May 6, 2005

To the Graduate School:

This dissertation entitled "Achieving Scalability in Parallel File Systems" and written by Philip H. Carns is presented to the Graduate School of Clemson University. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy with a major in Computer Engineering.

Walter B. Ligon, III, Advisor

We have reviewed this dissertation
and recommend its acceptance:

Adam Hoover

Tarek Taha

Jim Martin

Robert Ross

Accepted for the Graduate School:

ACHIEVING SCALABILITY IN PARALLEL FILE SYSTEMS

A Dissertation

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

Computer Engineering

---

by

Philip H. Carns

May 2005

Advisor: Dr. Walter B. Ligon, III

ABSTRACT

Parallel computing has become an essential tool for scientific computation. However, several supporting technologies beyond just raw processing speed are necessary in order to achieve balanced application efficiency in this domain. Parallel file systems in particular are an example of a supporting technology that has proven successful in achieving the I/O bandwidth demanded by parallel applications. However, the need for high performance continues to grow, prompting efforts to scale parallel computers to ever larger sizes in order to meet the computational demand. The current generation of large scale systems utilize thousands of dedicated processing nodes while even larger systems are planned for the near future. Conventional file system design assumptions are not sufficient for this class of parallel systems. We must therefore revisit parallel file system design techniques in order to achieve the scalability necessary for the the next generation of parallel computers.

We have identified five key obstacles that limit the ability of parallel file systems to scale to systems with thousands of processors: efficiency, complexity, management, consistency, and fault tolerance. In order to address these obstacles we present the techniques of intelligent servers and collective communication for parallel I/O. These techniques are used to offload work from client processes, optimize high level file system operations, and limit the overhead of network communication in order to provide a comprehensive framework for building scalable file systems. These techniques not only improve file system scalability, but also help to broaden the applicability of parallel file systems to problem domains beyond scientific computing. Intelligent servers are an original concept in which servers transparently take control of optimization decisions and communicate with each other in order to service individual operations. Collective communication is a well known optimization in the

fields of message passing and distributed shared memory which we have applied in a novel manner to the parallel file server environment.

In this work we present the Parallel Virtual File System 2 (PVFS2), along with several key extensions, as an experimental platform for this study. We then develop an analytical modeling framework for comparing a variety of file system algorithms in order to predict file system performance at scale and compare potential optimizations. These models are verified against a real world implementation with hundreds of processors and multiple network environments. Next we evaluate the implementation of intelligent servers and collective communication in PVFS2 with regard to the five previously listed obstacles to scalability. We show that throughput for metadata operations can be doubled for moderately sized systems and project an order of magnitude improvement for systems with thousands of servers. We simultaneously reduce client code complexity and decrease CPU overhead by 90%. We show that management is improved through intelligent server load balancing and performance monitoring. We also evaluate consistency improvements with case study analysis and demonstrate improved fault tolerance when compared to conventional design alternatives. This study concludes with a summary of how the research goals have been met and how previously intractable avenues of future work have been enabled.

DEDICATION

I dedicate this dissertation to all of my family and friends who have helped me through graduate school. Most importantly, thank you to my wonderful fiancé Christina for her patience and support, and thank you to my mother for always being there for me.

ACKNOWLEDGMENTS

I would like to acknowledge Dr. Walter Ligon III, Dr. Robert Ross, and all of the PARL Laboratory for their guidance and advice in the work presented in this dissertation. We also gratefully acknowledge use of "Jazz," a 350-node computing cluster operated by the Mathematics and Computer Science Division at Argonne National Laboratory as part of its Laboratory Computing Resource Center.

TABLE OF CONTENTS

Table of Contents (Continued)

Page

Table of Contents (Continued)

Page

LIST OF FIGURES

List of Figures (Continued)

List of Figures (Continued)

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Scientific Computing with Parallel Computers

The need for high performance computing has grown significantly in recent years. This is particularly true in the field of scientific computing. Scientific simulations can often be made more accurate through the use of more complex models or by operating on larger data sets. However, these improvements come at the cost of computational complexity. Performance improvements are therefore critical to computing more accurate results and obtaining them in a timely manner. Example problem domains include astrophysics, weather modeling, remote sensing, fluid dynamics, and bioinformatics.

A popular approach for achieving high performance for these application domains is to use parallel computers. Parallel computing overcomes the bottleneck associated with a single CPU by distributing computation across a collection of CPUs which operate concurrently. Parallel computers originally consisted of special purpose machines with custom hardware and software. However, recent years have seen the advent of Beowulf cluster computing, which takes advantage of commodity processors, networks, and software to create a parallel computer out of inexpensive components [77]. While Beowulf clusters have not completely replaced custom parallel computers, they have succeeded in making parallel computing technology available to a much wider audience.

1.1.1 Parallel I/O

The performance of commodity processors has been increasing at a tremendous rate [21]. Unfortunately, other system components such as main memory, secondary

storage, and networking have not improved at the same rate [34]. This has led to a situation in which CPU performance is not always the limiting factor for many scientific applications. In particular, many scientific applications must process enormous volumes of data, and therefore are limited primarily by I/O throughput.

Parallel file systems are commonly used to help overcome the I/O bottleneck for scientific applications. Parallel file systems distribute data across multiple processing nodes, each with its own storage resources. If many distributed processes access the same file system concurrently, then the load is spread across several servers rather than focusing the I/O on a single server. This distribution of resources not only allows the file system to leverage multiple independent storage devices, but also makes more effective use of the bisection bandwidth of the interconnection network. In other words, the file system throughput is not constrained by the maximum bandwidth of any single network link. The file system instead makes use of the aggregate sum of many separate network links. Parallel file systems are thus capable of obtaining relatively high throughput when compared to traditional network storage approaches. Parallel file systems have been used in scientific computing since the 1980's [26]. However, parallel I/O still presents many open avenues of research. Examples include fault tolerance, domain specific optimizations, grid awareness, and scalability.

It is also important to note that parallel file systems can be constructed using commercial off-the-shelf components (COTS). For example, the storage resources may consist of standard IDE or SCSI disk drives, while the transport between nodes may be provided by the same high performance local network that is used for interprocess communication. This is cost effective, but presents challenges at the software level to leverage these resources efficiently.

