows. We introduce the replication control protocol in Section 4.1.1. Section 4.1.2 then presents the primary server election algorithm. Section 4.1.3 discusses the update distribution strategy. In Section 4.1.4, we describe the handling of various kinds of failures.

4.1.1 Replication Control Protocol

Sequential consistency guarantees ordered writes; although replication servers do not necessarily process updates simultaneously, they are guaranteed to see all of them, and in the same order. The system works as follows.

When a client opens a file for writing, it sends the open request to the NFS server that it has selected for the mount point to which the file belongs. An application might open a file in write mode without actually writing any data for a long time, e.g., forever, so the server does nothing special until the client makes its first write request. When the first write request arrives, the server invokes the replication control protocol, a server-to-server protocol extension to the NFSv4 standard.

First, the server arranges with all other replication servers to agree on its primary role. Once consensus is achieved, all other replication servers forward client write requests for that file to the primary server. The primary server distributes (ordered) updates to other

servers during file modification. When the file is closed (or has not been modified for a long time) and all replication servers are synchronized, the primary server notifies the other replication servers that it is no longer the primary server for the file.

Directory updates are handled similarly, except for the handling of concurrent writes. Directory updates complete quickly, so a replication server simply waits for the primary server to relinquish its role if it needs to modify a directory undergoing change. For directory updates that involve multiple objects, a server must become the primary server for all objects. The common case for this is rename, which needs to make two updates atomically. To prevent deadlock, we group these update requests and process them together.

4.1.2 Primary Server Election

Two (or more) servers may contend to become the primary server for the same object (file or directory) concurrently. To guarantee correctness of our replication protocol, we need to ensure that no more than one primary server is chosen for a given object, even in the face of conflicts and/or failures. This problem is a special case of the extensively studied consensus problem.

In the consensus problem, all correct processes must reach an agreement on a single proposed value [41]. Many problems that arise in practice, such as electing a leader or agreeing on the value of a replicated object, are instances of the consensus problem. In our case, if we assign each replication server a unique identifier, the primary server election problem is easily seen to be an instance of the consensus problem: electing a primary server is equivalent to agreeing on the identifier of a primary server.

Achieving consensus is a challenging problem, especially in an asynchronous distributed system. In such a system, there is no upper bound on message transmission delays or the time to execute a computing step. A good consensus algorithm needs to maintain consistency, i.e., only a single value is chosen, and to guarantee progress so that the system is eventually synchronous for a long enough interval [37]. Unfortunately, Fischer et al. show that the consensus problem cannot be solved in an asynchronous distributed system in the presence of even a single fault [42].

Observing that failures are rare in practice, candidate consensus algorithms have been proposed to separate the consistency requirement from the progress property [28, 68–70,

```
Upon receiving a client update request, initiate primary
server election if the object's primary server is NULL
set the object's primary server to MyID  // ack self
loop until all active servers ack
    propose <MyID, object> to unacked servers
    wait until all those servers reply or timeout
    if the number of acks received is less than majority then
        identify competitors from the replies
        if any competitor is accepted by a majority of servers, or
        any competitor's identifier is larger than MyID, then
            set the object's primary server to NULL
            send abort <MyID, object> to all acked servers
            wait until the object's primary not NULL or timeout
            exit loop
    else  // have collected acks from majority
        mark timed out servers inactive

Upon receiving propose <ServerID, object>
if the object's primary server is NULL then
    set the object's primary server to ServerID
    send ack
else
    send nack <the object's primary server>

Upon receiving abort <ServerID, object>
if the object's primary server equals to ServerID then
    set the object's primary server to NULL
```

Figure 4.1: Primary server election. This pseudocode sketches the election protocol that ensures consistency. To guarantee progress in case of failure, we use the view change protocol analyzed by El-Abbadi to maintain liveness of replication servers. Section 4.1.4 discusses failure handling in more detail.

74]. That is, while consistency must be guaranteed at all times, progress may be hampered during periods of instability, as long as it is eventually guaranteed after the system returns to normal. Following this principle, we implement a primary server election algorithm that achieves the lower time bound of Fast Consensus [69]. The algorithm assumes that all messages are delivered in order, which can be achieved by including a serial number in each message or through a reliable transport protocol. Figure 4.1 presents the pseudo code of the algorithm.

It is easy to verify that the algorithm satisfies the consistency requirement: a primary server needs to accumulate the acknowledgments from a majority of the replication servers and a replication server cannot commit to more than one primary server, so only a single primary server is elected for a given object. Furthermore, for the common case - no failures and only one server issues the proposal request - primary server election completes with

only one message delay between the elected primary server and the farthest replication server. In fact, since the server can process the client's update request as soon as it receives acknowledgments from a majority of the replication servers, the conflict-free and failure-free response time is bounded by the largest round-trip time (RTT) separating the primary server and half of the nearest replication servers.

If multiple servers compete to be the primary server for an object, it is possible that none of them collects acknowledgments from a majority of the replication servers in the first round of the election. Absent failure, the conflict is quickly learned by each competing server from the replies it receives from other replication servers. In this case, the server with the largest identifier is allowed to proceed and its competitors abort their proposals by releasing the servers that have already acknowledged. A server that aborts its election then waits for a proposal request from the stronger server. The waiting is associated with a timer to guarantee progress in case that the stronger server fails. Upon receiving the anticipated proposal, the server acknowledges the request. Once consensus is achieved, the server forwards the client update to the newly elected primary.

In the presented algorithm, the winner of the competition continues to send proposal requests to replication servers that have not acknowledged its role, subject to timeout. However, the abort request from a yielding competitor may arrive at such a replication server after several rounds of proposal distribution, resulting in redundant network messages. Contention is infrequent, but happens from time to time in Grid computing. E.g., when we start a task on multiple computing nodes, clients connect to different replication servers can simultaneously create their own output files under the same directory, which causes these replication servers to issue primary server election requests for that directory around the same time. The situation can be improved with a small optimization in the second round of the election: the winning server can append the replies it has collected in the previous rounds to its subsequent proposals. With this information, a server that receives a late-round proposal can learn that the server it is currently treating as primary will soon abort the election. Thus, it can briefly delay replying to the new proposal, increasing the chance that the object is released by the old primary server before responding to the late-round proposal. We leave the detailed discussion of failures to Section 4.1.4, but point out that when the system is free of failure, primary server election converges in two message delays even in the face of contention.

4.1.3 Update Distribution

There are two primary reasons to maintain consistency among replication servers: first, to guarantee correctness during concurrent writes; and second, to guarantee the durability of data (i.e., no data lost) in the face of failure. In our system, the first requirement is satisfied by electing a single primary server for any file or directory to be modified, as described in Section 4.1.2. Here we discuss our design strategies regarding the second requirement.

For scientific applications, losing the results of a computation can be expensive or cause inconsistent results. E.g., scientific applications often keep track of the progress of a computation through log files. Suppose a failure occurs just after an application completes some computation and records that in a log file. In a system that does not provide strong data durability guarantees, the results may be lost after the recovery of failure even though the log file indicates that the computation has completed. As a result, the user and/or application cannot tell where the computation should be resumed.

As discussed in the previous chapters, a main purpose of my thesis is to develop a replicated file system that allows users and applications to access globally distributed data simply and reliably, which argues for the "principle of least surprise". Based on this consideration, it is essential that a replication protocol always guarantees the durability of data written by a client that is acknowledged by its connected server.

As mentioned in Section 4.1.1, a primary server is responsible for distributing updates to other replication servers during file or directory modification. To prevent data loss should the primary server fail, we require that a primary server not process a client update request until it receives update acknowledgments from a majority of the replication servers. With this requirement, the "current" version of data is well-defined and, as long as a majority of the replication servers are available, a fresh copy can always be recovered. Once all active servers synchronize with the most current copy, we are guaranteed that the data after recovery reflects all acknowledged client updates, and a client needs to reissue only its last pending request.

In my implementation, the primary server distributes updates to other replication servers in parallel. Updates must be delivered in order, which can be achieved by including a serial number in each message or through an ordered transport protocol. Initially, a replication server is unsure if a received update is valid — e.g., the primary server might send out

an update request and then immediately crash — so it does not apply the update at once. Rather, the update is cached until a commit request from the primary server arrives. To reduce network traffic, the primary server can piggy backs the commit request to the next message that it distributes to other replication servers.

In addition to the data payload, each update message from the primary server to other replication servers also includes metadata related to the update, such as the modification time. Each replication server modifies its copy of file metadata appropriately when updating file data. This guarantees that the metadata of the file is consistent among replicas, which as we show in Section 4.1.4, makes it easy to determine the most recent file copies during failure recoveries. [1]

The described update distribution strategy provides applications high failure resilience, but also brings several performance concerns.

First, distributing updates to other replication servers every time the primary server receives a write request might lead to redundant network traffic should some or all of a file be written several times. The concern is migrated by observing that an NFS client usually delays asynchronous writes for some seconds before sending them to the server. This increases the likelihood that these updates will be long-lived [15].

Second, a directory update involves two message delays, one for primary server election and the other for update distribution. To reduce this cost, we allow a primary server to send the primary server election request and the update request together in one compound RPC [92]. A replication server receiving this compound RPC caches the update, begins to block local update, and acknowledges the primary server election request. After receiving replies from a majority of the replication servers, the primary server acknowledges the client request. Simultaneously, the primary server could send a commitment message, notifying other replication servers to apply the update. As discussed previously, to reduce network traffic, the primary server piggy backs this notification to the next message that it distributes to other replication servers.

[1] Data update and metadata update do not need to be atomic. I.e., if a replication server crashes after updating its data copy but before updating its metadata copy, the data stored on it is considered stale when it returns from the failure. As we discuss in Section 4.1.4, the server then synchronizes itself with the most recent copy, which brings its metadata up to date.

Third, by making client-to-server writes synchronous with updates to other replication servers, we are increasing the latency of synchronous writes and metadata updates. Although our study assumes read-dominant applications, we are aware that many scientific workloads are characterized by large amounts of synchronous writes, bursty metadata updates, and widely separated replication servers. Evaluation data in Section 4.3 shows that the described replication protocol does indeed introduce considerable performance penalty for such workloads. Solving this problem requires a modification to the proposed update distribution strategy that better balances the tradeoff between performance and failure resilience; this issue is investigated thoroughly in Chapters 5 and 6.

4.1.4 Failure Recovery

The principle challenge in data replication is maintaining consistency in the face of failure. Different forms of failure may occur: client failure, replication server failure, network partition, or any combination of these. In this subsection, we describe the handling of each case. The correctness proof of the protocol is presented in Appendix. Our failure model is *fail stop* [107], i.e., no Byzantine failures [35]. Security of the protocol follows from the use of secure RPC channels, mandatory in NFSv4, for server-to-server communication.

Client Crash

Following the specification of NFSv4, a file opened for writing is associated with a lease on the primary server, subject to renewal by the client. If the client fails, the server receives no further renewal requests, so the lease expires. Once the primary server decides that the client has failed, it closes any files left open by the failed client on its behalf. If the client was the only writer for a file, the primary server relinquishes its role for the file. Unsurprisingly, the file content reflects all writes acknowledged by the primary server prior to the failure.

Replication Server Failure

To guarantee consistency upon server failure, we use view change algorithms analyzed by El-Abbadi et al. [8] to maintain an *active view* among replication servers and allow updates only among the servers contained in the active view.

An active view is required to include a majority of the replication servers to guarantee its uniqueness. To avoid unnecessary network traffic, we do not use periodic heartbeat to maintain active view. Rather, in our system, active view is refreshed during updates.

During file or directory modification, a primary server removes from its active view any replication server that fails to respond to its election request or update requests within a specified time bound. The primary server can relinquish its role only after distributing the new active view to all active servers. Each replication server records the active view in stable storage. A server not in the active view may have stale data, so the working servers must deny any requests coming from a server not in the active view.

A failed replication server has to synchronize itself with a live replication server before being re-added into the active view. Basically, when an active server detects the return of a failed server, either upon receiving an election or update request from the returning server or under the control of an external administration service, it synchronizes the returning server to the up-to-date state and then initiates a view change procedure to add the returning server to the active view.

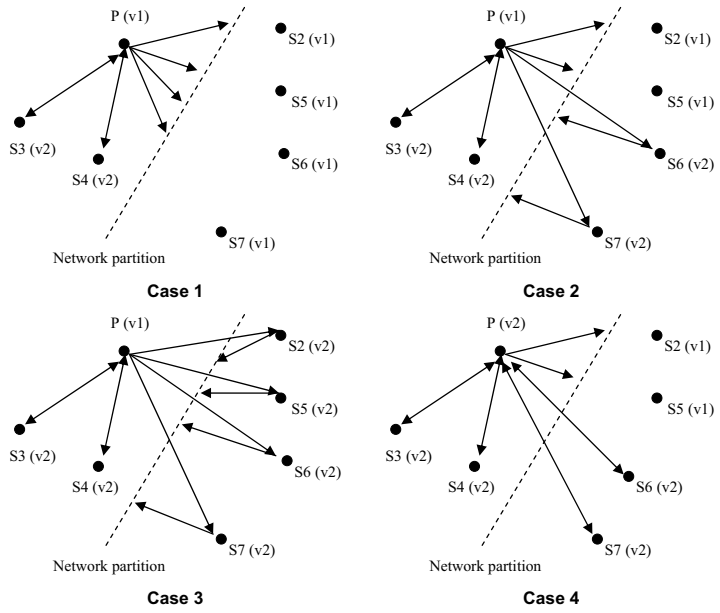


Figure 4.2: Possible situations when the primary server is in a minority partition.

This figure shows different situations that may occur after the primary server P is separated in a minority partition during processing a client write request. S2 - S7 represent other replication servers. The value in parentheses denotes the data version of the file copy at the corresponding replication server.

If a primary server's active view shrinks to less than a majority while processing a client write request, different situations may occur, as illustrated in Figure 4.2. Because the primary server cannot determine the data version in the majority partition, it fails the client write request. Some replication servers may have applied the previous update; consequently, the file content is inconsistent among replication servers. However, the primary server has not yet released its role for the file, and as we will describe later, our failure recovery procedure guarantees that the system converges in the majority partition before any further update can proceed. Thus, conflicting writes are prevented.

Primary Server Failure Recovery

A primary server failure (crash failure or network partition that separates the primary server in a minority partition) can be detected by a replication server whose forwarded request times out. Below, we refer to the replication server that has noticed the primary server failure as the *replacement server*, which then works with other replication servers to recover the failure as follows.

The replacement server first initiates a view change procedure that collects majority consensus among replication servers. It then eliminates the failed primary server from the active view. After that, the replacement server executes a synchronization process, during which it queries the other active replication servers for the modification times of the objects that are controlled by the failed server, i.e., those for which the failed server is the primary server at the moment it fails. The replacement server then synchronizes all of the active replication servers with the most up-to-date copies that it found, which it may first need to obtain itself. After synchronization, all of the objects that were controlled by the failed server are released.

If a replication server fails after sending primary server election requests to a minority of the replication servers, the failure can be detected by a subsequently elected primary server (this is possible since a majority of acknowledgments can still be gathered from the other active servers). As described above, that primary server eliminates the failed server from the active view and distributes the new view to the other replication servers. The servers that have acknowledged the failed server switch to the new primary server after employing the new active view. The consistency of the data is unaffected: the failed server

had not received acknowledgments from a majority of the replication servers, so it cannot process any client updates.

A client that connects to the failed server can switch to an active one through the client-side recovery mechanism for server migration, part of the NFSv4 standard. In brief, this entails cleaning up the state associated with the old server, reopening the files on the new server, and reissuing the failed request. If the failed I/O is a write, it is indeterminate whether the state of the recovered file reflects that write operation. However, this does not compromise data consistency as repeating a write request is safe in NFSv4.

No Majority Partition

As described above, the failure recovery procedure starts when the primary server fails or is isolated in a minority partition. However, if there are multiple partitions and no partition includes a majority of the replication servers, a new active view can not be formed until the partitions heal sufficiently to allow a quorum to assemble. In this case, read requests can be satisfied, but write requests must be refused.

To be safe, an external administrator should enforce a grace period after recovering from the failure. During the grace period, the administrator instructs each pair of the replication servers to alternately execute a synchronization program, such as `rsync`, with the option to update only files or directories whose modification time is older than the corresponding copy on the peer node.

4.2 Close-to-open Semantics and Synchronized Access

The protocol discussed so far guarantees ordered writes among replication servers. However, applications sometimes require stronger consistency guarantees, such as close-to-open semantics or synchronized access. This section presents schemes to provide these consistency semantics.

4.2.1 Support for Close-to-open Semantics

The conventional NFS consistency model, the so-called “close-to-open” semantics [109], guarantees that an application opening a file sees the data written by the last application

that writes and closes the file. This strategy has proved to provide sufficient consistency for most applications and users [91].

Close-to-open semantics is also important in Grid computing. Consider a simple edit-and-run example in which a program is edited on one client, whereupon a number of clients are instructed to execute it. Because the execution instruction might be issued immediately after the program editing, the access on the file must be coordinated. To guarantee correct synchronization behavior in such scenarios, we extend the described replication protocol as follows.

To guarantee close-to-open semantics, a primary server instructs other replication servers to forward client read requests as well as write requests for the written file to itself. Furthermore, the primary server must ensure that all replication servers have acknowledged its election when a written file is closed, not just a majority, so that subsequent reads on any server reflect the contents of the file when it was closed. If any replication server fails to acknowledge the primary server election request, the primary server rejects the client close request.

The first requirement sufficiently guarantees that any read after close is forwarded to the primary server before update distribution completes. Alternatively, we can let a primary server notify other replication servers to forward only client write requests during file modification, and distribute another request instructing other replication servers to forward both client read and write requests to itself upon file close. In this way, client reads received by another replication server are not forwarded to the primary server during file modification. However, this approach adds one more replication control message. Furthermore, it increases the response time for client close request since a primary server needs to wait for the acknowledgments from all of the other replication servers after distributing the second notification request. We consider concurrent writes to be infrequent while file close to be more common, so we elect to use the current solution.

Without failure, the second requirement is satisfied automatically if the client access to the written file lasts longer than the duration of the primary server election. However, an application that writes many small files can suffer non-negligible delays. We notice that these files are often temporary files, i.e., files that were just created (and are soon to be deleted), so we allow a new file to inherent the primary server that controls its parent directory for file creation. Since the primary server does not need to propose a new election

for writing a newly created file, close-to-open semantics is often automatically guaranteed without additional cost.

If one or more replication servers fail, the primary server rejects the client close request for the written file to prevent clients in a minority partition from accessing stale data after file close.

With close-to-open semantics, a replication server does not forward client read requests for a directory that is currently under modification on the primary server. This strategy is based on observing that the interval from the time that a client receives a directory update acknowledgment and the time that the other replication servers implement the update is small. [2] This model complies with the NFSv4 consistency semantics: in NFSv4, a client caches attributes and directory contents for some specified seconds before requesting fresh information from its server.