1.2 Scalability

The need for high performance computing has led to the construction of increasingly large parallel systems. The 2004 list of the top 500 supercomputing sites in the world shows that the top 25 positions are all held by machines with greater than 1000 processors [27]. Although several of these machines are Linux clusters, research into other architectures is far from dead. Parallel computers on the horizon include the ASCI Red Storm machine [47] with approximately 10,000 processors and the IBM Blue Gene/L [39] with approximately 65,000 processors. The software infrastructure must scale well in order to utilize systems such as these effectively. Some software components, such as interprocess communication libraries, have been the focus of in-depth scalability research. File systems, on the other hand, still present an open problem. Today's parallel file systems simply do not provide the level of scalability necessary to support thousands or tens of thousands of concurrent application processes. This lack of scalability can be attributed to a variety of factors as outlined in the next section.

1.2.1 Scalability Challenges in Parallel I/O

There are a number of obstacles to obtaining scalability in a parallel file system. Five particular problems are efficiency, consistency, complexity, management, and fault tolerance. We call attention to these specific problems because their prominence increases with the scale of a given parallel file system deployment, and therefore eventually limit the usefulness of file systems beyond certain sizes. These five issues are outlined in further detail below.

It is important to note that scalability is not just a concern for bulk read and write operations. These operations have been studied in-depth, and several high level libraries attempt to enhance scalability of these operations by introducing optimizations on top of the underlying file system. However, the overhead of small

scale control and management operations also becomes a barrier to scalability in large systems. Examples of such operations include file creation, performance monitoring, retrieval of file attributes, or the setup of bulk transfers. It is difficult to implement generic optimizations for this type of functionality in a high level library, because each of them is dependent upon the specific file system implementation. Operations such as these provide clear motivation for fundamental changes in file system level algorithms.

Efficiency: Adding more clients, servers, or storage resources to a parallel file system tends to eventually introduce more overhead in accessing these resources. This can lead to situations in which the file system overhead begins to marginalize the benefit of adding additional resources. It is important that we mitigate this effect in order to ensure that file systems continues to make efficient use of the underlying hardware, even in large clusters. We define efficiency in parallel I/O as the ratio of sustained measurable throughput to the peak theoretical throughput. This applies not only to I/O operations such as reading and writing but also to file system control operations.

Consistency: In the parallel file system context we define consistency as the degree of uniformity of the file system state that is visible across multiple servers or clients. As file systems (and the number of clients accessing them) grow larger, it becomes increasingly difficult to maintain consistent global state. This impacts several aspects of the file system, including caching, name space updates, recovery from errors in distributed operations, and resource allocation. The servers must be consistent with respect to each other, and they must provide coherent and correct information to each client. The introduction of more processes which may alter the state of the file system inherently leads to more opportunities for skew in the state. However, this skew must be constrained within the limits defined by the file system semantics.

Management: Management is an overlooked aspect of many file systems, but it is becoming more important. Particular areas of interest include performance tuning, health monitoring, and resource management. Health monitoring includes tasks such as identifying hardware failures before they disrupt the file system, or locating corrupted data. Resource management includes tasks such as adding or removing servers from the system or making sure that heterogeneous resources are leveraged properly. These operations must be simple and robust in order for a large scale file system to actually be useful in a production capacity and not just as a research endeavor.

Complexity: Parallel file systems are inherently complex due to the scope of the services that they must provide. This complexity is exacerbated in file systems which support extremely large scale computers, due to the difficulty of maintaining consistency and coordinating resources. Care must be taken to prevent this complexity from becoming a maintenance burden or an impediment to robust operation. In addition, client side complexity may introduce CPU overhead that has a significant impact on the performance of applications that access the file system.

Fault tolerance The aggregate mean time between failure (MTBF) for components such as disks and and processing nodes will decrease as more commodity hardware is added to the system. Large scale file systems must therefore acknowledge this and provide graceful recovery from errors. This design criterion must be a pervasive feature of the system architecture from the beginning. It is much more difficult to add fault tolerance to existing software [46].

1.3 Expanded Problem Domains

The problems outlined in the preceding section not only limit file system scalability, but also have an impact on the applicability of parallel file systems to problem domains beyond parallel scientific computing. Two notable problem domains include wide area

or grid environments, and Internet services such as web, mail, or electronic commerce. Both of these domains could benefit from parallel file systems because of their storage and performance requirements.

Wide area usage examples include situations in which a parallel file system resides at one geographic location while the application that accesses it resides at another. The unavoidable increase in network latency and and fault probability inherent in this environment serves to exacerbate the efficiency, consistency, and fault tolerance issues in parallel file system design.

Common Internet services tend to generate workloads that emphasize small disk accesses and metadata performance rather than streaming I/O bandwidth. This poses a challenge for parallel file systems such as PVFS2 that have been tailored to suit large scientific applications. The efficiency of latency bound operations becomes the dominant factor in performance for Internet services. These services also demand extremely robust fault tolerance for production deployment.

1.4 Proposed Research and Goals

Despite the fact that many of the challenges outlined in section 1.2.1 have arisen frequently in research (often under different problem domains), no current file systems adequately address all of them. We must examine new techniques to help overcome these challenges and make effective use of the high performance storage potential of large scale clusters. **We propose that two complimentary concepts, *intelligent servers* and *collective operations*, can be applied within the framework of parallel I/O to enable file systems to scale effectively to next generation systems with thousands of processors.**

Intelligent servers and collective operations do not guarantee file system scalability by themselves, but are critical to make sure that scalability is achievable. Existing file systems can be augmented to use these concepts in a way that intrin-

sically alters and enhances their behavior in large cluster environments. In addition we may be able to expand the applicability of parallel file systems to new problem domains. The concepts of intelligent servers and collective operations are presented in further detail in sections 1.4.1 and 1.4.2, respectively.

1.4.1 Intelligent Cooperative Storage Servers

We define a server as a file system component that controls access to storage devices, while we define a client as any component that wishes to access the devices, such as an application library or a kernel driver. The storage devices may be as basic as individual IDE disk drives, but other configurations such as RAID arrays and Storage Area Networks are possible back-ends as well. Traditional file system servers are relatively simplistic, in that they only act in response to client requests, and tend to treat each request as an independent operation acting upon local resources. The alternative that we propose is to make each server both intelligent and proactive. In particular, servers can be given the ability to communicate with each other and collaborate on aggregate operations. This approach offers several advantages.

From a performance optimization point of view, intelligent servers can make more sophisticated optimizations by taking into account global knowledge of the file system rather than just local server parameters. Intelligent servers can also achieve better resource utilization by redistributing tasks to the servers that are most suited for carrying out those task. Finally, the reduction of the client's role in carrying out operations will result in less CPU overhead for end-user applications that access the file system.

Client complexity could be reduced by offloading operations that consist of multiple steps to remote servers. Consider a general case parallel operation in which a client typically contacts multiple servers to carry out an aggregate operation. In a system with intelligent cooperative servers, this would instead result in a single

request to a single server which then coordinates with other servers in the file system to decide how to best carry out the aggregate operation. From the client's point of view, the operation has become a simple single step process, with little or no cleanup necessary if a failure occurs. This has significant implications for fault tolerance. Client libraries are more prone to faults than servers are, due to the fact that they are linked against unpredictable end-user applications. Dedicated server machines are also more likely to receive the benefit of fault tolerant hardware than clients are in large systems. It is therefore more logical to place the responsibility for recovering from a fault during an intermediate step of an operation in the hands of a server process.

Intelligent servers can also offer better control over consistency. Servers possess awareness of aggregate state changes that are transparent to the client and are in a better position to coordinate atomicity. One conventional technique for controlling consistency in a parallel file system is the use of distributed locking. In this model, a client must first acquire a lock for a given resource, perform some operation, and then release the lock. The overhead in distributed locking of this nature becomes a problem as contention increases in large systems. It also introduces design complexity in planning how to recover from failures while locks are held. However, an intelligent server may decide to serialize access to its own local resources while an operation is being performed without notifying other servers or clients. For example, it may lock a local directory entry while requesting that another server update the remote object that the entry refers to. There still must be a timeout mechanism to handle failure conditions, but there is no overhead in cleaning up remote lock resources in these conditions because the lock is not shared.

Finally, intelligent cooperative servers could better implement autonomous activity such as redistributing data, monitoring health, or verifying consistency. These

types of operations could be performed in the background while the system is idle and do not necessarily require client or user interaction.

The comprehensive concept of intelligent servers which offload tasks from clients, make optimization decisions, and automatically collaborate to service aggregate operations is unique in parallel file system design. Intelligent servers to some extent address all five distinct challenges to parallel I/O scalability listed in section 1.2.1:

- efficiency: by making better optimization and load balancing decisions
- management: by providing a mechanism for servers to independently maintain health and performance statistics
- consistency: through better coordination and synchronization of complex operations
- complexity: by simplifying client library implementations
- fault tolerance: by orchestrating operations from a point where faults can be handled more gracefully

1.4.2 File System Level Collective Communication

The performance of many common parallel file system operations is bound by the overhead involved in communicating control messages to large numbers of remote nodes. This problem persists even in high performance networks, due to message startup overhead and the difficulty in managing many concurrent network operations. Message passing libraries and shared memory implementations face similar difficulties. Luckily, those fields benefit from a large body of research on how to mitigate this bottleneck. Some of this work has even been applied to high level I/O libraries that operate on top of parallel file systems. However, we believe that some of these

techniques can be equally applicable within the file system itself where they can address problems unique to that environment.

Parallel communication libraries use *collective communication* methods to reduce the cost of aggregate messaging between many network hosts. The key principle of collective communication is to optimize complex network patterns by describing them from a high level rather than focusing on individual operations. A high level view offers insight into how to impose structure on the communication such that the aggregate operation becomes less expensive. For example, if a single machine wishes to send an identical message to every host on a large network, it may be more efficient to utilize a "tree" pattern in which each host relays the message to a subset of the remaining hosts, rather than forcing a single source machine to send each packet sequentially.

We can apply many of the same techniques to communication within a parallel file system; however, several key differences in the parallel file system environment must be addressed. First of all, hosts in the file system are divided into two groups: clients and servers. These two groups may differ significantly in ability, resources, and awareness of the file system as a whole. Secondly, servers must handle unexpected messages, unlike typical message passing scenarios in which every send is matched by a receive. Unexpected messages occur naturally in a file system because servers have no advance knowledge of when or how the file system will be accessed by clients. This is in contrast to message passing systems in which collectives are triggered by blocking library calls which allow each process explicit time to prepare for the operation. Finally, fault tolerance expectations are somewhat different between message passing libraries and file systems. Faults in a message passing library only impact the execution of a single application, while faults in a parallel file system may interrupt a system wide service used by many applications.

Collective communication itself is not a new concept. However, the application of collective communication algorithms within a parallel file system itself is a novel contribution, particularly when considering the unique problems posed by the environment. Successful integration of collective communication at the file system level will address three of the five distinct challenges to parallel I/O scalability that were listed in section 1.2.1:

- efficiency: by reducing the network overhead associated with critical file system operations
- management: by providing a convenient mechanism for gathering global health and performance statistics
- consistency: by serving as a building block for implementing synchronization primitives

Note that the use of collective communication does not explicitly address the problem of fault tolerance. However, any part of the communication infrastructure in a file system must collaborate with other facilities such as the client API and storage abstraction to make sure that fault tolerance is achievable. In particular, the fundamental algorithms employed in the collective operations must intrinsically accommodate fault tolerance.

1.4.3 Methodology

The Parallel Virtual File System 2, developed at Clemson University and Argonne National Laboratory, will be used as the test platform for this research [62]. PVFS2 is a modern parallel file system that targets scalability and high performance, and lends itself to the proposed research by virtue of its extensible modular design. We will extend PVFS2 to provide the features necessary for intelligent server and collective communication enhancement.

In addition to implementation, we will also develop an analytical framework for comparing file system algorithms both with and without intelligent servers and collective communication. These models will serve to predict file system performance at scale and allow us to evaluate optimization possibilities prior to implementation.

Several existing file system operations will be re-implemented using intelligent servers and collective communication enhancements. We will then use these implementations to evaluate how well the key obstacles to file system scalability have been addressed and project file system performance on systems with thousands of file servers.

1.5 Outline

The remainder of this dissertation will be outlined as follows. In chapter 2 we will discuss related work. In chapter 3, PVFS2 will be introduced as the basis for this research, including key contributions from the implementation of the file system itself. Chapter 4 will present extensions to PVFS2 that provide intelligent server and collective communication functionality. Chapter 5 will outline an analytical modeling framework for projecting file system performance and comparing potential optimizations. The models will be validated using raw performance results from real world file systems. Chapter 6 evaluates the implementation of intelligent servers and collective communication in terms of the key obstacles to scalability. Finally, chapter 7 concludes by summarizing the overall results of the study and pointing out future avenues of research.