4.2.2 Support for Synchronized Access

In a departure from previous versions of NFS, support for byte range locking is part of the NFSv4 protocol. With this support, applications can synchronize their access to shared resources through file locking and sync calls. E.g., an application can lock a range of a file, make some modification, flush the written data to the server through synchronous writes, and then release the lock. With a single NFSv4 server, the application is guaranteed that after the synchronous writes complete, other applications that subsequently open the file see the written data. An application can further ensure that subsequent reads on any client reflect the written data by using direct read access (i.e., read requests that are sent directly to the server instead of being served from client cache).

To provide the same consistency semantics, we need to ensure that when a synchronous write completes, the data written is committed by all of the replication servers. This can be enforced through an approach similar to that for supporting close-to-open semantics. I.e., when a primary server receives a synchronous write request from a client, it notifies the

[2]The primary server replies to a client update request after it receives acknowledgments from half of the other replication servers. In the current prototype, the primary server distributes an update commitment request to other replication servers after it receives the acknowledgments from all of the other replication servers for the distributed update request. Therefore, the difference between the two times is usually bounded by the longest RTT between the primary server and the other replication servers.

other replication servers to forward client read and write requests to itself. Furthermore, it must ensure that all of the other replication servers have acknowledged its role before replying to the client synchronous write request. If this requirement cannot be satisfied due to a replication server failure, the primary server rejects the client synchronous write request.

After the primary server has received acknowledgments from all of the other replication servers for its election request, update distribution for the subsequent (synchronous or asynchronous) client writes works the same way as that in the sequential consistency model. The primary server acknowledges a client write request after it gets acknowledgments from a majority of the replication servers. If there is a failure detected during update distribution, the primary server can still process client read and write requests as long as it is in the majority partition. If the primary server is separated in a minority partition, it fails any client write request. In the majority partition, the failure recovery mechanism described in Section 4.1.4 can be used to recover the fresh copy of data. After that, read requests can be served in the majority partition.

4.2.3 Discussion

As we have described, a primary server rejects a client close request or synchronous write request if it cannot collect acknowledgments for its election request from all of the other replication servers. In other words, in the case of failure, our system guarantees close-to-open semantics at the cost of sacrificing system availability for close operations, and guarantees synchronized access at the cost of sacrificing system availability for synchronous write requests. Several methods can be used to bypass this restriction if it is critical to guarantee system availability as well as strong consistency guarantees.

For example, we can use periodic heartbeat messages to detect partition failures and require a replication server to reject any client requests if it fails to receive heartbeat messages from a majority of the replication servers. With this requirement, even in case of failure, the system can continue to process client writes in a majority partition after waiting for a heartbeat period. Since any subsequent client access is disallowed in a minority partition, synchronized access is guaranteed.

Sequential Consistency	Close-to-open Semantics	Synchronized Access
A primary server ensures that a majority of the replication servers have acknowledged its role before replying to a client write request	A primary server ensures that a majority of the replication servers have acknowledged its role before replying to a client write request *and* all of the other replication servers have acknowledged its role before replying to a client close request	A primary server ensures that all of the other replication servers have acknowledged its role before replying to a client write request
Other replication servers are instructed to forward client write requests to the primary server	Other replication servers are instructed to forward client read and write requests to the primary server	Other replication servers are instructed to forward client read and write requests to the primary server
During failure, reads are allowed everywhere; writes are allowed in the majority partition.	During failure, reads are allowed everywhere; a client cannot close a written file	During failure, reads are allowed in the majority partition; a client cannot synchronously write a file

Table 4.1: Comparison of the schemes to guarantee sequential consistency, close-to-open semantics, and synchronized access.

Alternatively, we can rely on an external administration service, such as a Grid job manager. When failure occurs, active replication servers exclude the failed server(s) from the active view and report the failure to the administration service. The system can continue to operate after the failure is recorded. According to the described replication protocol, a server excluded from the active view cannot update any working server, which prevents the system from entering any inconsistent state. However, if the failure is caused by a network partition, close-to-open semantics or synchronized access are not guaranteed on the "failed" server(s), i.e., clients may have read stale data without awareness. In this case, the computation results of an application that accesses a failed server become dubious since they might be generated with the stale input data. To be safe, after the recovery of the failure, the administration service should roll back the application to the state before the failure happens.

In Table 4.1, we summarize the schemes for supporting sequential consistency, close-to-open semantics, and synchronized access.

4.3 Evaluation

In this section, we evaluate performance under different network conditions. In Section 4.3.1, we use a modified Andrew benchmark with replication servers running in a local area network and in the (simulated) high-latency, wide area networks. Section 4.3.2 evaluates

performance when synchronized access is required. In Section 4.3.3, we dig deeper into performance by running real scientific applications over replication servers in (simulated) global area networks.

We measured all of the experiments presented in this section with a prototype implemented in Linux 2.6.12 kernel. Servers and clients all run on dual 2.8GHz Intel Pentium4 processors with 1024 KB L2 cache, 1 GB memory, and dual Intel 82547GI Gigabit Ethernet cards on board. We use TCP as the transport protocol. The number of bytes that NFS uses for reading (rsize) and writing (wsize) files are set to 32 KB. In our experiments, we use Netem [55] to induce simulated wide area network latencies. All numbers presented are mean values from three trials of each experiment; standard deviations (not shown) are negligible.

4.3.1 Andrew Benchmark Evaluation

This subsection presents the experimental results of running a modified Andrew benchmark. The Andrew benchmark [57] is a widely used file system benchmark that models a mix of file operations. It measures five stages in the generation of a software tree. Stage (*I*) creates the directory tree, (*II*) copies source code into the tree, (*III*) scans all the files in the tree, (*IV*) reads all of the files, and finally (*V*) compiles the source code into a number of libraries. The modified Andrew benchmark used in our experiments differs from the original Andrew benchmark in two aspects. First, in the last stage, it compiles different source code from that included in the Andrew benchmark package. Second, the Andrew benchmark writes a log file in the generated directory. If writes are slow compared to reads, the cost of updating the log file dominates the overall cost of a stage that mostly reads, hindering analysis. Therefore, we use local disk to hold the log file.

Our first experiment looks at replication in a LAN environment, such as a cluster. Figure 4.3 depicts the performance of the modified Andrew benchmark as the number of replicas increases. The measured RTT between any two machines is around 200 μsec.

Figure 4.3 shows that in a LAN, the penalty for replication is small. Replication induces no performance overhead in Stages (*III*) and (*IV*), as these two phases consist of read operations only. Stage (*V*) is compute-intensive, so the performance difference between single server and replicated servers is negligible. Most of the performance overhead for repli-

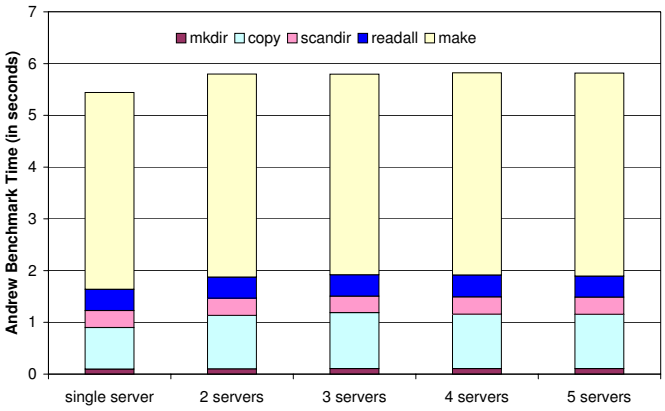


Figure 4.3: the modified Andrew benchmark in LAN replication.

cation comes from Stages (*I*) and (*II*), which consist of file and directory modifications. However, with a fast network, the aggregate penalty is still only a few percent. Furthermore, because a primary server distributes updates to other replication servers in parallel, performance is not adversely affected as the number of replication servers increases.

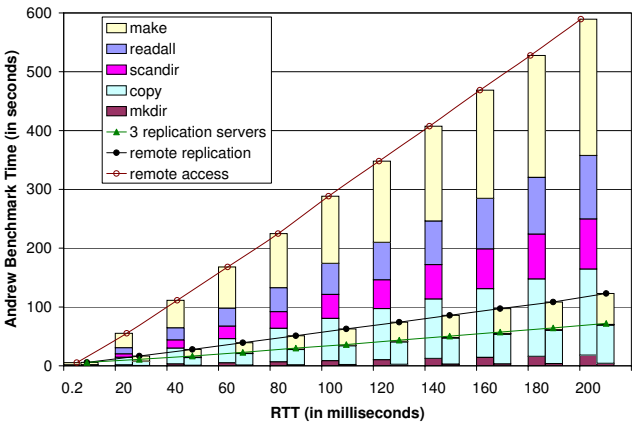


Figure 4.4: The modified Andrew benchmark in WAN replication.

The next experiment, depicted in Figure 4.4, compares the performance of replicating to a distant server vs. the performance of accessing a distant server directly. We ran the modified Andrew benchmark with an increasingly distant file server, the upper line in Figure 4.4, and again with a local replication server and an increasingly distant replication server, the middle of the three lines in Figure 4.4. The RTT marked on the *X*-axis shows the round-trip time between the primary server and the remote replication server for the

replication experiments, and between the client and the remote server for the single remote server access experiments. In Figure 4.4, the smallest RTT measured is 200 μsec., the network latency of our test bed LAN. For the other measurements, we use Netem [55] to simulate network delays. Each experiment first warmed the client's cache with two runs.

Figure 4.4 shows that replication outperforms remote access in all of the five stages. In Stages (*III*) and (*IV*), the read-only stages, replication is as fast as local access, since no messages need to be sent to the other replication server in these stages. Replication also dominates remote access in the other three stages. A close look at the network traffic in the measured experiments shows that with replication, fewer messages are sent to the remote server, accounting for its advantage. Roughly, the running time in each stage can be estimated as

$$T = Tbasic + RTT \times NumRPC \tag{4.1}$$

where *Tbasic* denotes the computation time at the client and the request processing time at the connected server. For replication, *RTT* represents the round-trip time between the primary server and the replication server, and *NumRPC* represents the number of RPC messages sent from the primary server to the replication server in the corresponding stage. For remote access, *RTT* represents the round-trip time between the client and the remote server, and *NumRPC* is the total number of RPC requests sent from the client to the server.

The *Tbasic* cost is about the same for replication and remote access, so any difference in performance must be accounted for by the second part of the formula. For example, Stage (*I*) creates 20 directories at the cost of 85 RPC requests sent from the client to the connected server. These RPC requests consist of 20 `create`, 13 `access`, 17 `getattr`, and 35 `lookup`. With a single remote server, the reported `access`, `getattr`, and `lookup` requests are unavoidable even with a warm cache because they are requesting information on newly created directories. However, with replication, `access`, `getattr`, and `lookup` requests are served locally at the primary server, eliminating their cost altogether at the scale of this experiment. Furthermore, with replication, although each `create` costs two RPC messages, the primary server replies to the client after receiving the response to only the first of these two. Consequently, with replication, the number of latency-inducing remote RPC messages decreases from 85 to 20. Table 4.2 summarizes RPC counts for all of the five stages.

System Model	Mkdir	Copy	Scandir	Readall	Make	Total
Replication	20	228	0	0	71	309
Single remote server access	85	735	154	510	589	2073

Table 4.2: Number of remote RPC messages sent during the five stages of the modified Andrew benchmark with replication and single remote server access.

On the other hand, Figure 4.4 also shows that compared with single local server access, remote replication adds considerable overhead on the performance of the application that consists of many metadata updates (see Stages (*I*), (*II*), and (*V*)). In Chapter 5, we describe the enhancements to the replication protocol that reduce this performance penalty.

An important feature of the presented protocol is that a primary server can reply to a client request as soon as it gets acknowledgments from half of the other replication servers. Then given a set of replication servers, the performance of the protocol is dictated by the largest RTT between the primary server and half of the nearest replication servers, which we call the *majority diameter*.

To illustrate how this feature can be used to advantage, we added a third replication server halfway (in terms of RTT) between the other two and re-ran the modified Andrew benchmark. The result is the lowest of the three lines in Figure 4.4. Placing a third replication server midway between the local and the remote replication servers cuts the majority diameter in half. For the modified Andrew benchmark, this cuts the overall running time, which is dominated by the cost of remote RPC, nearly in half.

The results imply that if most writes to a replicated file come from one site, the performance overhead for remote replication can be made scant by putting a majority of the replication servers near that site. Furthermore, if a site is using local replication, the penalty for adding a distant replication server, say, for very off-site backup, is negligible. However, we note that putting a majority of replication servers in a single site increases the probability that these servers fail at the same time. This introduces a tradeoff between performance and failure resilience. We explore this thoroughly in Chapter 6.

Generally, our system is designed to be used with a small number of replication servers, say, fewer than ten. Under this assumption, we do not expect performance to suffer when additional replication servers are added as long as the majority diameter is fixed. To test this conjecture, we measured the running time of the modified Andrew benchmark with a

Replication servers	RTT to the primary server
P (the primary server)	-
R2, R3	20 msec
R4	40 msec
R5, R6, R7	60 msec

Table 4.3: Servers used in Figure 4.5 experiments.

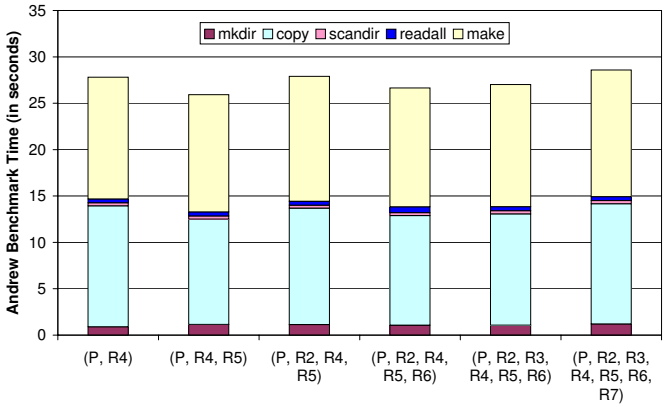


Figure 4.5: The modified Andrew benchmark with different replication server sets.

fixed majority diameter, varying the number of replication servers. Figure 4.5 shows the measured results. The servers used in this experiment are described in Table 4.3.

As the data shows, with the majority diameter fixed (at 40 msec), increasing the number of replication servers has negligible effect on system performance, which is key to good scaling. As a *gedanken experiment*, we might imagine the practical limits to scalability as the number of replication servers grows. A primary server takes on an output bandwidth obligation that multiplies its input bandwidth by the number of replication servers. For the near term, unicast communication and the cost of bandwidth seem to be a barrier to massive replication.

To summarize, the evaluation data presented in this subsection illustrates two main points. First, network RTT is the dominant factor in the performance of WAN data access. By locating a replication server close to the client , NFSv4.r can reduce RTT-induced latency. Second, NFSv4.r scales well in this workload. Application performance is unaffected by adding additional replication servers while keeping the majority diameter fixed.

4.3.2 Evaluation of Synchronized Access

This subsection evaluates the performance of NFSv4.r when synchronized access guarantee is required. In particular, we take a simple edit-and-run example in which a user edits a file on one client and starts running it on another client. For evaluation purpose, we emulate this example by overwriting a file on one client (writer) and then immediately reading it on another client (reader). To guarantee read-after-write synchronization, the first client issues a fsync call after overwriting the file.

Below, we first describe the experiment setup. After that, we report the execution time measured on the writer and the reader, respectively.

We measured the time to overwrite a file and to synchronously read a modified file in five distribution models, as illustrated in Figure 4.6. In the OneServer-120ms and OneServer-60ms distributions, the client (reader or writer) accesses a single remote server with the RTT between the server and the client set to 120ms and 60ms, respectively. In the 2Rep-120ms and 2Rep-60ms distributions, two replication servers are used with the intermediate RTT between them set to 120ms and 60ms, respectively. In both distributions, the client (reader or writer) connects to a replication server in its LAN. In the 3Rep distribution, we construct an unbalanced configuration by placing three replication servers with the RTTs between each pair set to 60ms, 60ms, and 120ms, respectively. The writer (client1) and the reader (client2) are remote from each other, and each connects to a server in its LAN.

Figure 4.7 shows the total time measured on the writer to overwrite and then fsync a file as the size of the file increases. As the data indicates, in general, replication outperforms single remote server access with the same distribution RTT. The performance benefit comes from the reduced number of the remote messages sent to open the file and to check the file's access mode and attributes, similar to the observation discussed in Section 4.3.1. Furthermore, write time jumps at different file sizes with replication and single remote server access. The jump is due to TCP congestion control that introduces an extra roundtrip delay when transferring written data. With replication, fewer messages are sent to the remote node, so performance decreases with larger file sizes.

As mentioned previously, the primary server needs to guarantee that all of the other replication servers have acknowledged its election request before replying to a client synchronous write request. After that, the primary server can reply to a client write request

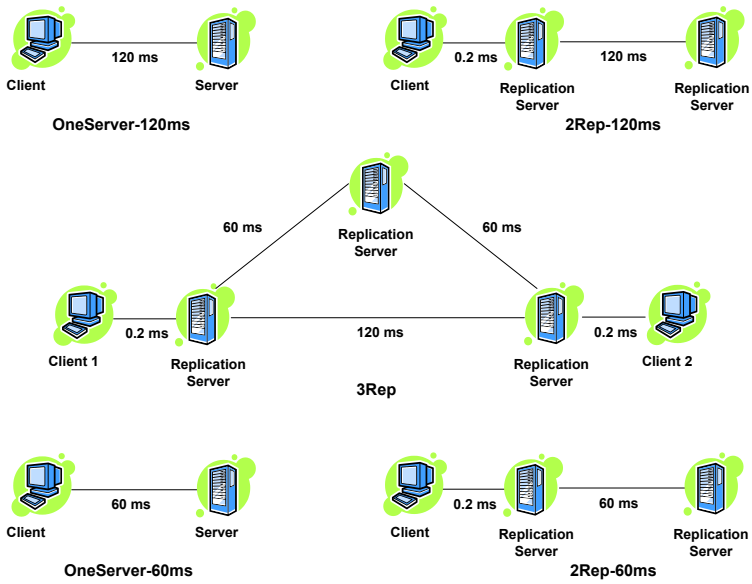


Figure 4.6: Experiment setup for synchronized access evaluation.

as soon as it gets acknowledgments from half of the other replication servers for the distributed update request. So we expect that the synchronous write performance in NFSv4.r is dictated by the majority diameter rather than the longest RTT between the primary server and the other replication servers. The measured experiment results validate our prediction. As observed, the execution time measured in 3Rep distribution is close to that measured in 2Rep-60ms distribution. A slightly longer delay is observed when the file size is small, corresponding to the waiting time when the primary server processes the first synchronous write request. The delay disappears as the file size becomes large. In those cases, when the first synchronous write request triggered by fsync reaches the primary server, it has already received the acknowledgments from the other two replication servers.