CHAPTER 2
RELATED WORK

The research presented in this document builds on previous work from many fields, including but not limited to: collective communication, performance evaluation of collective communication, high level I/O libraries, aggregate I/O optimizations, disk performance modeling, and fault detection. We will summarize important research from those fields that has been leveraged in this study, with particular emphasis on collective communication algorithms.

2.1 Collective Communication

Many research projects have investigated the impact of specific collective communication techniques on applications. Early work tended to focus on particular communication patterns or machine dependent optimizations [49, 69, 57]. Most of these efforts were geared towards use in message passing libraries, while others were used to implement distributed shared memory [59, 16].

Two trends in the 1990's led to a slight shift in focus in collective communication research. The first is that as parallel computing (the message passing paradigm in particular) has matured, it has become easier to determine the most important and commonly used collective operations. There is now a large body of applications in many problem domains that share similar communication characteristics. The second important trend is the tendency to use a common API for message passing on many different platforms. This effort has been led by the MPI specification [56] which is now almost universally adopted [32]. Prior to the advent of MPI, most vendors utilized their own unique messaging libraries. These vendor specific libraries were typically tailored to specific commercial hardware products.

Current research efforts leverage the fact that the most common collective operations are now well understood from an application perspective and that a common API exists for accessing them on many architectures. The focus has therefore shifted away from special case optimizations. Newer efforts have instead focused on portability, and on providing a complete family of collective operations based on common primitives.

The Interprocessor Collective Communication Library (InterCom) [5] is one example of a project that brings together many common portable optimizations into one comprehensive study. Although InterCom does not directly implement any portion of the MPI specification, it is built on the assumption that it could be used as a building block for an MPI implementation. InterCom identifies six core collective operations, described below assuming that P hosts are involved:

- broadcast: sending an exact copy of data from one host to P hosts
- scatter: dividing data into $1/P$ segments and distributing it among P hosts
- gather: the opposite of scatter; data is collected from P hosts and reassembled at one host
- collect: the same as gather, except that a copy of the data in its entirety is provided to all hosts rather than just one
- combine-to-one (or reduce): similar to a gather, except some computation (such as summation) is applied to the data as it is gathered
- combine-to-all: similar to combine-to-one, except that the a copy of the result is given to all P hosts rather than just one
- distributed combine: similar to combine-to-all, except that the result is divided and scattered among all P hosts

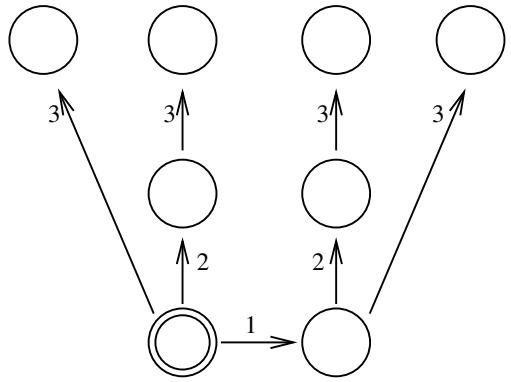
The InterCom project recognized that there are underlying primitives common to many of these operations, but that the optimal primitive may change depending on the size of the data to transfer. InterCom therefore advocated a hybrid design which used several building blocks combined in different ways depending on the message size. It also recognized that the algorithms must work for subsets of nodes in order to be generally applicable. Most of the algorithms assumed the use of a mesh or hypercube network because those were the dominant network topologies for parallel computers of the time.

The most recent collective communication research efforts have revisited this topic for modern networks. Thakur and Gropp of the MPICH team at Argonne National Laboratory have investigated collective communication optimizations for clusters of machines connected by a switch [79]. This architecture is prevalent today, with examples including the IBM SP2 [1] series of machines as well as Linux clusters connected by Myrinet [7] or Ethernet networks. This is particularly relevant to parallel file systems such as PVFS2 which target the same class of parallel computer. We will therefore use the algorithms outlined in their study as a basis for collective communications used in this dissertation.

Thakur and Gropp's work is a survey of collective algorithms taken from literature but adapted to work more optimally in a switched network. In many cases it was discovered that the optimal algorithm may vary depending on message size. In particular, at small message sizes the algorithms must minimize latency while at large message sizes the algorithms much minimize bandwidth utilization.

The algorithms are implemented on top of point-to-point primitives. A cost model for these points-to-point primitives can be parameterized by the startup time per message (α), the transfer time per byte (β), and the number of bytes transfered (n), resulting in the following equation:

Figure 2.1: Binary tree collective communication



$$T_{p2p} = \alpha + n\beta \tag{2.1}$$

On switched networks there is no need to take the route or number of links into account, since all network connections between hosts are assumed to be equal.

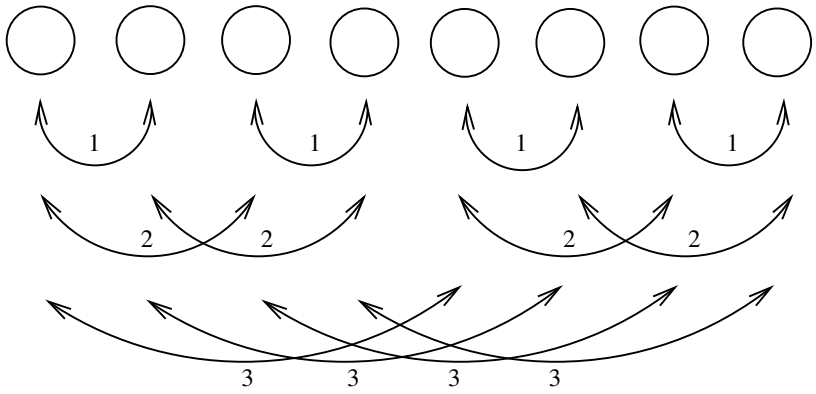
A broadcast operation for small messages can be implemented as a binary tree on top of point-to-point primitives. An example is provided in figure 2.1. The arrows represent communication between hosts, and are labeled with step numbers. In the first step, the root process sends data to one other process. Each of these then act as new subtree root and continues the algorithm recursively.

For p processes, the total cost can be modeled as:

$$T_{tree} = \lceil \log_2(p) \rceil (\alpha + n\beta) \tag{2.2}$$

For larger messages, the broadcast is more efficiently implemented as a scatter operation (where the data is divided into small parts and broadcast) followed by a collect operation that assembles the data at each host. This algorithm was first proposed by

Figure 2.2: Recursive doubling collective communication



3

Van de Geijn et al. *intercom*, and its cost is given as:

$$T_{vandegeijn} = (\log_2(p) + p - 1)\alpha + 2(\frac{p-1}{p})n\beta \tag{2.3}$$

A reduce operation for small messages is best performed with a binary tree, essentially the same as the one shown in figure 2.1 except that communication flowing in the opposite direction. The only difference in the model is that an extra factor, γ, is used to model the computation cost per byte of data, assuming that we will be modifying the data as it is reduced:

$$T_{tree} = \lceil \log_2(p) \rceil (\alpha + n\beta + n\gamma) \tag{2.4}$$

More efficient reduction algorithms exist for large message operations. However, they are difficult to manage for complex reduction computations that may result in asymmetric data.

The other network pattern of interest in parallel file systems is the combine-to-all (also known as all-to-all) pattern. In this case we wish to combine data from each of the processes and leave the result at all processes. For this pattern we will

use the recursive doubling algorithm as described by Rolf Rabenseifner [68]. Figure 2.2 illustrates and example of the communication pattern for eight processes. In the first step, peers that are distance 1 apart transmit data to each other. In the second step, peers that are distance 2 apart transmit their combined data to each other. This continues recursively until each processes has a copy of all data. Additional communication may be required at some steps to accommodate set sizes that are not a power of two. The recursive doubling algorithm takes advantage of the full duplex capability of most modern networks.

If the number of steps and amount of data sent in each step is taken into account, then the cost of a recursive doubling algorithm can be modeled as follows:

$$T_{rec} = \log_2(p)\alpha + \frac{p-1}{p} n\beta \tag{2.5}$$

2.2 Performance Evaluation of Collective Communication

Several papers have been published recently concerning techniques for measuring the performance of collective communications [74, 85]. Collective communication performance is difficult to quantify due to the complexity of accounting for time skew across a parallel computer. In addition, the quality of a collective communication implementation is best judged relative to the performance of the underlying network.

2.3 High Level I/O Libraries

High level I/O libraries serve three primary purposes. First of all, they provide an abstraction layer that allows application writers to create portable applications that will work with a variety of file system interfaces. Secondly, libraries can be used to provide domain specific APIs, with functionality that is tailored to the access patterns common to a certain class of applications. Finally, and perhaps most importantly, high level libraries can be used to implement optimizations portably without modi-

fying the underlying file system. Examples of high level parallel I/O libraries include MPI-IO [81, 20], HDF5 [36], and PNetCDF [50].

Libraries such as these can often complement optimizations that are implemented within the file system itself. Latham, Ross, and Thakur have studied how design choices in parallel file system API's impact the scalability of MPI-IO [48]. This is distinct and complimentary to the file system internal optimizations that will be presented in this dissertation, which are implemented mostly on the server side of the file system and are transparent to the library.

2.4 Aggregate I/O Optimizations

Aggregate I/O optimizations are optimizations, implemented in the file system or a high level library, that take advantage of information gathered by collectively submitted client I/O requests. These collective I/O operations provide a higher level view of the aggregate operation rather than servicing each client's request independently. This can lead to several improvements in how the file system data is accessed and distributed. Two popular optimizations of this class include two phase I/O [8, 25] and data sieving [78].

2.5 Disk Performance Modeling

Disk performance is clearly an important factor in the behavior of a parallel file system. Unfortunately, modeling disk behavior can be a complex task due to caching and queuing effects at the disk, controller, operating system, and file system level. However, various parts of this path can be analyzed independently to arrive at some baseline models. This requires background in modern hard disk architecture that is readily available in many texts. For the purpose of this discussion, however, we can assume that these specifications are known values that can be measured directly or determined from the disk manufacturer. Hennessy and Patterson [22] provide the

following equation for disk access time assuming no queuing delay and no cache:

$$T_{access} = T_{seek} + \frac{.5}{R} + \frac{size}{B} + T_{c\_overhead} \tag{2.6}$$

where T_{seek} is the average seek time, R is the rotational speed, and B is the transfer rate, and $T_{c\_overhead}$ is the controller overhead. Hennessy and Patterson also observe that the real world seek time is typically only 25% to 30% of the manufacturer's advertised value, due to disk locality. We will account for this in later models by including a F_{seek_local} as the first coefficient.

Other groups such as Ruemmler and Wilkes at Hewlett-Packard have investigated how to model disk accesses using in-depth simulation [70]. These simulations take into account more complete performance metrics as well as caching effects and locality. However, they do not result in analytical equations for performance.

Later works have expanded on these simulation techniques to develop thorough analytical models for disk drives [73, 72]. Modern models such as these include factors such as read ahead caching and request reordering.

2.6 Quorum and Heartbeat Systems

In recent years there has been renewed interest in software level redundancy in parallel file systems. As file systems become larger, it becomes less practical to rely on a per-server failover scheme. Furthermore, the size of data sets in many fields such as remote sensing [24] or bioinformatics [23] have made it more desirable to utilize parallel file systems as long term storage.

Implementing distributed redundancy is not a straight forward task, however. In this section we will focus on two particular problems. The first issue is how to accurately detect when a server has failed. The second is how to reach a consensus when there is a conflict of data between servers. Fortunately, both problems are well

known in the distributed database community and have been the subject of years of research. Heartbeat systems and quorums have gained prominence as solutions to each of the respective problems.

The fundamental idea of a heartbeat system is that hosts will periodically send status messages, or heartbeats, to their peers. If a status message has not been received within some timeout period, then it can be assumed that either the host or the network link connecting it have failed. However, naive use of this approach can lead to systems that do not scale well as the number of hosts is increased. Gupta, Chandra, and Goldszmidt have more recently tried to address this by focusing on the scalability and efficiency of failure detectors [33]. In their work, they clearly separate the goals of *completeness* (the requirement that every group member eventually detect the failure) and *efficiency* (the speed and accuracy of the algorithm). They then define algorithms that offer trade-offs in either category depending on application requirements.