We now move to evaluating read performance during synchronized access. Figure 4.8 shows the synchronous read time for a file that was just modified by the writer. To evaluate performance when read forwarding occurs, we pay special attentions to the 3Rep distribution. In NFSv4.r, a primary server immediately responds to a client close request for a written file but delays releasing its role until all of the file updates are acknowledged by every active replication server. With an unbalanced replication server distribution, a slow or remote server can fall behind from file modification with a burst of writes. In such cases,

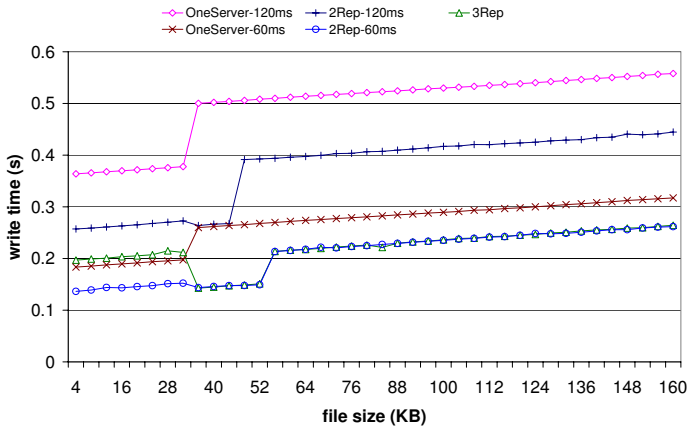


Figure 4.7: Synchronized writes. The figure shows the measured time to overwrite & fsync a file as the file size increases. The experiment setup is shown in Figure 4.6.

when the reader starts accessing the file, the replication server it connects to forwards the client read requests to the primary server.

In the experiments, we first start reading the file on the reader by immediately sending an `ssh` command from the writer after the file modification finishes, which we consider a typical edit-and-run scenario. The network delay between the writer and the reader is set to 120ms, as the experiment setup depicts. However, with this method, no forwarded reads are observed because the latency of sending an `ssh` command dominates the delay of update distribution. As observed, the read performance in such cases (depicted as 3Rep-ssh-start) is similar to that when both the writer and the reader connect to a single local server (depicted as OneServer-local).

For evaluation purpose, we artificially start reading the file right after file modification. In this situation, we observe that the first read request from the reader is forwarded to the primary server. After that, the update distribution completes and the primary server releases its role, so subsequent read requests are processed by the nearby server. As Figure 4.8 shows, the performance with no-delay start (depicted as 3Rep-nodelay-start) stays nearly the same as the file size increases, compared with the climbing delay observed when reading the file from a single remote server (depicted as OneServer-120ms). The overhead caused by the forwarded read corresponds to the performance difference observed between 3Rep-nodelay-start and reading a single local server, which is about the same as the RTT between the replication server and the primary server.

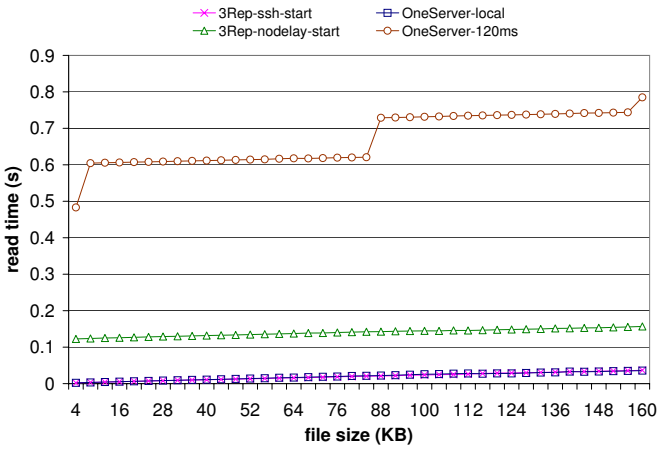


Figure 4.8: Synchronized reads. The figure shows the synchronous read time for a file that was just modified on another client. The experiment setup is shown in Figure 4.6. In the experiment, 3Rep-ssh-start and 3Rep-nodelay-start both use the 3Rep configuration as depicted in Figure 4.6. In 3Rep-ssh-start, we start reading the file by sending an `ssh` command from the writer after writes complete. In 3Rep-nodelay-start, we artificially start reading the file immediately after file modification, without the ssh latency between the two clients. For comparison, we also present the time measured with the writer and the reader both connected to a single local server and a single remote server, as represented by OneServer-local and OneServer-120ms, respectively. In OneServer-120ms, the RTT between the server and the client is set to 120ms.

In summary, the evaluation results presented in this subsection show that users usually observe no additional cost during synchronized access in NFSv4.r. A slight performance penalty is charged when read forwarding occurs, but even then, the performance of NFSv4.r still outperforms single remote server access.

4.3.3 ATLAS Evaluation

In this subsection, we explore the performance of NFSv4.r for scientific application workloads. Our evaluation uses the ATLAS detector simulation software that consists of a set of cluster-based, data-intensive, distributed applications poised for deployment on the Grid. After a brief description of the ATLAS software, we describe the experiments with these applications.

ATLAS is a particle physics project that searches for new discoveries in high-energy proton collisions [1]. Protons will be accelerated in the Large Hadron Collider (LHC) accelerator, currently under construction at the European Laboratory for Particle Physics

(CERN) near Geneva [5]. The accelerator is expected to begin operation in 2007. After that, on the order of a petabyte of raw data will be produced each year, i.e., a continuous stream of over 300 KBps, and distributed to a multi-tiered collection of decentralized sites for analysis.

ATLAS is the largest collaborative effort ever attempted in the physical sciences. 1,800 physicists from more than 150 universities and laboratories in 34 countries participate in this experiment. With the massive amount of data to be processed and the widely distributed collaborators, ATLAS stands to benefit from a scalable and reliable data access and management scheme, which is also a target of our design.

At this writing, ATLAS is performing large-scale simulation of physics events that will occur within an ATLAS detector. These simulation efforts support the detector design and the development of real-time event filtering algorithms that are critical for controlling the flood of data when the LHC accelerator is running.

The ATLAS event simulation model consists of four stages. The first stage, `Event Generation`, uses a seed to produce pseudo-random events drawn from a statistical distribution deduced from other experiments. The second stage, `Simulation`, reads the generated events and simulates the passage of particles through the detectors. The third stage, `Digitization`, converts simulated hit events into digital outputs (called digits). The digits are fed to the fourth stage, `Reconstruction`, which performs pattern recognition and track reconstruction, converting raw digital data into meaningful physics. The four stages have different computational requirements and generate different amounts of output data. For example, when processing 1000 events on a dual 2.4 GHz Pentium4 processors with 1 GB memory, `Event Generation` takes two minutes and generates 20 MB of output data; `Simulation` takes 33 hours and generates 800 MB of output data; `Digitization` takes 8 hours and generates 1.6 GB of output data; `Reconstruction` takes 8 hours, generating 8 to 20 MB of output data.

In this dissertation, we skip over the discussion of the first two stages: the time spent on `Event Generation` is considerably less than the other three, while `Simulation` is utterly CPU bound, so their performance is not likely to be sensitive to our work on the I/O side. Our analysis focuses on `Digitization` and `Reconstruction`. We want to explore the following questions:

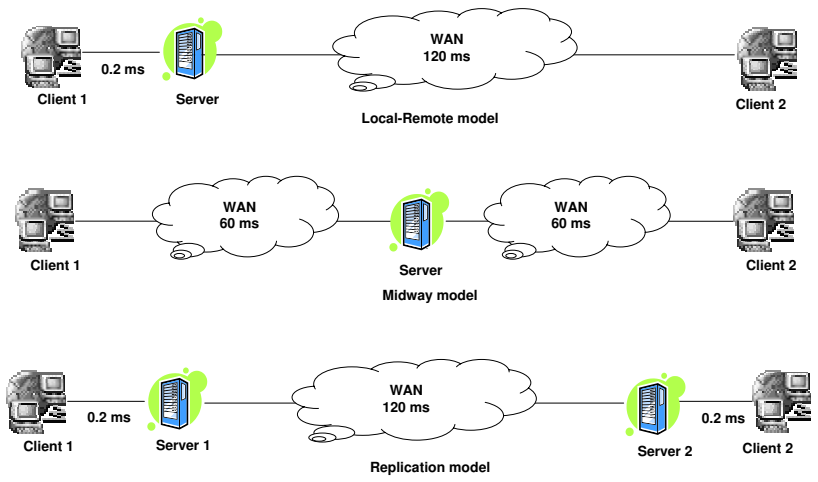


Figure 4.9: ATLAS experiment setup.

- Will these applications that are normally CPU bound become I/O bound when they are globally distributed?
- If I/O performance becomes a critical factor, can file system replication help to improve the performance of these applications?
- For the two applications, one of which generates a modest amount of output while the other produces a large volume of data, what is the performance cost of replicating their outputs remotely?

To answer these questions, we measured the time of running ATLAS `Digitization` and `Reconstruction` with three distribution models: Local-Remote, Midway, and Replication. In each distribution model, we ran the ATLAS applications on a pair of NFSv4 clients, with the RTT between them set to 120 msec, the measured `ping` time from our experimental test bed in Ann Arbor to CERN. In the Local-Remote distribution, the two clients access data from a single NFSv4 server that locates on one client's LAN. In the Midway distribution, we place a single NFSv4 server half way between the two clients. In the Replication distribution, we place a replication server on each client's LAN. Figure 4.9 illustrates the experiment setup.

In the experiments presented in this subsection, each client processes 50 events. In our experiments, we used the `Digitization` and `Reconstruction` software from the ATLAS 10.0.4 installation package. For `Reconstruction`, we applied all of the algo-

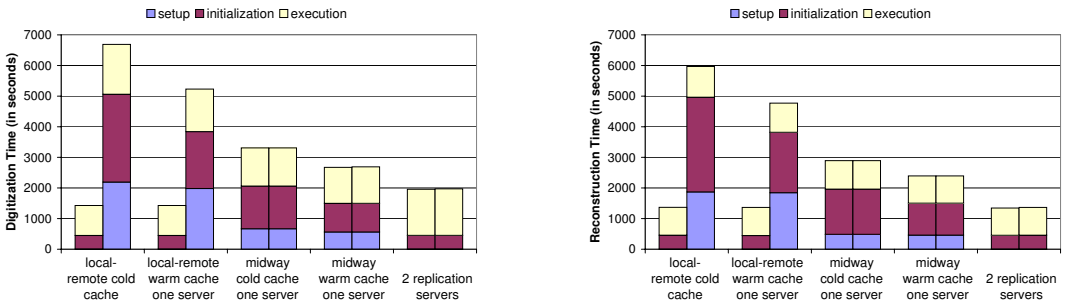


Figure 4.10: Left: ATLAS Digitization. Right: ATLAS Reconstruction. In each category, the first column shows the running time measured on the left side client in Figure 4.9, and the second column shows the running time measured on the other client.

rithms included in the default installation. For the first two distribution models, we present the performance measured with both cold and warm client caches. With Replication, cache temperature has little influence on the running time of either application. Even with a cold client cache, the un-cached data is retrieved from a nearby replication server. The cost of this is relatively small compared with the overall running time of the applications.

Figure 4.10 shows the measured running time of ATLAS `Digitization` and `Reconstruction`. In both diagrams, the first column of each category shows the running time measured on the left side client in Figure 4.9, and the second column shows the running time measured on the other client.

Figure 4.10 shows that Replication outperforms Local-Remote and Midway for both applications. To see just where the performance improvements come from, we further divided the applications into three phases and measured the time spent on each of them.

In the `Setup` phase, the applications prepare their run-time environments. In the `Initialization` phase, the applications read header files and libraries, and link them into executables. We measured the initialization time by setting the number of simulation events to zero. The `Execution` phase processes events. We calculate the execution time by subtracting the time spent on the first two phases from the total running time.

Examining the detailed experimental results presented in Figure 4.10, we see that for both applications, most performance benefit of replication comes from `Setup` and `Initialization`. We find that even with a warm client cache, the performance of these two phases still suffers dramatically as the RTT between the server and the client increases. Taking a close look at the network traffic with a warm client cache, we were surprised to

see that a large number of file open requests are sent during these two phases, even though NFSv4 has a delegation scheme that allows a client to perform subsequent open requests locally after the first call in the absence of shared writers. Further examination reveals that most of these open requests are met with "`No entry exists`" error, obviating any potential delegation advantage since NFSv4 does not provide a way for a client to cache negative results.

We believe that the applications are issuing these open requests as a way of examining the configuration of the running environment. The cost of doing this on a local file system or with a nearby server is too small to make a substantial difference in the running time, but begins to have an impact as the server is made more and more remote. Replication helps by allowing these open requests to fail on a nearby server, with the performance comparable to accessing a single local server.

In the `Execution` phase, ATLAS `Digitization` produces approximately 7.2 MB of output data, most of which is written to a single output file (per process). The output data is read by `Reconstruction` during the `Execution` phase, whose output size shrinks to 1.4 MB per process, according to our measurements.

Figure 4.10 shows that for ATLAS `Reconstruction`, the performance of the `Execution` phase is similar in all of the three distribution models, but this is not the case for ATLAS `Digitization`. There, `Execution` with a remote server is about 50% more costly than `Execution` with a local server, and `Execution` with replication is comparable to the latter.

ATLAS `Digitization` generates a significant amount of output data, nevertheless we were surprised that the performance penalty for remote replication is so high. To find the reason, we examined the trace data collected during the experiment and found that the high performance cost observed during `Execution` is mainly caused by a large number of fsync system calls: more than 900 fsync calls are used with 50 event digitization, compared with 60 fsync calls observed with 50 event reconstruction.

We discussed this observation with the ATLAS developers. It seems that the overwhelming use of fsync is an implementation issue rather than necessaries. To estimate the performance of ATLAS `Digitization` without the impact of the aggressive use of synchronous writes, we eliminated these fsync calls and re-ran the experiment. Figure 4.11 shows the results. As the evaluation data demonstrates, the performance difference be-

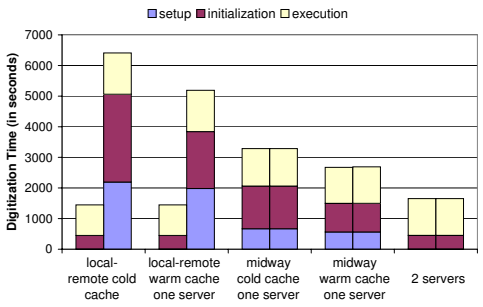


Figure 4.11: ATLAS Digitization without fsync.

tween local server access and remote replication is reduced: `Execution` with replication is about 20% slower than `Execution` with single local server access. The remaining performance difference is caused by large bursty writes that exhaust the client cache, which happens more frequently with remote replication since its response time for individual client write is larger.

A fundamental problem reflected here is that applications often use a large amount of synchronous writes to ensure the durability of written data. The cost of doing this on a local file system is small, but starts to affect performance as the system becomes widely distributed. However, such applications can be common in practice since most programs in use today are developed in local environments. To address this issue, in Chapter 5, we discuss refinements of the protocol to improve performance for synchronous writes.

In part, the performance result presented here reflects a worst case scenario. Typically, the four ATLAS applications are run together as a pipeline. Thus, the described problem can be avoided by keeping intermediate output data in local temporary files and writing only final results over shared storage. Researchers have shown that a diamond-shaped storage profile is a characteristic behavior among scientific applications [113], i.e., small inputs are expanded by early stages into large intermediate results, which are reduced by later stages to small results. This observation implies that it is more efficient to store intermediate results in local storage rather than distributing them remotely. However, when making such decisions, users should also consider the tradeoff between performance and the cost to recompute intermediate results if a failure occurs. As our evaluation shows, for applications that generate a moderate amount of output, the cost of replication is negligible. This suggests that it is appropriate to replicate the output produced by an intermediate stage if its performance is not heavily dependent on write I/O throughput but requires a long execu-

tion time. In Chapter 6, we explore the tradeoff between performance and failure resilience when using different update distribution policies in a replicated file system.

4.4 Summary

Supporting consistent mutable replication in large-scale distributed file systems is traditionally considered too expensive to consider. This chapter demonstrates that, on the contrary, it is possible to provide mutable replication with minor impact on common case performance. The main contribution of this chapter is the development of a mutable replication protocol that provides flexible consistency guarantees, easy failure recovery, and superior read performance, the implementation of the protocol by extending the standard NFSv4, and the evaluation of a prototype that demonstrates its promise. By providing these features, our study exhibits a practical, reliable, efficient, and forward-looking way for accessing and sharing data in wide-area collaborations.

CHAPTER 5

Decreasing the Cost of Replication through Hierarchical Replication Control

As discussed in Chapter 4, although the fine-grained replication control protocol breaks new ground in performance and availability for read-dominant applications, further performance evaluation shows considerable overhead for large synchronous writes, bursty metadata updates, and widely separated replication servers, data access patterns common in Grid computing. The observed performance penalty is mainly due to the cost of guaranteeing durability and the cost of synchronization. Specifically, the durability requirement delays the response to a client update request until a majority of the replication servers have acknowledged the update. This provides a simple recovery mechanism for server failure but synchronous writes suffer when replication servers are far away. The synchronization requirement, which amounts to an election for consensus gathering, also delays applications — especially when they emit a burst of metadata updates — while waiting for distant replication servers to vote.

We assume (and observe) that failures are rare in practice. Furthermore, the computation results by scientific applications can often be reproduced by simply re-executing programs or restarting from a recent checkpoint. This suggests that we may relax the durability requirement to improve performance for synchronous updates. Instead of automatically guaranteeing durability to a client, we may elect to report some failures to the application immediately, e.g., by making the file under modification inaccessible. The application can then decide whether to wait for server recovery or to regenerate the computation results. To reduce the cost of synchronization, we propose a *hierarchical replication control protocol*

that allows a primary server to assert control at granularities coarser than a single file or directory, allowing control over an entire subtree rooted at a directory. This amortizes the cost of synchronization over multiple update requests.

In this chapter, we describe these extensions in detail. In particular, the majority of the chapter is devoted to the design, implementation, and evaluation of hierarchical replication control protocol. More evaluation of the tradeoff between performance and failure resilience follows in the next chapter. Unless specifically noted, the discussion in this chapter assumes close-to-open semantics as the consistency model, but the approaches described here apply to the other consistency models as well.

The remainder of the chapter is organized as follows. Section 5.1 describes modifications to the orignal update distribution strategy that improve the performance of synchronous writes at the cost of sacrificing the accessibility of written data in the case of failure. Section 5.2 presents a hierarchical replication control protocol that allows a primary server to assert control at various granularities to amortize the performance cost of primary server election over more update requests. Section 5.3 evaluates the performance of the refined protocol. In Sections 5.4 and 5.5, we discuss related work and summarize.

5.1 Asynchronous Update Distribution

As specified in the previous chapter, a primary server, which is responsible for distributing updates to other replication servers during file or directory modification, must not process a client update request until it receives update acknowledgments from a majority of the replication servers. Then as long as a majority of the replication servers are available, a fresh copy can always be recovered from them. Having all active servers synchronize with the most current copy, in this way, we guarantee that files after recovery reflect all acknowledged client updates, so that a client needs to reissue only its last pending request, which is the ordinary behavior of an NFS client.

The replication control protocol transparently recovers from a minority of server failures and balances performance and availability well for applications that mostly read. However, performance suffers for scientific applications that consist of many synchronous writes or directory updates when replication servers are far away from each other. Meeting the performance needs of Grid applications requires a different tradeoff.