Aguilera, Chen, and Toueg have also extended the concept of heartbeats, with a particular emphasis on implementation practicality [2] [3]. They have addressed two issues of particular interest: how to implement heartbeat systems that do not rely on timeouts, and how to achieve reliable consensus on a partitioned network. Partitioned networks are networks in which two hosts may have different views of which nodes are reachable (i.e. only some paths are down). This is of significant concern in real world systems.

Other groups have evaluated failure detectors in the context of real time systems [35]. Their focus is on analyzing trade-offs between neighbor detection and end-to-end detection, with a focus on quantifying the maximum latency.

The second key problem in distributed redundancy is how to resolve conflicts between copies of data, even if all servers and network links are working properly. Thomas and Gifford pioneered research in this area in the 1970's by defining quorum

algorithms in which servers vote to decide how to resolve conflicts [83] [30]. A quorum is simply any subset of servers who's agreement is sufficient to validate a distributed operation, such as to decide what order to commit operations in, or to decide which replica to use for a given operation. It may not be necessary for all servers to agree in order to reach a valid decision. There are also many variations on these algorithms to include ideas such as weighted voting and sequential read/write consistency.

More recent work in this area has extended the concept of quorums to handle Byzantine faults [4] [54]. Byzantine faults include elaborate scenarios such as intermittent failure, or failures that result in the injection of undetected erroneous data into the system. These scenarios must be handled by more sophisticated quorum systems in order to account for out of date or incorrect reporting from servers which otherwise appear to be operating correctly.

CHAPTER 3
PARALLEL VIRTUAL FILE SYSTEM 2

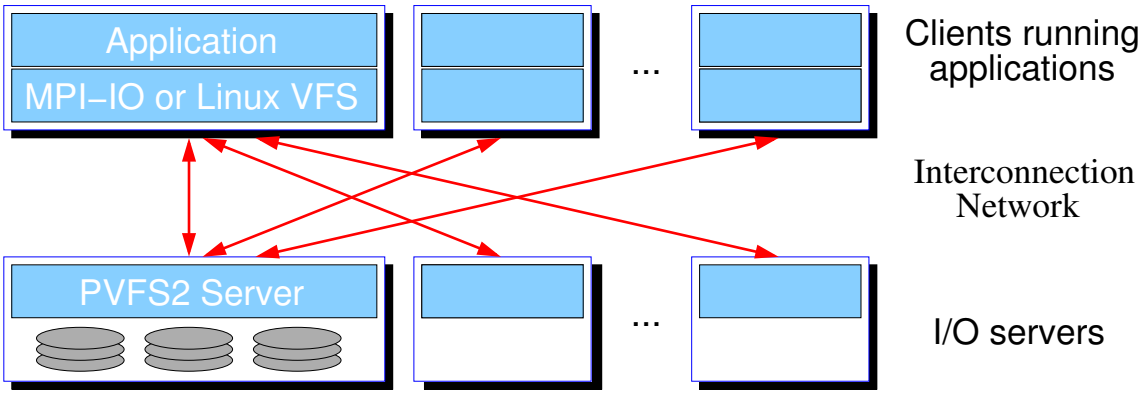
3.1 Overview

The Parallel Virtual File System 2 (PVFS2) is a parallel file system designed for Linux clusters, though it is portable to other architectures as well. PVFS2's primary design goal is to provide scalable, high performance access to data for parallel scientific computing applications. It incorporates optimizations and features that may make it useful for other workloads as well, as long as those optimizations do not compromise behavior for scientific parallel applications. PVFS2 aims to serve as both a vehicle for I/O research and a production level tool for use by the high performance computing community.

PVFS2 is the successor to the original PVFS [14] file system developed at Clemson University. The PVFS project was started in the mid 1990's, and has since grown to become a standard tool for the scientific computing community and has fostered a wide array of related research. PVFS2 is not a direct evolution, however; it is a completely independent design and implementation. It addresses several inherent limitations of the original project, and incorporates research findings and design principles learned from first hand experimentation.

Three motivating concepts stand out in particular when comparing PVFS and PVFS2. First of all, PVFS2 embraces modularity in almost all aspects of the architecture. This allows for easier experimentation with key components of the file system, while minimizing perturbation of the architecture as a whole. The second concept is that PVFS2 must be able to take advantage of the underlying hardware as efficiently as possible. This means that the PVFS2 abstractions must map well to underlying

Figure 3.1: PVFS2 system architecture



hardware and not impose artificial bottlenecks in the system. This efficiency goal is aided by the aforementioned modularity, in that a pervasively modular design results in a system that can more easily adapt to emerging technology. Finally, PVFS2 must be capable of supporting the next generation of parallel computers. The trend in system architecture is to scale these parallel computers to ever larger sizes in order to accommodate increasing demand for computational performance. The file system must scale accordingly to obtain balanced overall performance.

3.2 Architecture

3.2.1 System Layout

PVFS2 uses a client/server architecture, with both the server daemon and client side libraries residing fully in user space. There may be any number of servers, and each server may provide either metadata, file data, or both. Metdata refers to attributes such as timestamps and permissions as well as file system specific parameters. File data refers to the actual data stored in the system. This data is distributed according to a user tunable distribution module. The default scheme is to stripe data evenly in a similar manner to that of a RAID array [64]. Metadata may also be distributed, though at the granularity level of one server per individual file or directory.

It is important to note that the PVFS2 architecture places an emphasis on minimizing shared state. This is evidenced in several design decisions:

- By default, PVFS2 servers communicate exclusively with clients. There is no dependence between servers. This reduces complexity and helps to ensure that failures on individual servers do not propagate to peer servers.

- No locks are shared across hosts participating in the file system. This is contrary to conventional parallel file system designs, which rely on distributed locking mechanism to maintain consistency. PVFS2 instead provides semantics that, while suitable for high performance parallel applications, do not require locks to implement. These semantics will be covered in greater detail in section 3.2.3. In addition, the order of operations in complex file system functions is carefully chosen to maintain consistency in the name space and metadata without holding shared locks.

- Clients are not allowed to cache file data at all in the general case. This imposes a penalty on some single process workloads, but results in a much simpler protocol, better consistency, and no need to invalidate remote caches. This in turn simplifies the handling of faults, and improves performance for common parallel workloads may be prone to cache collisions and false sharing.

- All servers are have a stateless request interface. Each request is treated as a separate entity, and interdependence between servers and clients is strictly limited. For example, the servers do not track which files a given client has open; in fact there is no "open" operation in the server request protocol. This again reduces the risk of error propagation and allows servers to continue without incident if clients become unresponsive over the network. The lack of shared locking and client caching as outlined above also factor into the stateless design.

Figure 3.2: PVFS2 client interface layers

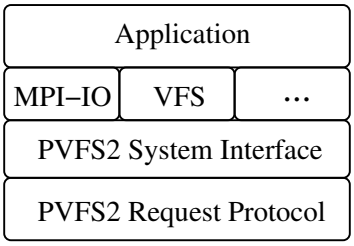


Figure 3.1 diagrams a typical system layout. Notice that there is no network communication between servers, and that each server accesses its own local storage resources (hard disks).

3.2.2 User Interfaces

PVFS2 is capable of supporting a variety of application interfaces. The two most common are MPI-IO [20] and the POSIX file system interface [41]. Implementors may wish to provide other domain specific abstractions as well. PVFS2 accommodates this by exposing a single low level programmer interface that can be used as a building block for any number of abstraction layers. This API is called the *PVFS2 system interface*. The system interface is implemented as a user level library which may be linked into any application. Figure 3.2 illustrates how these components fit together. It is important to note that the PVFS2 system interface hides the complexity of the server communication protocol from the interfaces which are built on top of it.

The primitives provided by the system interface are intentionally low level, so that they do not bias their use towards any particular abstraction. For example, the system interface does not operate in terms of file descriptors, because these are a POSIX specific construct and could impede other abstractions. The system interface does, however, expose many PVFS2 specific features. It provides access to user tunable parameters such as the file distribution, or the number of servers to use when

storing files. API's that do not support these features may ignore them or use default values.

MPI-IO is the preferred path for user level parallel applications to access PVFS2. The MPI-IO API is tailored to parallel access, including several concepts such as arbitrary datatypes and collective I/O. In addition, some implementations offer several file system independent optimizations [82, 80]. Above all, MPI-IO is portable to many different file systems, which gives it a distinct advantage for applications which may run on several different machines. MPI-IO support for PVFS2 is provided through the ROMIO MPI-IO implementation [81]. ROMIO leverages an abstract device interface to support multiple file systems. The PVFS2 abstract device is built directly on top of the aforementioned PVFS2 system interface.

ROMIO takes advantage of the system interface in several ways:

- MPI-IO datatypes can be passed along to PVFS2 with minimum conversion, because the system interface natively supports its own form of abstract datatypes.
- The system interface exposes parallel I/O specific tuning parameters which can be accessed through existing ROMIO hints.
- Some operations may be implemented more efficiently because of the format of the system interface primitives. For example, opening files on a traditional file system requires each process to carry out the open step to obtain a file descriptor. In PVFS2, however, there is no open step. Only one process is required to look up an opaque identifier for the target file and then broadcast this identifier if needed. File open therefore remains roughly constant as the number of processes involved in the open increases.

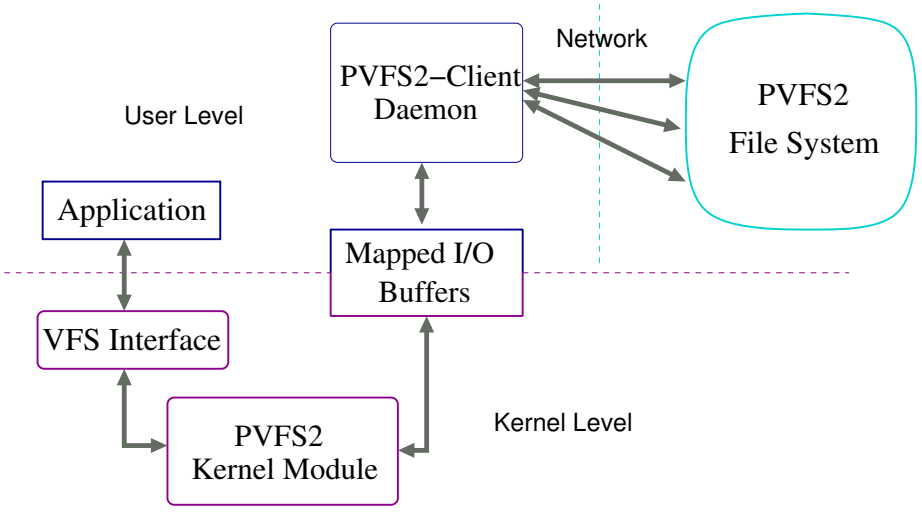
The POSIX file system interface is also a popular choice for interacting with PVFS2. Its API is not as rich as that of MPI-IO, and is does not offer any primitives that are tailored for parallel access. However, it does possess an advantage in that

it works with non-MPI applications as well as legacy MPI applications that do not leverage MPI-IO. It is also the most convenient interface for interactive work because of compatibility with existing system tools such as *ls* and *cp*.

The Linux operating system supports multiple file systems transparently through the kernel level Virtual File system Switch (VFS). New file systems can be added by implementing a loadable kernel module. PVFS2 employs this approach.

The PVFS2 kernel module implementation is based on a somewhat unorthodox design, although similar techniques have been leveraged in previous file systems such as Coda [9] and PVFS1 [75]. Figure 3.3 illustrates the architecture of this design. In this design, the kernel module itself is as lightweight as possible. This module simply translates kernel file system requests (from the Linux VFS layer) and passes them on to a user level daemon, called the pvfs2-client. This daemon is responsible for actually carrying out the operation and communicating with the file system servers. Communication with the PVFS2 client daemon is carried out using intermediate memory buffers that are shared between kernel and user space. This indirection results in a minor performance penalty, but it offers several advantages. First of all, the module itself is easier to port to different operating systems since it isn't carrying out any low level operations such as network transmission. Secondly, the design is easier to debug because most of the complexity resides in a user level component. This also minimizes the impact of file system faults on the kernel. Finally, the user level daemon can use the same user level system interface as that employed by MPI-IO and other domain specific libraries. This includes inherent access to a variety of network protocols (covered in section 3.3.1) without the need to implement kernel level hooks to those networks.