Failures do occur in distributed computations, but are rare in practice. Furthermore, the results of many scientific applications can be reproduced by simply re-executing programs or re-starting from the last checkpoint. This suggests a way to relax the costly update distribution requirement so that the system provides higher throughput for synchronous updates at the cost of risking the accessibility of data undergoing change in the face of failure.

Adopting this strategy, we allow a primary server to respond immediately to a client write request before distributing the written data to other replication servers. Thus, with a single writer, even when replication servers are widely distributed, the client experiences longer delay only for the first write (whose processing time includes the cost of primary server election), while subsequent writes have the same response time as accessing a local server (assuming that the client and the chosen server are in the same LAN). Of course, should concurrent writes occur, performance takes a back seat to consistency, so some overhead is imposed on the application whose reads and writes are forwarded to the primary server.

A primary server may fail during file or directory modification. With the relaxed update distribution requirement, other active servers cannot recover the most recent copy among themselves. The "principle of least surprise" argues the importance of guaranteed durability of data written by a client and acknowledged by the server, so we make the object being modified inaccessible until the failed primary server recovers or an outside administrator reconfigures the system. However, clients can continue to access data that is outside the control of the failed server, and applications can choose whether to wait for the failed server to recover or to re-produce the computation results.

Since our system does not allow a file or a directory to be modified simultaneously on more than one server even in the case of failure, the only valid data copy for a given file or directory is the most recent copy found among the replication servers. This feature simplifies failure recovery in our system: when an active server detects the return of a failed server, either upon receiving an election or update request from the returning server or under the control of an external administration service, it notifies the returning server to initiate a synchronization procedure. During synchronization, write operations are suspended, and the returning server exchanges the most recent data copies with all active replication

servers. [1] After recovery, all the objects that were controlled by the returning server, i.e., those for which it was the primary server at the moment it failed, are released and the server is added to the active view.

5.2 Hierarchical Replication Control

Even with an efficient consensus protocol, a server can be delayed waiting for acknowledgments from slow or distant replication servers. This can adversely affect performance, e.g., when an application issues a burst of metadata updates to widely distributed objects. Conventional wisdom holds that such workloads are common in Grid computing, and we have observed them ourselves when installing, building, and upgrading Grid application suites. To address this problem, we introduce a hierarchical replication control protocol that amortizes the cost of primary server election over more requests by allowing a primary server to assert control over an entire subtree rooted at a directory. In this section, we detail the design of this tailored protocol.

The remainder of this section proceeds as follows. Section 5.2.1 introduces two control types that a primary server can hold on an object. One is limited to a single file or directory, while the other governs an entire subtree rooted at a directory. Section 5.2.2 discusses revisions to the primary server election needed for hierarchical replication control. Section 5.2.3 then investigates mechanisms to balance the performance and concurrency trade-off related to the two control types.

5.2.1 Shallow vs. Deep Control

We introduce nomenclature for two types of control: shallow and deep. A server exercising *shallow control* on an object (file or directory) **L** is the primary server for **L**. A server exercising *deep control* on a directory **D** is the primary server for **D** and all of the files and directories in **D**, and additionally exercises deep control on all the directories in **D**. In other words, deep control on **D** makes the server primary for everything in the subtree rooted at

[1] This process can be done easily by alternately executing a synchronization program, such as `rsync`, between the returning server and each active replication server, with the option to skip any file or directory whose modification time is newer than the source node.

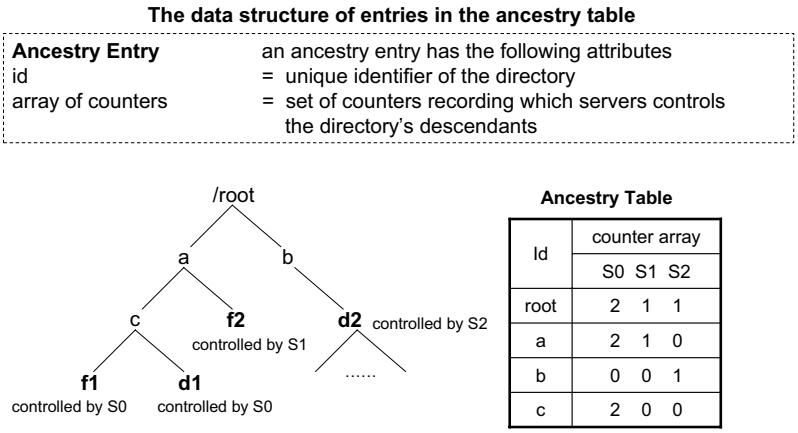
```
Upon receiving a client update request for object L
if L is controlled by self then serve the request
if L is controlled by another server then forward the request
else  // L is uncontrolled
        if L is a file then request shallow control on L
        if L is a directory then
                if a descendant of L is controlled by another server then
                        request shallow control on L
                else
                        request deep control on L

Upon receiving a shallow control request for object L from peer server P
grant the request iff L is not controlled by a server other than P

Upon receiving a deep control request for directory D from peer server P
grant the request iff D is not controlled by a server other than P
and no descendant of D is controlled by a server other than P
```

Figure 5.1: Using and granting deep and shallow controls.

D. In the following discussion, when a replication server **P** is elected as the primary server with shallow control for an object **L**, we say that **P** *has shallow control* on **L**. Similarly, when a replication server **P** is elected as the primary server with deep control on a directory **D**, we say that **P** *has deep control* on **D**. Relinquishing the role of primary server for an object **L** amounts to revoking shallow or deep control on **L**. We say that a replication server **P** *controls* an object **L** if **P** has (shallow or deep) control on **L** or **P** has deep control on an ancestor of **L**.

We introduce deep control to improve performance for a single writer without sacrificing correctness for concurrent updates. Electing a primary server with the granularity of a single file or directory allows high concurrency and fine-grained load balancing, but coarser granularity is suitable for applications whose updates exhibit high temporal locality and are spread across a directory or a file system. A primary server can process any client update in a deeply controlled directory immediately, so it improves performance for applications that issue a burst of metadata updates.

Introducing deep control complicates consensus during primary server election. To guarantee that an object is under the control of a single primary server, we enforce the rules shown in Figure 5.1. We consider single writer cases to be more common than concurrent writes, so a replication server aggressively attempts to acquire deep control. On the other hand, we must prevent an object from being controlled by multiple servers. Therefore, a replication server needs to ensure that an object in a (shallow or deep) control request is



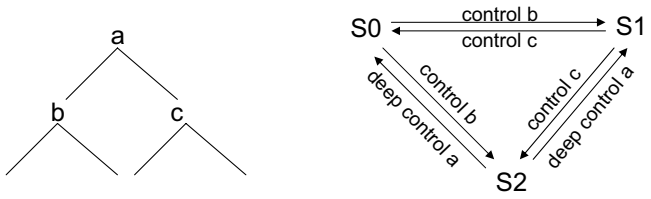
Consider three replication servers: S0, S1, and S2. Currently, S0 is the primary server of file f1 and directory d1, S1 is the primary server of file f2, and S2 is the primary server of directory d2. The right table shows the content of the ancestry table maintained on each replication server.

Figure 5.2: Structure and maintenance of the ancestry table.

not already controlled by another server. Furthermore, it must guarantee that a directory in a deep control request has no descendant under the control of another server.

To validate the first condition, a replication server scans each directory along the path from the referred object to the mount point. If an ancestor of the object has a primary server other than the one who issues the request, the validation fails. Checking the second condition is more complex. Scanning the directory tree during the check is too expensive, so we do some bookkeeping when electing a primary server: each replication server maintains an *ancestry table* for files and directories whose controls are granted to some replication servers. An entry in the ancestry table corresponds to a directory that has one or more descendants whose primary servers are not empty. Figure 5.2 shows entries in the ancestry table and an example that illustrates how the ancestry table is maintained.

An ancestry entry contains an array of counters, each of which corresponds to a replication server. E.g., if there are three replication servers in the system, an entry in the ancestry table contains three corresponding counters. Whenever a (deep or shallow) control for an object **L** is granted or revoked, each server updates its ancestry table by scanning each directory along the path from **L** to the mount point, adjusting counters for the server that owns the control. A replication server also updates its ancestry table appropriately if a controlled object is moved, linked, or unlinked during directory modifications.



Consider three replication servers: S0, S1, and S2. Simultaneously, S0 requests (deep or shallow) control of directory b, S1 requests control of directory c, and S2 requests deep control of directory a. According to the rules listed in Figure 3, S0 and S1 succeed in their primary server elections, but S2's election fails due to conflicts. S2 then retries by asking for shallow control of a.

Figure 5.3: Potential conflicts in primary server election caused by deep control.

A replication server needs only one lookup in its ancestry table to tell whether a directory subtree holds an object under the control of a different server: It first finds the mapping entry of the directory from its ancestry table, and then examines that entry's counter array. If the counter on any replication server other than the one that issues the deep control request has a non-zero value, the replication server knows that some other server currently controls a descendant of the directory, so it rejects the deep control request.

5.2.2 Primary Server Election with Deep Control

With the introduction of deep control, two primary server election requests on two different objects can conflict if one of them wants deep control on a directory, as the example in Figure 5.3 illustrates. To guarantee progress during conflicts, we extend the consensus algorithm for primary server election as follows.

When a replication server receives a shallow control request for an object **L** from a peer server **P** but cannot grant the control according to the rules listed in Figure 5.1, it replies to **P** with the identifier of the primary server that currently controls **L**. On the other hand, if a replication server judges that it cannot grant a deep control request according to deep control rules, it simply refuses. A server downgrades a deep control request to shallow if it fails to accumulate acknowledgments from a majority of the replication servers. Then with shallow controls only, primary server election reverts to the original consensus algorithm.

5.2.3 Performance and Concurrency Tradeoff

The introduction of deep control introduces a tradeoff between performance and concurrency. A primary server can process any client update in a deep-controlled directory,

which substantially improves performance when an application issues a burst of updates. This argues for holding deep control as long as possible. On the other hand, holding a deep control can introduce conflicts due to false sharing. In this subsection, we strive for balance in the trade-off between performance and concurrency when employing shallow and deep controls.

First, we postulate that the longer a server controls an object, the more likely it will receive conflicting updates, so we start a timer on a server when it obtains a deep control. The primary server resets its timer if it receives a subsequent client update under the deep-controlled directory before the timeout. When the timer expires, the primary server relinquishes its role.

Second, recall that in a system with multiple writers, we increase concurrency by issuing a revoke request from one server to another if the former server receives an update request under a directory deep-controlled by the latter. Locality of reference suggests that more revoke requests will follow shortly, so the primary server shortens the timer for relinquishing its role for that directory.

Third, when a primary server receives a client write request for a file under a deep-controlled directory, it distributes a new shallow control request for that file to other replication servers. The primary server can process the write request immediately without waiting for replies from other replication servers as it is already the primary server of the file's ancestor. However, with a separate shallow control on the file, subsequent writes on that file do not reset the timer of the deep controlled directory. Thus, a burst of file writes has minimal impact on the duration that a primary server holds a deep control. Furthermore, to guarantee close-to-open semantics, a replication server need only check whether the accessed file is associated with a shallow control before processing a client read request, instead of scanning each directory along the path from the referred file to the mount point.

Fourth, a replication server can further improve its performance by issuing a deep control request for a directory that contains many frequently updated descendants if it observes no concurrent writes. This heuristic is easy to implement with the information recorded in the ancestry table: a replication server can issue such a request for directory **D** if it observes that in the ancestry entry of **D**, the counter corresponding to itself is beyond some threshold and the counters of all other replication servers are zero.

The introduction of deep control provides significant performance benefits, but can adversely affect data availability in the face of failure: if a primary server with deep control on a directory fails, updates in that directory subtree cannot proceed until the failed primary server is recovered. Recapitulating the discussion of false sharing above, this argues in favor of a small value for the timer. In the next section, we show that timeouts as short as one second are long enough to reap the performance benefits of deep control. Combined with our assumption that failure is infrequent, we anticipate that the performance gains of deep control far outweigh the potential cost of server's failing while holding deep control on directories.

5.3 Evaluation

In this section, we evaluate the performance of the hierarchical replication control protocol with a series of experiments over simulated wide area networks. We start with a coarse evaluation in Section 5.3.1 using the SSH-Build benchmark, and find that hierarchical replication control is very successful in reducing overhead, even when the time that deep control is held is short. In Section 5.3.2, we explore system performance with the NAS Grid Benchmarks in simulated wide area networks and find that our replicated file system holds a substantial performance advantage over a single server system. At the same time, it provides comparable and often better performance than GridFTP [12], the conventional approach for moving large data sets in the Grid.

We conducted the experiments presented in this section with a prototype implemented in the Linux 2.6.16 kernel. Servers and clients run on dual 2.8GHz Intel Pentium4 processors with 1 MB L2 cache, 1 GB memory, and onboard Intel 82547GI Gigabit Ethernet card. The NFS configuration parameters for reading (rsize) and writing (wsize) are set to 32 KB, the recommended value for WAN access. We use Netem [55] to simulate network latencies. Our experiments focus on evaluating the performance impact of WAN delays. Hence, we do not simulate packet loss or bandwidth limits in our measurements, and enable the `async` option (asynchronously write data to disk) on the NFS servers. Although not comprehensive, we expect that our settings closely resemble a typical Grid environment — high performance computing clusters connected by long fat pipes.

All measurements presented are mean values from five trials of each experiment; measured variations in each experiment are negligible. Each experiment is measured with a warm client cache, but the temperature of the client cache has little effect on the presented results.

5.3.1 SSH Build Evaluation

In this section, we evaluate the performance of hierarchical replication control using the SSH-Build benchmark. The SSH-Build benchmark [123] runs in three phases. The *unpack* phase decompresses a tar archive of SSH v3.2.9.1. This phase is relatively short and is characterized by metadata operations on files of varying sizes. The *configure* phase builds and runs various small programs that check the configuration of the system and automatically generates header files and Makefiles. The *build* phase compiles the source tree and links the generated object files into the executables. The last phase is the most CPU intensive, but it also generates a large number of temporary files and a few executables in the compiling tree.

Before diving into the evaluation of hierarchical replication, we look at the performance when accessing a single distant NFSv4 server. Figure 5.4 shows the measured times when we run the SSH-Build benchmark with an increasingly distant file server. In the graph, the RTT marked on the *X*-axis shows the round-trip time between the client and the remote server, starting with 200 μsec, the network latency of our test bed LAN.

Figure 5.4 shows (in log-scale) that the SSH build that completes in a few minutes on a local NFSv4 server takes hours when the RTT between the server and the client grows to tens of milliseconds. The results further confirm our finds presented in Chapter 4 — it is impractical to execute update-intensive applications using a stock remote NFS server. Network RTT is the dominant factor in NFS WAN performance, which suggests the desirability of a replicated file system that provides client access to a nearby server.

Next, we compare the time to build SSH using fine-grained replication control and hierarchical replication control with a local replication server and an increasingly distant replication server. The results, shown (in linear scale) in Figure 5.5, demonstrate the performance advantage of file system replication. Even with fine-grained replication control, adding a nearby replication server significantly shortens the time to build SSH, as expensive

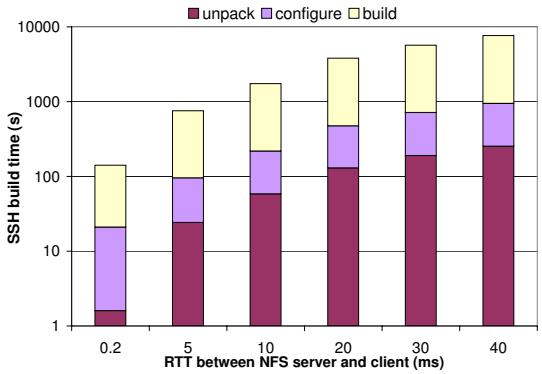


Figure 5.4: SSH build on a single NFSv4 server.

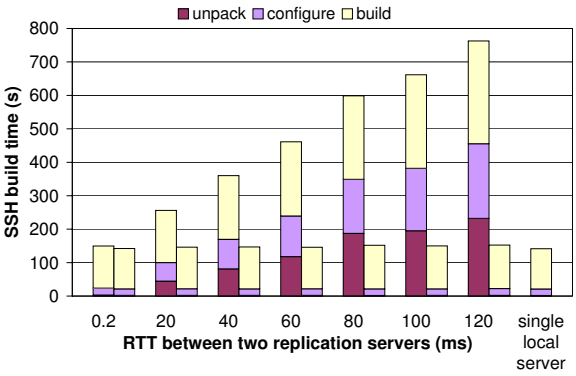


Figure 5.5: Fine-grained replication control vs. hierarchical replication control.
The first column shows the time to build SSH using fine-grained replication control. The second column shows the time when using hierarchical replication control. For runs with hierarchical replication control, the primary server relinquishes deep control if it receives no client updates for one second.

reads from a remote server are now serviced nearby. Moreover, we see dramatic improvement with the introduction of hierarchical replication control — the penalty for replication is now negligible, even when replication servers are distant.

In Section 5.2, we discussed the use of a timer for each deep-controlled directory to balance performance and concurrency but did not specify the timeout value. To determine a good value for the timer, we measured the time to build SSH for timeout values of 0.1 second, 0.5 second, and 1 second, as the RTT between the local replication server and the remote replication server increases. Figure 5.6 presents the results.

As the data shows, when we set the timeout value to one second, the SSH build with a distant replication server runs almost as fast as one accessing a single local server. Furthermore, almost all of the performance differences among the three timeout values come from

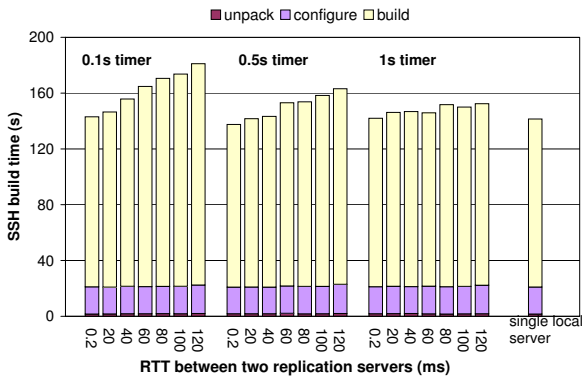


Figure 5.6: Deep control timeout values.
The diagram shows the time to build SSH using hierarchical replication control when the timeout for releasing a deep control is set to 0.1 second, 0.5 second, and 1 second.

the CPU intensive *build* phase. For the *unpack* and *configure* phases, which emit updates more compactly, even a tiny timeout value yields performance very close to that for single local server access. Of course, in practice the "optimal" timeout value depends on the workload characteristics of the running applications. However, the SSH build experiment suggests that a small timer value — a few seconds at most — captures most of the bursty updates.

5.3.2 Evaluation with Grid Benchmarks

The NAS Grid Benchmarks (NGB), released by NASA, provide an evaluation tool for Grid computing [48]. The benchmark suite evolves from the NAS Parallel Benchmarks (NPB), a toolkit designed and widely used for benchmarking on high-performance computing [14]. An instance of NGB comprises a collection of slightly modified NPB problems, each of which is specified by class (mesh size, number of iterations), source(s) of input data, and consumer(s) of solution values. The current NGB consists of four problems: Embarrassingly Distributed (ED), Helical Chain (HC), Visualization Pipe (VP), and Mixed Bag (MB).

ED, HC, VP, and MB highlight different aspects of a computational Grid. ED represents the important class of Grid applications called parameter studies, which constitute multiple independent runs of the same program, but with different input parameters. It requires virtually no communication, and all the tasks in it execute independently. HC represents long chain of repeating processes; tasks in HC execute sequentially. VP simulates

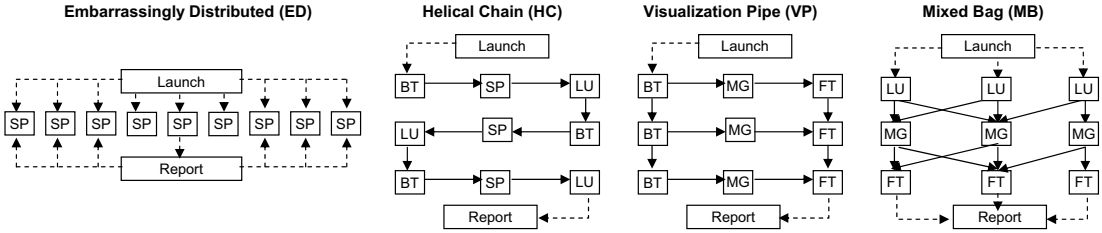


Figure 5.7: Data flow graphs of the NAS Grid Benchmarks.

	HC			VP				MB		
Class	BT→SP	SP→LU	LU→BT	BT→MG	MG→FT	BT→BT	FT→FT	BT→LU	LU→MG	MG→FT
S	169K	169K	169K	34K	641k	169K	5.1M	N/A	34K	641K
W	1.4M	4.5M	3.5M	271K	41M	1.4M	11M	N/A	702K	41M
A	26M	26M	26M	5.1M	321M	26M	161M	N/A	5.1M	321M
B	102M	102M	102M	21M	321M	102M	641M	102M	21M	321M

Table 5.1: Amount of data exchanged between NGB tasks.

logically pipelined processes, like those encountered when visualizing flow solutions as the simulation progresses. The three tasks included in VP fulfill the role of flow solver, post processor, and visualization, respectively. MB is similar to VP, but introduces asymmetry. Different amounts of data are transferred between different tasks, and some tasks require more work than others.

Figure 5.7 illustrates the Data Flow Graph for each of these benchmarks. The nodes in the graph, indicated by the rectangular boxes, represent computational tasks. Dashed arrows indicate control flow between the tasks. Solid arrows indicate data as well as control flow. Launch and Report do little work; the former initiates execution of tasks while the latter collects and verifies computation results.

The NGB instances are run for different problem sizes (denoted *Classes*). For the evaluation results presented in this dissertation, we use four Classes: S, W, A, and B. Table 5.1 summarizes the amount of data communicated among tasks for these Classes. [2]

A fundamental goal of Grid computing is to harness globally distributed resources for solving large-scale computation problems. To explore the practicality and benefit of using NFS replication to facilitate Grid computing, we compare the performance of running NGB

[2]Figure 5.7 illustrates the Data Flow Graphs for Class S, W, and A. For Class B, ED includes 18 parallel SP tasks; HC includes 6 layers of BT, SP, and LU; VP includes 6 layers of BT, MG, and LU; MB includes 4 layers: BT, LU, MG, and FT, and each layer includes 4 tasks [48].

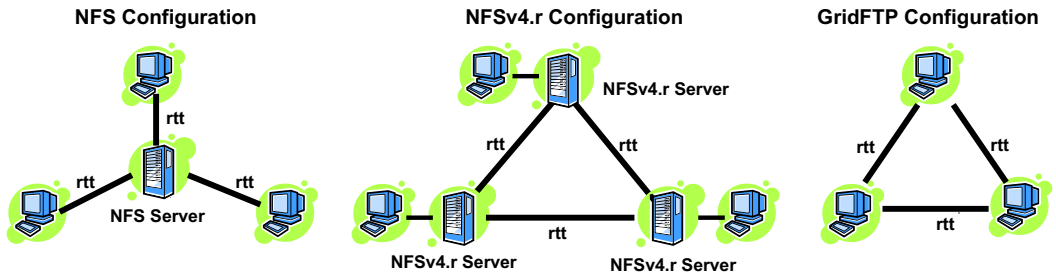


Figure 5.8: NGB evaluation experiment setup.

under three configurations, referred as NFS, NFSv4.r, and GridFTP. Figure 5.8 illustrates the experiment setup.

In the experiments, we use three computing nodes to emulate three computing clusters, with the RTT between each pair varying from 200 μsec to 120 msec. In the NFS configuration, the three computing nodes all connect to a single NFS server. In the NFSv4.r configuration, we replace the single NFS server with three replicated NFS servers, with each computing node connected to a nearby server. In the GridFTP configuration, we use GridFTP to transfer data among computing nodes. The GridFTP software we use is `globus-url-copy` from Globus-4.0.2 toolkit. In our experiments, we start eight parallel data connections in each GridFTP transfer, which we find provides the best measured performance for GridFTP. The NFSv4.r prototype also supports parallel data connections between replicated NFS servers, but in the experiments presented here, the performance improvement using multiple data connections is small, so we report results measured with a single server-to-server data connection only.

For the GridFTP configuration, we run the NGB tasks using the Korn shell Globus implementation from the NGB3.1 package. In this implementation, a Korn shell script launches the NGB tasks in round robin on the specified computing nodes. Tasks are started through the `globusrun` command with the batch flag set. After a task completes, output data is transferred to the computing node(s), where the tasks require the data as input. A semaphore file is used to signal task completion. Tasks poll their local file systems for the existence of the semaphore files to monitor the status of the required input files. After all tasks start, the launch script periodically queries their completion using `globus-job-status` command.

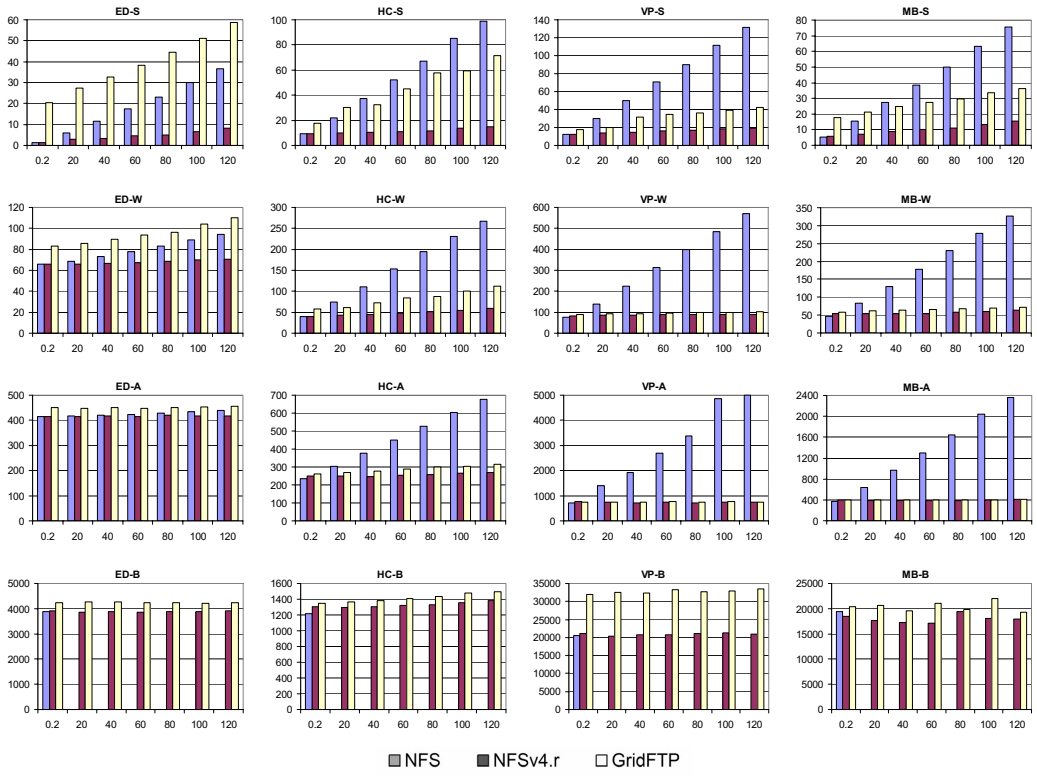


Figure 5.9: Turnaround times (seconds) of NGB on NFS, NFSv4.r, and GridFTP. In each figure, *X*-axis denotes the RTT (milliseconds) shown in Figure 5.8, and *Y*-axis denotes the measured turnaround times. For Class B, we omit the experiments of accessing a single remote NFS server, which take extremely long times; the results of using a single local NFS server are presented on the left side of each figure for comparison.

For the NFS and NFSv4.r setups, we extended the original NGB Korn shell scripts. The modified programs use `ssh` to start NGB tasks in round robin on the specified computing nodes. The tasks and the launch script poll for the status of the required input data and tasks with semaphore files, as above.

Figure 5.9 shows the results of executing NGB on NFS, NFSv4.r, and GridFTP as the RTT among the three computing nodes varies from 200 μsec to 120 msec. The data presented is the "measured turnaround" time, i.e., the time between starting a job and obtaining the result. With GridFTP, turnaround time does not include deployment and cleanup of executables on Grid machines. The time taken in these two stages ranges from 10 seconds to 40 seconds as the RTT increases from 200 μsec to 120 msec.

Evidently, in Grid computing, deployment and cleanup can sometimes take significant time with large size of executables and input data [56]. Furthermore, in some cases, it is

class	S				W				A				B			
Bench	ED	HC	VP	MB	ED	HC	VP	MB	ED	HC	VP	MB	ED	HC	VP	MB
Time (s)	2	1	9	6	217	31	83	101	138	223	930	870	12775	1231	25567	18290

Table 5.2: Turnaround times of executing NGB on a single computing node with a local ext3 file system.

hard for users to determine which files to stage [113]. With NFS and NFSv4.r, on the other hand, there is no extra deployment and cleanup time, because computing nodes access data directly from file servers. Even so, the times we report do not reflect this inherent advantage.

The histograms in Figure 5.9 show that performance with a single NFS server suffers dramatically as the RTT between the server and the computing nodes increases. Except for the ED problem - whose tasks run independently - on larger data sets (W and A), the experiments take a very long time to execute when the RTT increases to 120 msec. In fact, the times are even longer than the times measured when running the problems on a single computing node without parallel computing. (Table 5.2 shows the NGB execution times on a single computing node with a local ext3 file system.) Clearly, in most cases it is impractical to run applications on widely distributed clients connected to a single NFS server, even for CPU intensive applications.

On the other hand, with NFSv4.r and GridFTP on large class sizes, run times are minimally affected by increasing RTT. When the class size is small (e.g., the results of Class S), NFSv4.r outperforms GridFTP, because the latter requires extra time to deploy dynamically created scripts and has extra Globus-layer overhead. For larger class sizes, the performances of NFSv4.r and GridFTP are generally comparable. The only exception is the experiment with Class B of VP. There, the running time on GridFTP is about 50% longer than the time measured when running the benchmark on NFSv4.r.

A closer analysis shows that the observed performance difference is caused by memory contention introduced by Globus. Globus starts a job manager for each scheduled task on a computing node. The performance overhead added by Globus job managers is usually small. However, an FT task of Class B demands a large amount of memory. As a result, the Globus job managers suffer memory contention on the computing node where FT tasks execute, which leads to a noticeable slowdown. The experiment with Class B of MB also experiences memory contention. Class B of MB consists of 4×4 tasks (see Footnote 2). As

we assign tasks in round robin, two FT tasks are scheduled to run on the same computing node. Due to memory contention, the execution times for these two tasks are significantly longer than executing them individually and we observe the relatively high variances in the measured results of this experiment. However, all the experiments with Class B of MB experience this slowdown, so the performance difference is small. We anticipate that the memory contention problem described here can be avoided with a well-designed job scheduler. Since the focus of our study is on data access in Grid computing, more detailed discussions are beyond the scope of this discussion.

As presented in Section 5.2, an NFSv4.r server asynchronously distributes updates to other replication servers for any file or directory modified on it. When a task finishes writing an output file and closes it, it takes a short delay for the written data to be completely transferred to a remote replication server. Since a task usually creates a semaphore file immediately after closing its output file, the written data may be still in transmission at the time that another task on a remote site starts reading the file. As the replication protocol specifies, these read requests are forwarded to the primary server on which the file is written. When the file copies on all of the replication servers are synchronized, the primary server releases its control for the written file and the subsequent reads for the file are served on a nearby replication server. In contrast, with GridFTP, a file needs to be explicitly transferred to a computing node where the file is required as the input of a task. A task cannot start running until the transmission of its input file(s) completes. As the NGB tasks are heavily computation intensive, the advantage of saving the first-byte latency is relatively small. However, when the size of data that needs to be transferred becomes large, paralleling computation and data distribution can provide more performance benefits.

In summary, the NGB experiments demonstrate that well-engineered replication control provides superior file system semantics and easy programmability to WAN-based Grid applications without sacrificing performance.

5.4 Related Work

The use of multiple granularities of control to balance performance and concurrency has been studied in other distributed file systems and database systems. Many modern transactional systems use hierarchical locking [51] to improve concurrency and performance

of simultaneous transactions. In distributed file systems, Frangipani [114] uses distributed locking to control concurrent accesses among multiple shared-disk servers. For efficiency, it partitions locks into distinct lock groups and assign them to servers by group, not individually. Lin et al. studied the selection of lease granularity when distributed file systems use leases to provide strong cache consistency [122]. To amortize lease overhead across multiple objects in a volume, they propose volume leases that combine short-term leases on a group of files (volumes) with long-term leases on individual files. Farsite [10] uses content leases to govern which client machines currently have control of a file's content. A content lease may cover a single file or an entire directory of files.

5.5 Summary

To improve the performance of applications that consist of many synchronous writes or metadata updates, we refined the update distribution strategy presented in Chapter 4 by relaxing the failure resilience of data under modification and developed a hierarchical replication control protocol that dynamically elects a primary server at various granularities. This chapter describes these extensions in detail and evaluates their effects on application performance. Our experimental results indicate that with the introduction of the hierarchical replication control, the overhead of replication is negligible even when applications mostly write and replication servers are widely distributed. Furthermore, evaluation with the NAS Grid Benchmarks demonstrates that our system provides comparable — and often better — performance than GridFTP, the de facto standard for Grid data sharing.

CHAPTER 6

Tradeoff between Performance and Failure Resilience

In the previous chapters, we describe a mutable replicated file system that provides Grid applications efficient data access with conventional file system semantics. With mutable replication, a fundamental challenge is to maintain consistent replicas without introducing high performance overhead. Preserving consistency is essential to guaranteeing correct behavior during concurrent writes. It is also needed to guarantee durability of data modifications in the face of failure.

To reduce the cost of concurrency control, we introduce a hierarchical replication control protocol that exploits locality of reference in application updates to amortize the cost of synchronization over multiple update requests. Performance evaluation shows that with hierarchical replication, the overhead of concurrency control is reduced to a negligible amount, even for update-intensive applications.

To address the cost of durability guarantees, we relax the expensive update distribution requirement to improve performance of synchronous writes and metadata updates. Specifically, we allow a primary server to respond immediately to a client update request before distributing the update to other replication servers. If the primary server fails, we make the object being modified inaccessible to guarantee the durability of data written by the client and acknowledged by the primary server. Applications can decide for themselves whether to wait for the failed server to recover or to reproduce the computation results. With this strategy, we trade the failure resilience of our system for the improved write performance.

The replicated file system discussed so far offers good performance in failure-free cases. However, there is a tradeoff between performance and cost to reproduce lost computation results during failures. In a hypothetical failure-free world, the benefits of maintaining syn-

chronous replication servers may be entirely lost for write-mostly applications. However, back in the real world, failure *is* an option, especially in an experimental computing platform such as the current Grid. Furthermore, most Grid applications have lengthy execution time. They have much to lose if the failure of a file server destroys the only copy of output. Recall the evaluation of the ATLAS applications in Chapter 4. It shows that the cost of maintaining synchronous data copies on remote replication servers is negligible if applications write at a moderate rate. This suggests that it is proper to synchronously replicate the output of an application if its performance depends little on the write I/O throughput but requires long execution time.

For long-running applications, maintaining up-to-date replication servers acts as a kind of insurance for compute time already spent. Furthermore, multiple synchronous replication servers provide extra insurance by protecting the computation from multiple failures. However, the amount of protection provided by multiple servers depends not just on their number, but also on their location. In fact, we find that failure is (inversely) correlated with the distance between servers. For the same reason that system administrators insist on storing backups off-site, it is unwise to place all of the synchronous replication servers on a single local area network. This indicates a vexing tension: on one hand, placing multiple synchronous replication servers close to the computation site minimizes update and synchronization costs; on the other hand, placing them far from one another increases their resiliency to correlated failure.

In this chapter, we explore the tradeoff between performance and failure resilience when using different update distribution policies. The remainder of the chapter proceeds as follows. Section 6.1 discusses the tradeoff between performance and failure resilience in the presented replicated file system. Section 6.2 develops a failure model for distributed resources using PlanetLab trace data. Section 6.3 introduces a Markov model to evaluate the performance of a Grid application over a replicated file system in the presence of failures. In Section 6.4, we combine the failure and performance models to predict the performance of applications with different running time and write characteristics. Section 6.5 reviews related work and Section 6.6 summarizes.

6.1 Performance and Reliability Tradeoffs

When evaluating the tradeoff between performance and failure resilience, we focus on a specific class of Grid applications: those whose output can be reproduced by restarting or rolling back to a saved checkpoint, a strategy characteristic of long-running applications executing on clusters. In our replicated file system, updates are distributed to multiple file servers. If one or more file servers fail, the system can fully recover as long as one replication server holding fresh data is accessible. Applications connected to a failed file server can continue their executions straightaway by diverting their requests to the available serves. However, if no surviving server holds a fresh copy of data, the system cannot hide the failure from applications. In that case, the applications need to roll back to a saved checkpoint or restart their executions after switching to a working server.

Accordingly, the durability guarantee that our system provides determines the expected cost to recover from a failure that might occur during the execution of the program. Introducing synchronous replication servers into the file system improves durability and reduces the risk of losing the results of long-running applications if failure happens. On the other hand, the strength of the durability guarantee is determined by (1) the number of synchronous data copies maintained on different replication servers, and (2) the incidence of correlated failure among these servers. Guaranteeing high data durability requires the system to maintain up-to-date data copies on a number of replicas unlikely to fail at the same time. When applications emit updates at a high rate, this requirement can lead to a concomitantly high performance expense. In some cases, it makes more sense to trade durability for performance and let applications regenerate their execution results when the system cannot mask a failure.

Reducing the cost of updating replication servers suggests a number of design options, each providing a different tradeoff between performance and failure resilience.

As one option, we can allow a primary server to respond to a client update request when it has received update acknowledgments from a majority of the replication servers, the strategy we described in Chapter 4. With this requirement, as long as a majority of the replication servers are available, a fresh copy of the modified file or directory can always be recovered. However, for scientific applications that consist of many synchronous updates, performance suffers when a majority of replication servers are distant.

On the other hand, we can allow a primary server to respond immediately to a client update request and distribute the modified data to the other replication servers asynchronously, as discussed in Chapter 5. This has the obvious advantage of eliminating the update latency penalty. However, updates are at risk of loss if the primary server fails.

Between these two options, we can require that a primary server distribute updates to a specified number of replication servers before acknowledging a client update request. This still puts durability at risk, but reduces the risk: data is lost only if all of the updated servers fail simultaneously. Furthermore, while this approach reduces the cost of updating replication servers, it does not eliminate that cost.

We observe that the cost of updating a remote replication server is accounted for by its distance: updating nearby servers introduces low latency while updating far away servers leads to long latency. However, we hypothesize that the closer two servers are from each other, the more likely it is that they might fail at the same time. This introduces another tradeoff in designing a replication strategy.

Summarizing, maintaining synchronous replication servers can insulate a computation from failure, but increases the running time. For failure rates below some threshold, it is better not to distribute updates synchronously. When synchronous replication is advantageous, increasing the number of up-to-date replication servers improves the durability of application updates. Meanwhile, failure is correlated with the distance among these servers, so we should maintain synchronous data copies on distant servers as well as nearby ones. However, the cost of replication increases with the distance to the servers.

To determine the best replication strategy, we need to consider the failure conditions of the running environment, as well as application characteristics. Generally, we want to maintain more synchronous data copies when component failures are frequent and when applications are computation intensive. If failures are rare or applications rely heavily on synchronous writes or metadata updates, a delayed update distribution policy might provide a better performance tradeoff. In the following sections, we explore these tradeoffs.

6.2 Modeling Failure

To evaluate a replication strategy, we need to know the frequency, probability distribution, and correlation of failure. Our focus is on wide-area distribution, so we use Plan-

etLab [32] to exemplify a wide-area distributed computing environment. PlanetLab is an open, globally distributed platform, consisting (at this writing) of 716 machines, hosted at 349 sites, spanning 25 countries. All PlanetLab machines are connected to the Internet, which creates a unique environment for conducting experiments at Internet scale. We find the PlanetLab platform well-suited to study failure characteristics of large-scale distributed computing: PlanetLab nodes experience many of the correlated failures expected in widely distributed computation platforms. Moreover, failure traces of PlanetLab are collected over a long term and publicly available.

We use failure distribution data from the all-pairs ping data [112] collected from January 2004 to June 2005. The data set consists of ICMP echo request/reply packets ("pings") sent every 15 minutes between all pairs of PlanetLab nodes, 692 nodes in total. Each node recorded and stored its results locally and periodically transferred the results to a central archive. We consider a node to be live in a 15-minute interval if at least one ping sent to it in that interval succeeded. If the archive received no data from a node for a given time period, that node was considered to have failed. Thus, the failures detected in our study include nodes that crashed as well as network failures that partitioned nodes from the others. This agrees with the failure conditions in Grid computing: from an application's point of view, a network failure makes the data generated on a partitioned node inaccessible to other computing elements and requires that the partition be recovered to advance the computation.

An important measure in reliability study is time-to-failure (TTF), i.e., continuous time intervals when a node is live. Figure 6.1 shows the cumulative frequency of PlanetLab node TTF. The mean TTF is 122.8 hours. Previous studies have shown that TTF can be modeled by a Weibull distribution [54, 85, 108] and our analysis agrees: the best-fit Weibull distribution generated with MATLAB, shown in Figure 6.1, agrees with the empirical data. The scale and shape parameters of the best-fit Weibull distribution are 8.0556E+04 and 0.3549, respectively.

We next investigate correlated failures among PlanetLab nodes. In related work, Chun et al. use conditional probabilities P(X is down $\mid$ Y is down) to characterize the correlated failures between nodes X and Y [33]. Since we assume that a failed node can be replaced with an active one when failure happens, we are more interested in the frequency that two nodes fail at the same time instead of the amount of time that two nodes are down

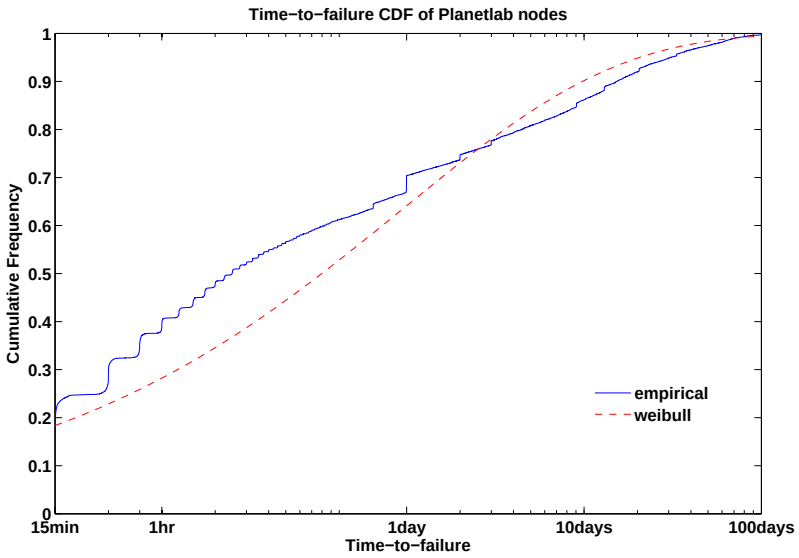


Figure 6.1: Time-to-failure CDF of PlanetLab nodes. The mean TTF is 4.42204E+05 seconds. The best-fit Weibull distribution parameters: scale = 8.0556E+04, shape = 0.3549.

simultaneously. We therefore quantify the failure correlations for nodes X and Y with the conditional probabilities P(X fails at time t | Y fails at time t). Similarly, we measure the failure correlations for nodes X_1, X_2, ..., X_n by computing the conditional probabilities P(X_2, ... , X_n all fail at time t | X_1 fails at time t). We note that in the formula, X_1, X_2, ..., X_n are all supposed to be alive before time t. Thus, given a group of nodes, our calculation uses only the failure times that satisfy this condition.

We first look at the failure correlations for nodes in the same site. Our analysis proceeds as follows. We first pick a node from each PlanetLab site and then select a different node from the same site to calculate the failure correlations. In the failure data we analyzed, 264 sites have more than two nodes (but only 259 of them contain more than two nodes that simultaneously live), 65 sites have more than three nodes, 21 sites have more than four nodes, and only 11 sites have more than five PlanetLab nodes.

Table 6.1 presents the average failure correlations computed with different number of nodes and PlanetLab sites. In the table, the first column indicates the number of nodes we select from a PlanetLab site to compute the failure correlations. The first row enumerates the number of PlanetLab sites that contain more than 2, 3, 4, and 5 nodes, respectively. The data marked in bold on row **N** is calculated with the failure data from all of the PlanetLab sites that contain at least **N** nodes. For comparison, we also compute the failure correlations with fewer sites, shown in the upper right part of the table above the diagonal.

Number of nodes	259 sites	65 sites	21 sites	11 sites
2	**0.526**	0.593	0.552	0.561
3		**0.546**	0.440	0.538
4			**0.378**	0.488
5				**0.488**

Table 6.1: Failure Correlations for PlanetLab nodes from the same site.

In spite of the small numbers of sites available for computing the failure correlations among multiple nodes, several inferences can be drawn from Table 6.1. First, there is a high probability that two nodes in the same site fail simultaneously - more than half of the time, if one node fails, another node in the same site also fails. Furthermore, as we increase the number of nodes that we consider within a site, correlated failures do not fall dramatically. Table 6.1 suggests that it is common for all nodes at a site to fail simultaneously. These failures might include administrators powering down all PlanetLab nodes in a site, or network failures that partition an entire site from the rest of network.

Next, we explore the failure correlations among nodes chosen from different sites. We hypothesize that failure correlation decreases with increasing number of nodes and distance between nodes, so we focus on the impact that these two aspects have on failure correlations.

To analyze the impact of distance on failure correlations, we partition nodes into equivalence classes for various RTT intervals, with the length of each RTT interval set to 10 milliseconds. Specifically, for a given node X, a number n, and a range $[rtt, rtt+10]$, we find all groups of n-1 nodes whose maximum RTT to X is between rtt and $rtt+10$. We then calculate the average failure correlations for all of these groups with different n values.

Figure 6.2 shows the results. For a given point $< x,y >$ in the figure, the x value gives the median RTT of the corresponding RTT interval and the y value shows the average failure correlations for that RTT interval.

We observe that correlated failure for nodes chosen from different sites is half of that shown in Table 6.1. Moreover, although increasing the number of nodes reduces failure correlations, we still see correlated failures of 5-10%, even when we consider failure of four or five nodes. These correlated failures are mostly caused by the testbed-wide DDoS attacks and system bugs.

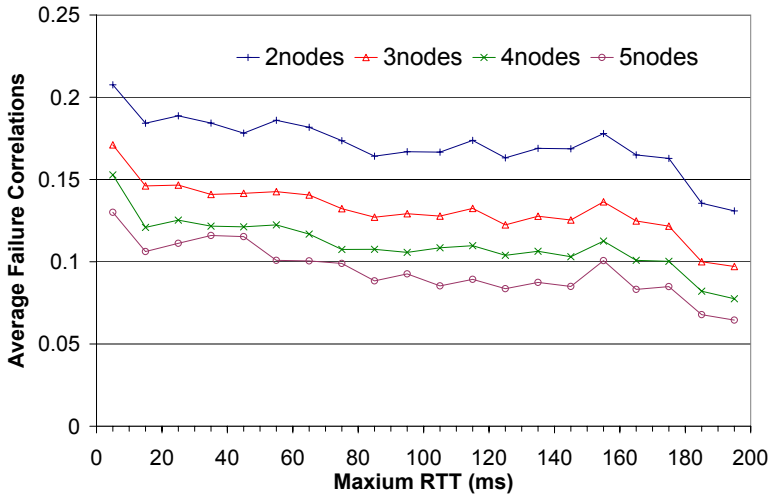


Figure 6.2: Failure correlations for PlanetLab nodes from different sites.

Figure 6.2 bears out our hypothesis that failure correlation tends to decline as the RTTs between nodes increase. For example, when the RTT between two PlanetLab nodes is a few msec., the failure correlation is around 0.2, but when the RTT is 200 msec., the failure correlation drops to 0.13.

Overall, the analysis of PlanetLab failure data shows that correlated failures are reduced as the number of nodes increases and as the distance between nodes increases. This suggests that we can improve durability by maintaining copies on remote replicas and by increasing the number of replicas. However, both of these strategies come at a cost: the former increases update latency while the latter imposes storage and network overheads. In the next section, we propose a model that uses failure statistics and application characteristics to estimate the expected execution time of an application for various replication policies. We then show how to use the model to minimize the expected execution time of a Grid computation by selecting an optimal replication policy given available storage resources.

6.3 The Evaluation Model

In this section, we describe a model for estimating the expected running time of an application that uses a replicated file system subject to failure. We use the following nomen-

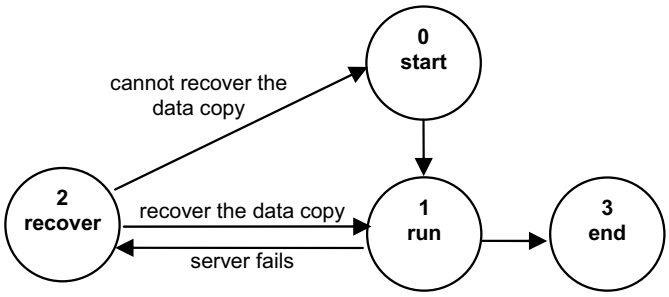


Figure 6.3: Four-state Markov chain describing the execution of an application over a replicated file system in the presence of failures.

clature, with some terms borrowed from previous studies by other researchers on optimal checkpoint intervals [86, 116, 119].

Failure-free no-replication running time (F) is the running time of an application in the absence of failure without replication. This is equal to the execution time with a single local server that does not fail.

Replication overhead (C) is the performance penalty for maintaining synchronous data copies on replication servers (which we call backup servers in the following discussion) in a failure-free execution of the application. We can estimate C as follows. First, we assume (and our experiments confirm) that the replication overhead is strictly proportional to the maximal distance between the primary server and the backup servers. Let RTT represent the maximal round-trip time (in msec.) between the primary server and backup servers and let C_{msec} denote the replication overhead to update a backup server with a one msec. round-trip time from the primary server. C_{msec} depends only on application write characteristics and can be measured during a test run of the application. We can then estimate the replication overhead $C = RTT \times C_{msec}$.

Recovery time (R) is the time for the system to detect the failure of a replication server and replace it with another active server.

Expected execution time (E) is the expected application execution time in the presence of failures. The model developed in this section is used to calculate E.

Utilization ratio (U), defined as $U = F/E$, describes the fraction of time that the system spends doing useful work. The model developed in this section is used to optimize U.

We model the execution of an application with a four-state Markov chain, shown in Figure 6.3. Application execution begins in an initial *start* state and makes an immediate

transition to the *run* state, where it remains until a replication server fails or the execution completes. Upon replication server failure, the execution is suspended by transferring to the *recover* state. During recovery, a replacement server is sought and the system attempts to recover the data under modification on the failed server. If a synchronous data copy survives on any active replication server, the system can recover the data on the application's behalf. On the other hand, if the failed server holds the only valid copy of the data (i.e., the server distributes updates to other replication servers asynchronously) or if all of the replication servers that maintain synchronous copies fail simultaneously, then the system cannot recover the data generated up to the point that the execution halted. After failure recovery, application I/O is directed to the replacement server. Then, depending on whether the output data generated by the application is recovered, the application either resumes its computation (continues on the *run* state) or restarts from the beginning (from the initial *start* state). When execution finishes, the application exits to the *end* state.

In the Markov model just described, the expected running time of an application in the presence of failure can be expressed as the expectation of the time to transit from the initial *start* state to the *end* state. This can be estimated using the specified time-to-failure distribution and the failure correlations of the replication servers that maintain synchronous data copies. In particular, the time-to-failure distribution determines the waiting time in the *run* state before moving to the *recover* state, while the failure correlation gives the probability of moving from the *recover* state to the *start* state. We assume that the replacement server to which the client is migrated is in the same local area network as the failed server, i.e., it has an identical latency to the client. Future work is required to examine this assumption in more detail.

In our study, we calculate the expected execution time of an application through simulation. We wrote a simulator that takes as input the failure distribution data and the running time parameters of an application with a specified replication policy, i.e., F, C, and R. The simulation proceeds as follows. The simulator begins with the *start* state and moves directly to the *run* state. In the *run* state, the simulator either waits for F+C and then transitions to the *end* state, or jumps to the *recover* state if a failure happens within F+C. After spending an amount of time R in the *recover* state, the simulator either moves back to the *run* state or restarts from the *start* state, with the probability of the latter equal to the given failure correlation. We assume that the same replication policy is used for an application throughout

a simulation. Implicitly, the replication overhead C does not change after an application I/O is directed to a replacement server.

6.4 Simulation Results

In this section, we use discrete event simulation, based on the analyzed PlanetLab failure statistics from Section 6.2, to evaluate the efficiency, i.e., the utilization of different replication policies with various application running time characteristics.

We use the replicated file system described in the last two chapters as the reference model for our study. Since the system can automatically detect and recover from the failure of a replication server, we suggest that a small amount of time for failure recovery is reasonable. In an experiment (not presented in this dissertation), we measured the recovery time for a 100 MB file. The whole process takes about 2 minutes, which includes the timeout period for the replacement server to detect the failure and the time to synchronize the file on all active replication servers. In our simulation experiments, we fix the failure recovery time R to 10 minutes. Further analysis (not detailed in this dissertation) shows that varying R in the range from 1 minute to 1 hour does not have much effect on the results for the (much larger) expected application running times we are most interested in.

In our simulation, each measured expected execution time is the average execution time from 100,000 consecutive runs of simulation. The PlanetLab data does not contain enough failures for so many simulations, so we use MATLAB to generate time-to-failure data from the Weibull distribution that best fits the PlanetLab failure data, analyzed in Section 6.2. For failure correlations with different replication configurations, we use the probability data calculated in Section 6.2.

Figure 6.4 shows the results of the simulation. In each graph, the *X*-axis indicates the maximum RTT (in milliseconds) between the primary server and backup servers, and *Y*-axis indicates the utilization ratio.

We assume that asynchronous update distribution adds no performance cost to an application's execution, i.e., C is equal to 0. Furthermore, with asynchronous update distribution, no synchronous data copy is available if the primary server fails, so we always restart an execution from the beginning. Thus, the utilization ratio with asynchronous update distribution depends only on the application running parameters and time-to-failure

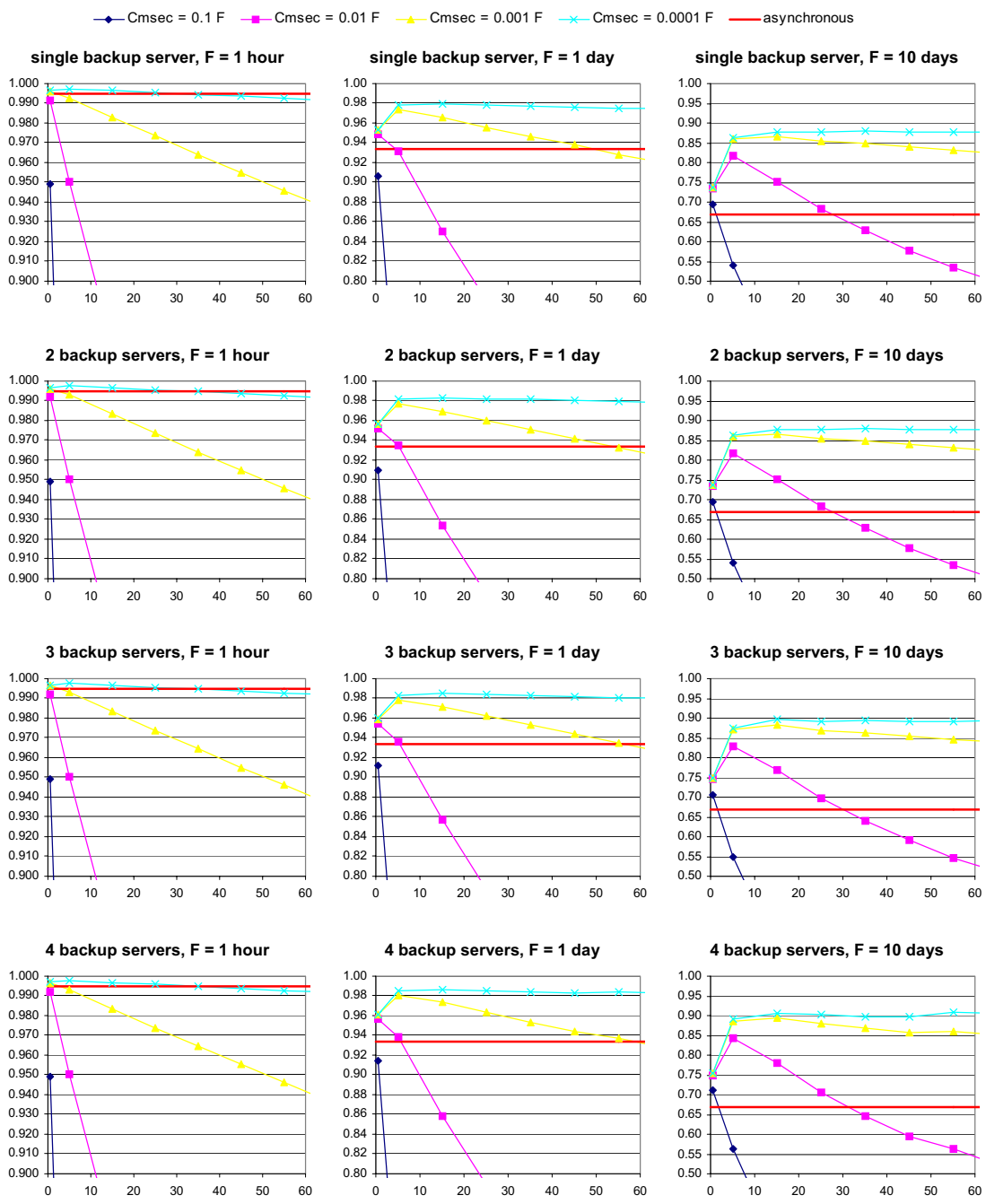


Figure 6.4: Utilization ratio (F/E) as the RTT between the primary server and backup servers increases. In each graph, *X*-axis indicates the maximum RTT (in ms) between the primary server and backup servers, and *Y*-axis indicates the utilization ratio.

distribution. The utilization ratios with asynchronous update distribution for F = 1 hour, F = 1 day, and F = 10 days are 0.996048, 0.947075, and 0.689764, respectively, which are marked as a red horizontal line in each graph.

The results suggest that applications with different characteristics benefit from different replication policies.

For applications that make heavy use of synchronous writes or metadata updates (C_{msec} = 0.1F), whether long- or short-running, maintaining synchronous replicated data copies is costly even with nearby backup servers, so asynchronous update distribution is usually prescribed. For very long-running applications (10 days), the cost of losing intermediate computation results becomes enormous, so it is beneficial to maintain synchronous data copies on local backup servers. We observe that the utilization ratio for long-running applications is relatively low. This indicates the benefit of using checkpoints to reduce the amount of computation placed at risk.

For applications that write at a moderate rate (C_{msec} = 0.01F), maintaining nearby backup servers provides the highest utilization. When the running time of an application is small, a local backup server offers the best tradeoff between performance and failure resilience. However, when the execution time is larger, i.e., F = 10 days, the risk of losing intermediate computation results because of multiple failures becomes overwhelming. Here, maintaining synchronous data copies in the same local area network is inadequate since as we saw in Table 6.1, this replication policy cuts correlated failures only in half. Instead, the simulation indicates that the performance penalty of backing up data to a different site is more than compensated by the expected reduction in the execution time lost to correlated failure.

If applications make few synchronous writes or metadata updates (C_{msec} = 0.0001F), replication overhead is relatively small even when we maintain synchronous data copies far away from the primary server. For these applications, maintaining remote backup servers provides the highest utilization.

Finally, we find that increasing the number of backup servers does not yield much improvement in utilization. For example, with F = 10 days, the maximum utilization ratio increases from 0.68 to 0.71 as we raise the number of backup servers from 1 to 4. Furthermore, we observe that increasing the distance between the primary server and backup servers provides limited advantage even for read-dominant applications. That is, although

the failure analysis in Section 6.2 shows that increasing the number of synchronous data copies and the distance among the maintained data copies helps to reduce correlated failures, they offer small benefits for reducing the expected running time. These findings follow from the low overall failure rate and the relatively rapid recovery time compared with the time required to regenerate the lost computation results; correlated failures are addressed effectively by maintaining a single backup server in a different site.

In summary, our simulation results indicate that applications with different characteristics benefit most from different replication policies. A Grid infrastructure that provides a mechanism for choosing a replication policy based on application characteristics and the failure conditions of the environment can improve the utilization of computational resources. Focusing on the tradeoff between performance and failure resilience, our evaluation omits other replication overhead such as network bandwidth and storage space. However, the work presented in this chapter constitutes a first step towards dynamic replication management in the Grid computing.

6.5 Related Work

Our work is related to three research fields: availability studies on system, Internet services and experimental wide-area computing platforms, optimal checkpoint interval analysis, and wide-area replication studies.

6.5.1 Availability Studies

Availability is widely studied by other researchers on different computing systems. In particular, we take many insights from the previous works on availability of cluster systems, Internet services, the PlanetLab testbed [32], and the continuously growing Grid computing platforms [30, 45].

There is a large amount of work on measuring and characterizing failure in cluster systems. Xu et al. [120] studied the error logs from Windows NT servers. Their analysis shows that while the average availability of individual servers is over 99%, there is a high probability that multiple servers fail within a short interval. Sahoo et al. [100] analyzed the failure data collected at an IBM research center. They find that failure rates exhibit

time varying behavior and different forms of strong correlation. Heath et al. [54] studied the reboot logs from three campus clusters and observed that the time between reboots is best modeled by a Weibull distribution. This observation is also indicated by Nurmi et al. [85] who investigate the suitability of different statistical distributions to model machine availability and by Schroeder et al. in more recent work [108] that analyzes failure logs collected over the past 9 years at Los Alamos National Lab.

Pang et al. [89] investigate the availability characteristics of the Domain Name Service (DNS). They observe that DNS server unavailability is most often not correlated within individual network domains. Padmanabhan et al. [88] measured the faults when repeatedly downloading content from a collection of websites. Regarding websites that have replicas, they find that most correlated replica failures are due to websites replicated on the same subnet. Recent availability studies on peer to peer systems [21, 26, 31, 52, 103] reveal low host availabilities in such environments, as most of their participants are unreliable end-users' desktops and can depart the system at will.

Several recent works investigate the availability characteristics of the globally distributed PlanetLab platform. Chun et al. [33] study all-pairs ping data set [112] collected on PlanetLab over a three-month period. They find that failures on the PlanetLab exhibit high correlations. A similar finding is also observed and further addressed by Yalagandula [121] and Nath [83] in their studies on correlated failures of PlanetLab nodes.

As the Grid technology is still under rapid development, few papers study component failures of the Grid infrastructure; instead, existing works mostly focus on job failures. The Grid2003 report [44] indicates that some projects observed job failure rates as high as 30% and a large number of such failures are caused by over-filled disks. Li et al. [72] analyzed the job failure data collected from the LHC computing Grid and argued for the importance of taking into account historical failure patterns when allocating jobs. Hwang et al. [58] propose a framework that allows Grid applications to choose desired fault tolerant mechanisms and evaluate the effects of recovery techniques supported.

6.5.2 Optimal Checkpoint Interval

Our work is similar in spirit to determining optimal checkpoint intervals in high-performance computing. Checkpoint is a time-tested technique for ameliorating the amount of re-

execution in the case of failure. Checkpoint introduces substantial performance overhead, so it is important to select a checkpoint frequency that minimizes the expected execution of an application in the presence of failures.

The selection of optimal checkpoint intervals has been studied for a long time. The problem was first formalized by Chandy et al. on transactional systems [29]. Subsequently, Vaidya [116] derived equations of average performance with checkpointing and rollback recovery in high-performance computing by assuming a Poisson failure distribution model. Wong et al. [119] model the availability and performance of synchronous checkpointing in distributed computing with and without load redistribution during failures. Plank et al. [93] examine three collections of workstation failure data to assess the applicability of the theoretical checkpoint equations that assume Poisson failure distributions. Their work shows that many of the theoretical results are applicable for checkpointing long-running computations even though actual failure rates do not follow Poisson processes. Later, the authors investigated the performance of parallel computing with checkpoints, assuming that failures and repairs follow Poisson models [94]. Their results show that the optimal number of active processors can vary widely, and the number of active processors can have a significant effect on application performance. Oliner et al. [86] evaluate the periodic checkpoint behavior of BlueGene with a failure trace collected from a large cluster. The study shows that when the overhead of checkpoint is high, overly frequent checkpointing can be more detrimental to performance than failure itself.

6.5.3 Related Works on Replication

Many systems use replication to reduce the risk of data loss. Total Recall [22] measures and predicts the availability of its constituent hosts to determine the appropriate redundancy mechanisms and repair policies. Glacier [53] uses massive redundancy to mask large-scale correlated failures. Carbonite [34] strives only to create data copies faster than they are destroyed by permanent failures in order to reduce the bandwidth cost of replication maintenance. However, all of these studies focus on masking host unreliability in peer-to-peer systems. The tradeoff between performance and failure resilience is not addressed.

Some recent studies investigate fault-tolerant techniques against correlated failures. Phoenix [59] takes advantage of platform diversity in cooperative systems. Oceanstore

[117] uses introspection to discover groups of nodes that are independent in their failure characteristics, then chooses data replicas from such a group to enhance system availability. We note that our system can utilize these techniques as well, although the evaluation of their benefits is beyond the scope of this dissertation.

6.6 Summary

In this chapter, we describe an evaluation model for determining the best-fit replication policy given specified failure statistics and application characteristics. We focus on a special class of Grid applications: long-running, compute-intensive, and write-mostly. Using failure data from the PlanetLab platform, we evaluate the efficiency of different replication policies in terms of the overhead they introduce and the expected cost to reproduce the execution results in case the system cannot mask a failure from an application. Our results show that application characteristics dictate desire different replication policies to maximize the utilization of computational resources. A replication system should balance the tradeoff between performance and failure resilience flexibly, based on the failure conditions of the running environment as well as these application characteristics.

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

Sharing data in scientific collaborations that involve many institutions around the world demands a large-scale storage system that is reliable and efficient, yet at the same time, convenient to use. To meet these requirements, this dissertation focuses on enhancing network transparency of data access in global collaborations. We developed a naming scheme that supports a global name space and location independent naming, which facilitates data sharing, distribution, and management. To improve performance and failure resilience, we designed and implemented a replicated file system that supports mutable replication with strong consistency guarantees, high failure resiliency, and good scaling properties.

In system design, we first developed a fine-grained replication control protocol that dynamically elects a primary server at the granularity of a single file or directory. The protocol demonstrates a significant performance advantage over the conventional single server distribution model and breaks new ground in performance and availability for read-dominant applications. However, further performance evaluation shows considerable overhead for large synchronous writes and bursty metadata updates. To improve performance for synchronous updates, we relaxed the expensive durability requirement, trading the failure resilience of data under modification for reduced response time. To improve performance for bursty metadata updates, we developed a hierarchical replication control protocol that allows a primary server to asset control at different granularities: a file, a directory, or a whole directory subtree. Using experimental evaluation, we showed that with the introduc-

tion of hierarchical replication control, the overhead of replication is negligible even when applications mostly write and replication servers are widely distributed.

The two replication control protocols provide different tradeoffs between performance and failure resilience. The fine-grained replication control protocol allows transparent failure recovery as long as a majority of replication servers are available, while the hierarchical replication control protocol trades failure resilience for improved performance by making data under modification inaccessible in case a primary server fails. We further developed an evaluation model for determining the best-fit replication policy given the specified failure statistics and application characteristics. The results show that different applications desire different replication policies to maximize the utilization of computational resources. A replication system should balance the tradeoff between performance and failure resilience flexibly, based on the failure conditions of the running environment as well as application characteristics.

7.2 Future Work

There are several directions for potential future work.

Optimize primary server release

In the current design, a primary server withdraws from its leading role by notifying other replication servers to stop forwarding their requests when the updated file is closed on it or has not been accessed for a long time. This strategy is not optimal in some situations. E.g., in a single writer case where write requests for a file are always received by one server, having the primary server delay releasing its role can eliminate redundant network traffic. In another example where write traffic switches from one replication server to another without any file close, better performance can be achieved if the first server advances the release of its primary role. Future study is required to develop an optimized solution that automatically adapts with various application workloads.

Optimize update distribution

In my design, a primary server distributes updates to other replication servers for each received client write request. However, data is not always shared among replication servers. For temporary files that are accessed by a single user, there is no need to distribute the written data to other replication servers. This suggests that a primary server should delay update distribution as late as possible. On the other hand, delaying update distribution might increase the response time of read requests later received on another replication server. Ideally, we want to distribute updates to other replication servers immediately for files that will be soon accessed on them, but defer distributing updates for a file that is accessed by a single user and soon to be deleted. This requires pre-knowledge of the data access pattern, which is hard to achieve in a dynamic running environment such as Grid. Some existing work has investigated automatic detection of data access patterns and sharing behavior [75]. Future study is required to explore the integration of these techniques in our system.

Deployment on real wide area networks

My evaluation so far are conducted over simulated wide-area networks. Real world deployment would provide more accurate information on wide area data access patterns and system performance.

Performance study of MPI based applications

My performance evaluation focuses on distributed applications that perform inter-process communication through files. Future study is required to explore system performance for scientific applications that perform inter-process communication through MPI.

MPI stands for Message Passing Interface [43]. It is a library standard used for message passing in a parallel program. As discussed in Section 3, my design assumes that concurrent writes are rare among multiple clients. Based on this assumption, we choose the smallest granularity of a primary server as a single file. However, this assumption is not valid for MPI based applications. Multiple processes of an MPI-based application typically access the same file on different clients simultaneously. When running such an application over my replicated file system, many client requests are forwarded to a remote server that is

elected as the primary server of the file. The problem can be solved by electing a primary server at a byte range granularity. Further investigation is needed to determine the workload characteristics of MPI based applications and to design an efficient primary server election algorithm that suits these patterns.

APPENDICES

APPENDIX A

Proof of Sequential Consistency Guarantee

In databases, one-copy serializability requires that the concurrent execution of transactions on replicated data be equivalent to a serial execution on non-replicated data [36]. In replicated file systems, sequential consistency is comparable to one-copy serializability by viewing each file or directory operation as a transaction. A file operation executes on a single object, the accessed file, but a directory operation may involve more than one object.

In this section, we prove that our replication control protocol guarantees sequential consistency by demonstrating that it obeys the view change properties and rules of El. Abbadi et al. [8], which they show are sufficient conditions for a replicated database to guarantee one-copy serializability.

We introduce essential definitions, properties and rules in Section A.1 and prove correctness in Section A.2.

A.1 Background of View Change Protocol

In this subsection, we introduce the definitions and the theorem proposed by El. Abbadi, et al. [8] for later discussion.

Definitions

In a distributed system that consists of a finite set of processors, a processor's *view* is defined as its estimate of the set of processors with which communication is possible.

Let P be the set of processors, p a member of P, and $\mathcal{P}$(P) the power set of P. The function

$$view : P \rightarrow \mathcal{P}(P)$$

gives the view of each processor p in P.

A *virtual partition* is a set of communicating processors that share a common view and a test for membership in the partition. We assume that at any time, a processor is in at most one virtual partition.

Let V denote the set of all possible virtual partitions. The instantaneous assignment of processors to virtual partitions is given by the partial function

$$vp : P \rightarrow V$$

vp is undefined for p if p is not assigned to any virtual partition.

The total function

$$defview : P \rightarrow true, false$$

characterizes the domain of vp, i.e., defview(p) is true if p is currently assigned to some virtual partition, and is false otherwise.

A function $join(p,v)$ denotes the event where p changes its local state to indicate that it is currently assigned to v. Similarly, function $depart(p,v)$ denotes p changing its local state to indicate that it is no longer assigned to v.

The function

$$members : V \rightarrow P$$

yields for each virtual partition v the set of processors that were at some point in their past assigned to v.

El. Abbadi et al. decompose a replication management algorithm into two parts: a replication control protocol that translates each logical operation into one or more physical operations, and a concurrency control protocol that synchronizes the execution of physical operations. Based on this decomposition, they present three properties and five rules, and prove that any replication control protocol that satisfies these properties and rules guar-

antees one-copy serializability when combined with a concurrency control protocol that ensures conflict-preserving serializability. Below we describe them in turn.

Three Properties

(S1) *View consistency*: If defview(p) & defview(q) and vp(p) = vp(q), then view(p) = view(q).

(S2) *Reflexivity*: If defview(p), then p $\in$ view(p).

(S3) *Serializability of virtual partitions*: For any execution E produced by the replicated data management protocol, the set of virtual partition identifiers occurring in E can be totally ordered by a relation $\ll$ that satisfies the condition: *if v $\ll$ w and p $\in$ (members(v) $\cap$ view(w)), then depart(p, v) happens before join(q, w) for any q $\in$ members(w).* Loosely speaking, this property states that p's departure from v must be visible to all the members of its new cohort in w.

Five Rules

(R1) *Majority rule*: A logical object L is accessible from a processor p assigned to a virtual partition only if a majority of copies of L reside on processors in view(p).

(R2) *Read rule*: Processor p implements the logical read of L by checking if L is accessible from it and, if so, sending a physical read request to any processor q $\in$ (view(p) $\cap$ copy(L)). (If q does not respond, then the physical read can be retried at another processor or the logical read can be aborted.)

(R3) *Write rule*: Processor p implements the logical write of L by checking if L is accessible from it and, if so, sending physical write requests to all processors q $\in$ (view(p) $\cap$ copies(L)) which are accessible and have copies of L. (If any physical write request can not be honored, the logical write is aborted).

(R4) *Execution rule*: All physical operations carried out on behalf of a transaction t must be executed by processors of the same virtual partition v. In this case we say that t executes in v.

(R5) *Partition initialization rule*: Let p be a processor that has joined a new virtual partition v, and let L_p be a copy of a logical object L that is accessible in v. The first operation

on L_p must be either a write of L_p performed on behalf of a transaction executing in v, or a *recover*(L_p) operation that writes into L_p the most recent value of L written by a transaction executing in some virtual partition u such that u $\ll$ v for any legal creation order $\ll$.

CP-Serializability

Two physical operations *conflict* if they operate on the same physical object and at least one of them is a write.

An execution E of a set of transactions T is *conflict-preserving (CP) serializable* if there exists an equivalent serial execution E_S of T that preserves the execution order of conflicting operations.

A concurrency control protocol ensures CP-Serializability if it guarantees that the execution of any set of transactions is conflict-preserving.

In serializability theory studies, serialization graphs are often used to determine whether an execution is CP serializable. The *serialization graph* of an execution E (denoted SG(E)) is a directed graph where each node represents a transaction and an edge from t_i to t_j indicates that these transactions issued conflicting physical operations o_i and o_j on some physical object and that o_i "happens-before" [66] (denoted $<$) o_j. A fundamental theorem of serializability indicates [18, 90, 111]:

Theorem 1 *An execution E is CP-serializable iff SG(E) is acyclic.*

Theorem 1 can be extended to replicated database systems [19].

Let *read(p, l, t)* denotes a physical read of the copy of l at processor p by transaction t, and *write(p, l, t)* denotes a physical write of the copy of l at processor p by transaction t respectively. For some physical copy l_p and transactions t_i and t_j, we say t_j read-l_p-from t_i if write(p, l, t_i) $<$ read(p, l, t_j) and there exists no t_k such that write(p, l, t_i) $<$ write(p, l, t_k) $<$ read(p, l, t_j). For some logical object l and transactions t_i and t_j, we say t_j read-l-from t_i if for some p, t_j read-l_p-from t_i.

Given a directed graph G, the *precedence* relation (denoted $<_G$) is defined by $n_i <_G n_j$ if there is a directed path in G from node n_i to node n_j.

A *one-copy serialization graph* of an execution E (denoted 1SG(E)) is a serialization graph with enough edges added so that [1]

1. if t_i and t_j issue conflicting logical operations, then either $t_i <_{1SG(E)} t_j$ or $t_j <_{1SG(E)} t_i$.
2. if t_i reads-l-from t_j, then there exists no t_k that writes l, such that $t_i <_{1SG(E)} t_k <_{1SG(E)} t_j$.

Theorem 2 *If E has an acyclic one-copy serialization graph, then E is one-copy serializable.*

El. Abbadi, et al. extend Theorem 2 to incorporate virtual partition creation events and partition recovery operations, and prove that [9]

Theorem 3 *Let R be a replication control protocol obeying properties S1-S3 and rules R1-R5, and let C be a concurrency control protocol that ensures CP-Serializability of physical operations. Any execution of transactions produced by R and C is one-copy serializability.*

A.2 Correctness Proof

The serializability theory can be extended to file systems by viewing each file operation as a transaction. It is easy to follow that if a replicated file system guarantees ordered writes, it ensures one-copy serializability, as Figure A.1 shows. If rather, a replicated file system allows conflicting writes, one-copy serializability can be invalidated, as illustrated in Figure A.2.

$$W_0 \longrightarrow W_1 \longrightarrow R_1 \longrightarrow \ldots \longrightarrow R_1 \longrightarrow W_2 \longrightarrow \ldots \longrightarrow W_n \longrightarrow R_n$$

Figure A.1: One-copy serialization graph for ordered writes execution.
In the figure, R_i denotes a read of value i, and W_i denotes a write of value i. The figure shows that given a file, ordered writes guarantee one-copy serializability because we can organize a one-copy serialization graph by inserting read operations of value i between write operations W_i and W_{i+1}.

[1] This definition taken from El. Abbadi, et al.'s tech report [9], is equivalent to the definition given in [19].

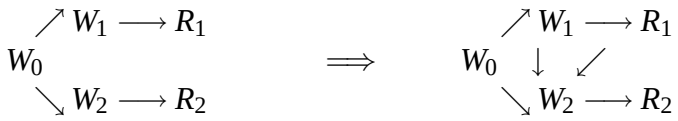
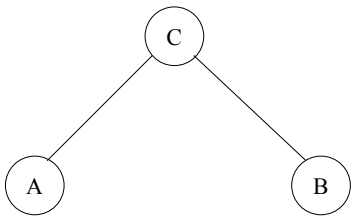


Figure A.2: One-copy serialization graphs with conflicting writes.

This figure shows that given a file, conflicting writes can invalidate one-copy serializability. In the example, R_1 reads the file from W_1 and R_2 reads the file from W_2. Because W_1 and W_2 are conflicting operations, according to the first condition of 1SG, either $W_1 <_{1SG(E)} W_2$ or $W_2 <_{1SG(E)} W_1$. Here we assume $W_1 <_{1SG(E)} W_2$. R_1 and W_2 are also conflicting operations. Since according to the second condition of 1SG, R_1 can not precede W_2, we get $R_1 <_{1SG(E)} W_2$, as shown in the right graph. Then we get into the dilemma with another pair of conflicting operations: R_2 and W_1. Similar to the relation between R_1 and W_2, it has to be $R_2 <_{1SG(E)} W_1$. However, in this way, the 1SG is not acyclic.

Below, instead of using one-copy serialization graph, we make use of Theorem 3 to prove that our protocol guarantees sequential consistency by showing that it satisfies properties S1-S3, rules R1-R5, and CP-Serializability.

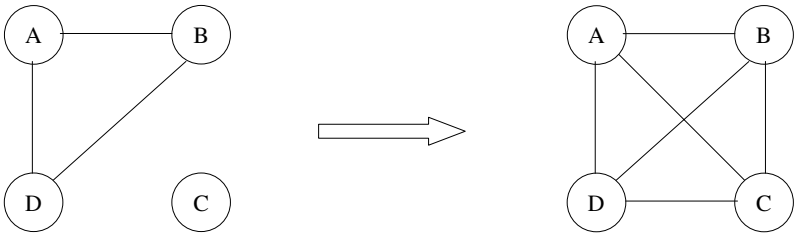
The purpose of Property S1 is to prevent anomalies that may occur when network connections are not transitive. Figure A.3 illustrates such an example. In the example, processors A and B can not communicate with each other. However, both of them are able to communicate with C, and vice versa. If each processor decides its view independently, we may have: view(A) = {A, C}, view(B) = {B, C} and view(C) = {A, B, C}. Consequently, the uniqueness of the majority virtual partition can not be guaranteed among these processors.



	view
A	A, C
B	B, C
C	A, B, C

Figure A.3: An example shows that processors have inconsistency views due to intransitive network connections.

The purpose of Property S3 is to prevent the anomalies as illustrated by the example shown in Figure A.4, where a processor (A) detects changes in the can-communicate relation, adopts a new view, and begins processing operations before another processor (B) in



	Old view	New view
A	A, B, D	A, B, C, D
B	A, B, D	A, B, D
C	C	A, B, C, D
D	A, B, D	A, B, C, D

Figure A.4: An example shows that processors update their views asynchronously upon a partition healing.

the same communication cluster detects the changes. Since B is unaware of C's joining, it will not forward C any updates that it has received. As a result, C might miss those updates processed on B.

S1 and S3 imply that processors can not independently and asynchronously update their view. Our protocol guarantees these properties by having a single server (primary server or replacement server) decide the view of the majority partition. This view is then distributed to and sustained by all the members contained in it. Thus, a unique and consistent majority view is guaranteed.

In our protocol, a replication server may not accept a view that does not include itself. Therefore, Property S2 is ensured.

We observe that a replication server's view always contains a majority of the file replicas in our protocol. [2] So Rule R1 is guaranteed.

Rule R2, the read rule, follows with similar reasoning. Since a file copy l is always accessible on a replication server, that server simply serves the read request with its own copy, as implemented in our protocol.

[2]However, it is possible that a replication server's view is stale.

Rule R3, the write rule, requires a replication server to first check if its view covers a majority of replicas before serving a write request. Our protocol satisfies R3 for the primary server updates its local copy and acknowledges a client write request only after it receives update acknowledgments from a majority of the replication servers.

Rule R4 is automatically satisfied since a file or directory operation is executed atomically in file systems.

Rule R5 says that before copy l_p can be written in a partition v, it must contain the most recent value assigned to l. The rule is satisfied in our protocol by requiring each replica to synchronize with the up-to-date copy before being added to the active view (during replication server recovery) or forming a new view (during primary server failure recovery).

Finally, in our protocol, CP-Serializability is ensured by requiring all write requests for a file to be served on a single server (the primary server).

This completes the proof that our view-based replication control protocol is correct.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] ATLAS. http://atlasinfo.cern.ch/Atlas/Welcome.html.

[2] A backgrounder on IEEE Std 1003.1, 2003 edition. http://www.opengroup.org/austin/papers/backgrounder.html.

[3] CMS. http://cms.cern.ch/.

[4] GridNFS. www.citi.umich.edu/projects/gridnfs/.

[5] LHC. http://atlasinfo.cern.ch/Atlas/Welcome.html.

[6] The Basic Local Alignment Search Tool (BLAST). http://www.ncbi.nlm.nih.gov/BLAST/.

[7] The Educational Global Climate Modeling (EdGCM). http://edgcm.columbia.edu/.

[8] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, 1985.

[9] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. Technical report, IMB Research, San Jose, United States, 1985.

[10] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Operating Systtem Review*, 36(SI):1–14, 2002.

[11] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, Steven Tuecke, and Ian Foster. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *Proceedings of the Eighteenth IEEE Symposium on Mass Storage Systems and Technologies*, 2001.

[12] William Allcock, John Bresnahan, Rajkumar Kettimuthu, and Michael Link. The Globus striped GridFTP framework and server. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

[13] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd international conference on Software engineering*, 1976.

[14] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[15] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, 1991.

[16] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The SDSC storage resource broker. In *Proceedings of CASCON'98*, 1998.

[17] John Bent, Venkateshwaran Venkataramani, Nick LeRoy, Alain Roy, Joseph Stanley, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Flexibility, manageability, and performance in a Grid storage appliance. In *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC-11)*, July 2002.

[18] P. A. BERNSTEIN, D. W. SHIPMAN, and W.S. WONG. Formal aspects of senalizability in database concurrency control, May 1979.

[19] Philip A. Bernstein and Nathan Goodman. The failure and recovery problem for replicated databases. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 114–122, 1983.

[20] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[21] Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[22] Ranjita Bhagwan, Kiran Tati, YuChung Cheng, Stefan Savage, and Geoffrey Voelker. Total recall: System support for automated availability management. In *Proceedings of the First ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[23] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Technical Report 111, Digital Equipment Corp. Systems Research Center, October 1993.

[24] Matt Blaze. NFS tracing by passive network monitoring. In *Proceedings of the USENIX Winter 1992 Technical Conference*, 1992.

[25] Matthew Addison Blaze. *Caching in large-scale distributed file systems*. PhD thesis, Princeton, NJ, USA, 1993.

[26] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. *SIGMETRICS Performance Eval. Review*, 28(1), 2000.

[27] Berthold Butscher and William J. Heinze. A file transfer protocol and implementation. *SIGCOMM Computer Communication Review*, 9(3):2–12, 1979.

[28] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[29] K. Mani Chandy and C.V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, pages 546–556, June 1972.

[30] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 1999.

[31] Jacky Chu, Kevin Labonte, and Brian Neil Levine. Availability and locality measurements of peer-to-peer file systems. In *Proceedings of ITCom: Scalability and Traffic Control in IP Networks*, July 2002.

[32] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[33] Brent Chun and Amit Vahdat. Workload and failure characterization on a large-scale federated testbed. Technical Report IRB-TR-03-040, Intel Research Berkeley, November 2003.

[34] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2006.

[35] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, 1985.

[36] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computer Survey*, 17(3):341–370, 1985.

[37] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[38] Michael Eisler, Alex Chiu, and Lin Ling. RPCSEC_GSS protocol specification. RFC 2203, 1997.

[39] Mike Eisler. LIPKEY - a low infrastructure public key mechanism using SPKM. RFC 2847, 2000.

[40] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[41] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, pages 127–140, 1983.

[42] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[43] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 3-4(8), 1994.

[44] Ian Foster et al. The Grid2003 production Grid: Principles and practice. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, 2004.

[45] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[46] Ian Foster, Jens Voeckler, Michael Wilde, and Yong Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings of the 14th Conference on Scientific and Statistical Database Management*, 2002.

[47] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-G: A computation management agent for multi-institutional Grids. *Cluster Computing*, 5(3):237–246, 2002.

[48] Michael Frumkin and Rob F. Van der Wijngaart. NAS Grid benchmarks: A tool for Grid space exploration. *Cluster Computing*, 5(3):247–255, 2002.

[49] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, 1979.

[50] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Proceedings of the 7th International Conference on Very Large Data Bases*, 1981.

[51] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Readings in database systems (2nd ed.)*, pages 181–208. Morgan Kaufmann Publishers Inc., 1994.

[52] Saikat Guha, Neil Daswani, and Ravi Jain. An experimental study of the Skype peer-to-peer VoIP system. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems*, 2006.

[53] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation*, 2005.

[54] Taliver Heath, Richard P. Martin, and Thu D. Nguyen. Improving cluster availability using workstation validation. *SIGMETRICS Performance Evaluation Review*, 30(1), 2002.

[55] Stephen Hemminger. Netem - emulating real networks in the lab. In *Proceedings of the 2005 Linux Conference Australia (LCA2005)*, April 2005.

[56] Koen Holtman. CMS Data Grid system overview and requirements. The Compact Muon Solenoid (CMS) Experiment Note 2001/037, CERN, Switzerland, 2001.

[57] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer System*, 6(1):51–81, 1988.

[58] Soonwook Hwang and Carl Kesselman. GridWorkflow: A flexible failure handling framework for the Grid. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.

[59] Flavio Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo, and Geoffrey M. Voelker. Surviving Internet catastrophe. In *Proceedings of USENIX Annual Technical Conference*, 2005.

[60] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, 1991.

[61] Steve R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Summer*, pages 238–247, 1986.

[62] John Kohl and B. Clifford Neuman. The kerberos network authentication service (v5). RFC 1510, 1993.

[63] Puneet Kumar and M. Satyanarayanan. Log-based directory resolution in the Coda file system. In *Proceedings of the second international conference on Parallel and distributed information systems*, pages 202–213, 1993.

[64] Puneet Kumar and M. Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. In *Workshop on Workstation Operating Systems*, pages 66–70, 1993.

[65] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[66] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[67] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[68] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[69] Leslie Lamport. Lower bounds on asynchronous consensus. In Hakim Weatherspoon André Schiper, Alex A. Shvartsman and Ben Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 22–23. Springer, 2003.

[70] Leslie Lamport. Fast Paxos. Technical Report MSR-TR-2005-112, Microsoft Research, Mountain View, CA, USA, 2005.

[71] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[72] Hui Li, David Groep, Lex Wolters, and Jeff Templon. Job failure analysis and its implications in a large-scale production Grid. In *Proceedings of the 2nd IEEE International Conference on e-Science and Grid Computing*, 2006.

[73] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, 1991.

[74] Dahlia Malkhi, Florin Oprea, and Lidong Zhou. Omega meets Paxos: Leader election and stability without eventual timely links. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC)*, 2005.

[75] Michael Mesnier, Eno Thereska, Gregory R. Ganger, and Daniel Ellard. File classification in self-* storage systems. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, 2004.

[76] SUN microsystems. NFS: Network file system protocol specification. RFC 1094, 1989.

[77] SUN microsystems. NFS: Network file system version 3 protocol specification. RFC 1813, 1994.

[78] PV Mockapetris. Domain names - concepts and facilities. RFC 1034, 1987.

[79] PV Mockapetris. Domain names - implementation and specification. RFC 1035, 1987.

[80] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. The spread of the Sapphire/Slammer worm. Technical report, CAIDA, 2003.

[81] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, 1986.

[82] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[83] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in tolerating correlated failures. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation*, 2006.

[84] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[85] Daniel Nurmi, John Brevik, and Rich Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. In *Proceedings of 2005 Europar*, 2005.

[86] Adam J. Oliner, Ramendra K. Sahoo, Jose E. Moreira, and Manish Gupta. Performance implications of periodic checkpointing on large-scale cluster systems. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, Workshop 18*, 2005.

[87] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, 1988.

[88] Venkata N. Padmanabhan, Sriram Ramabhadran, and Jitendra Padhye. Client-based characterization and analysis of end-to-end Internet faults. Technical Report MSR-TR-2005-29, Microsoft Research, March 2005.

[89] Jeffrey Pang, James Hendricks, Aditya Akella, Roberto De Prisco, Bruce Maggs, and Srinivasan Seshan. Availability, usage, and deployment characteristics of the domain name system. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[90] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[91] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. NFS version 3: Design and implementation. In *USENIX Summer*, pages 137–152, 1994.

[92] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Thurlow. The NFS version 4 protocol. *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, 2000.

[93] James S. Plank and Wael R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *Proceedings of the The 28th Annual International Symposium on Fault-Tolerant Computing*, 1998.

[94] James S. Plank and Michael G. Thomason. The average availability of parallel checkpointing systems and its importance in selecting runtime parameters. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, 1999.

[95] Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann. Replication in Ficus distributed file systems. In *Proceedings of the Workshop on Management of Replicated Data*, November 1990.

[96] Jon Postel. User datagram protocol. RFC 768, 1980.

[97] David Patrick Reed. Naming and synchronization in a decentralized computer system. Technical Report 205, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[98] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The Oceanstore prototype. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2003.

[99] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of the USENIX 2002 Technical Conference*, 2002.

[100] Ramendra K. Sahoo, Anand Sivasubramaniam, Mark S. Squillante, and Yanyong Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, 2004.

[101] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. *SIGOPS Operating System Review*, 36(SI):15–30, 2002.

[102] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, 1985.

[103] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of 2002 Multimedia Computing and Networking*, 2002.

[104] M. Satyanarayanan. A survey of distributed file systems. Technical Report CMU-CS-89-116, Pittsburgh, Pennsylvania, 1989.

[105] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. The ITC distributed file system: principles and design. *SIGOPS Operating System Review*, 19(5):35–50, 1985.

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

[107] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *Computer Systems*, 2(2):145–154, 1984.

[108] 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'06)*, 2006.

[109] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Michael Eisler, and David Noveck. Network file system (NFS) version 4 protocol. RFC 3530, 2003.

[110] Bob Sidebotham. Volumes – the Andrew file system data structuring primitive. In *Proceedings of European UNIX Systems User Group Autumn'86*, 1986.

[111] R. E. Stearn, P.M. II Lewis, and D.J. Rosenkrantz. Concurrency controls for database systems, October 1976.

[112] Jeremy Stribling. Planetlab all-pairs ping. `http://infospect.planet-lab.org/pings`.

[113] Douglas Thain, John Bent, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Pipeline and batch sharing in Grid workloads. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.

[114] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

[115] Khanh Tran and Rhonda Rundle. Hackers attack major Internet sites, cutting off Amazon, Buy.com, eBay. *The Wall Street Journal*, A(3), February 2000.

[116] Nitin H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers*, 46(8), 1997.

[117] Hakim Weatherspoon, Tal Moscovitz, and John Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, 2002.

[118] Brian S. White, Michael Walker, Marty Humphrey, and Andrew S. Grimshaw. Legionfs: a secure and scalable file system supporting cross-domain high-performance applications. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, 2001.

[119] Kenneth F. Wong and Mark A. Franklin. Distributed computing systems and checkpointing. In *Proceedings of the 2nd IEEE Symposium on High Performance Distributed Computing HPDC'93*, 1993.

[120] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Networked Windows NT system field failure data analysis. In *Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing*, 1999.

[121] Praveen Yalagandula, Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems. In *Proceedings of the First Workshop On Real Large Distributed Systems*, 2004.

[122] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Volume leases for consistency in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):563–576, 1999.

[123] Tatu Ylonen. SSH - secure login connection over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, 1996.

[124] Edward R. Zayas and Craig F. Everhart. Design and specification of the cellular Andrew environment. Technical Report CMU-ITC-070, June 1988.