Figure 3.3: PVFS2 kernel VFS interface architecture



3.2.3 Semantics

Proper choice of file system semantics are critical to achieving high performance for the anticipated workload of a file system. For parallel file systems such as PVFS2 which target scientific workloads, the standard POSIX file system specification [40] is not appropriate. Other common Linux file systems such as EXT3 and NFS likewise elect not to comply fully with the POSIX definition. EXT3 does not enforce atomic writes across block boundaries by default, and NFS caches both reads and writes too aggressively to insure concurrent consistency across multiple clients.

The most significant limitation to the POSIX definition is that it requires sequential consistency. That is, all read and write operations must be atomic with respect to each other. In parallel file systems, this means that even writes that span a large number of servers simultaneously must be atomic. Implementation of this semantic would require implicit locking or inter-server communication which would hinder performance even in scenarios in which those semantics are not required.

PVFS2 instead implements *nonconflicting writes* semantics [63]. This means that any concurrent writes which do not access the same bytes of a given file are guar-

anteed to be consistent, and subsequent reads of any region are guaranteed to obtain correct data. This applies not just on block boundaries, but also byte boundaries. However, if concurrent writes are issued to the same file region, then the results for that region are undefined. Likewise, reads which conflict with concurrent writes will receive undefined data.

These semantics are sufficient for the majority of parallel applications (including standard MPI-IO calls) and do not require additional locking or server communication to implement.

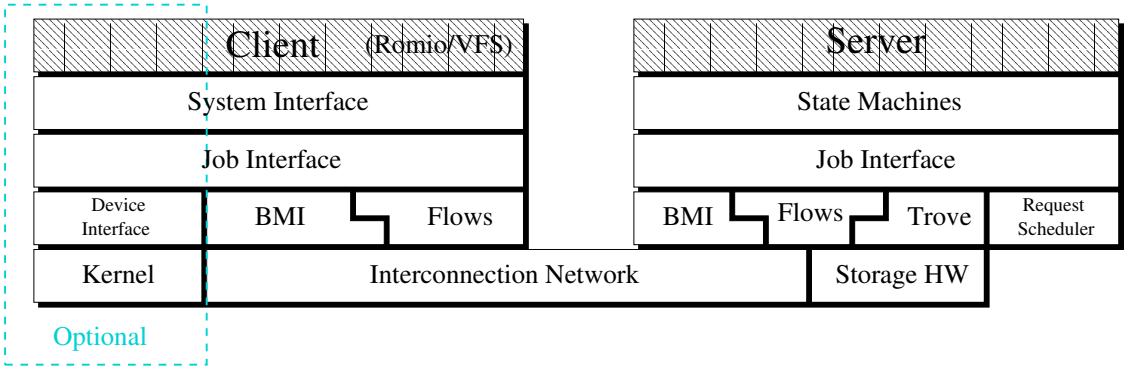
PVFS2 also does not implement any form of explicit locking as dictated by the POSIX standard, including but not limited to the *flock* interface. We will show later that locking subsystems are an unnecessary burden on the complexity of parallel file systems and are not necessary for most applications.

PVFS2 provides sufficient semantics for implementation of an MPI-IO abstraction, with the exception of atomic mode and shared pointer mode. The most popular MPI-IO implementation, ROMIO [81], requires file system locks to implement these modes. We are currently researching alternatives to file system level locks for implementation of these modes of operation.

3.2.4 Software Components

Figure 3.4 shows a diagram of the primary internal I/O components of PVFS2. The lowest level network abstraction is provided by a component known as the Buffered Message Interface [13, 12]. The counterpart disk abstraction, which provides both stream and key/value style access to local storage resources on each server, is called Trove. These two components are coordinated by flows, which handle buffering, scheduling, and datatype processing between network and disk for bulk transfers. The client includes an additional component for managing communication with the kernel device if the optional kernel module is used. The server includes a request

Figure 3.4: Primary PVFS2 components



scheduler for controlling consistency between concurrent requests. All of these lowest level components are coordinated by the job interface, which manages threading and provides a consistent interface for testing of completion of any pending low-level I/O operation, regardless of which underlying component is ultimately responsible for it. Both the servers and client libraries are implemented on top of the job interface. The servers organize operations in terms of state machines, while the clients organize operations in terms of a system interface library.

3.3 I/O Path Detail

Several components from figure 3.4 are relevant to discussion of the research proposed in section 1.4 of the introduction. The details of four such components are outlined below. BMI and flows are pertinent because they are the elements responsible for network communication between file system hosts. The request scheduler is important because of its role in controlling the timing of operations. Jobs serve as the glue that hold together all of the preceding infrastructure and are thus critical to any high level discussion of their interaction.

3.3.1 BMI

BMI is a software network abstraction layer which allows PVFS2 to operate on a variety of dissimilar networks. It has been specifically designed to provide semantics and scalability to suit a large scale parallel file system. Previous work has demonstrated the importance of implementing a messaging system that is optimized for its intended environment [11]. Particular requirements of this problem domain include: efficiency, support for common parallel I/O access patterns, support for multiple concurrent networks, thread safety, explicit buffer management, API support for an asymmetric client / server model, fault tolerance capability, and minimization of state exposed to the user.

BMI is intended for use by system level services. It is constructed with a layered interface model; BMI presents a high level API for BMI users while also providing an internal device API for specific network implementations. The latter interface eases the task of porting to new network infrastructures. Each device resides in an independent module. BMI has been implemented and tested extensively on three different protocols: TCP/IP, GM, and InfiniBand. In the following sections we will describe some of the specific features that help it to accommodate the parallel I/O problem domain requirements.

API BMI implements a nonblocking interface for all network I/O operations. The basic model is to first *post* an operation and then *test* the operation until it is completed. Completion in this model refers to *local* completion; it offers no guarantee of success on the remote peer. The nonblocking interface allows many network operations to be in service concurrently, each possibly in a different stage of communication. Operations are referenced by unique identifiers while they are in service. Receive buffers do not have to be posted in advance, though some networks will achieve higher performance if they are.

BMI embraces the client/server model used by parallel file systems through the use of *tags* and *unexpected messages*. Tags are integer parameters which can be used to match messages exchanged as part of a single overall file system operation. Normally, incoming messages are paired with receive operations with the proper sender, size, and tag parameters. Unexpected messages are an exception, however, in that they do not require a matching receive to be posted. Instead, the receiver simply polls to check for new unexpected messages. If such a message is found, then a descriptor is filled in that describes the parameters of the message and provides a pointer to the data buffer. This reduces complexity on the server side because the server does not have to anticipate buffer use in advance. Instead it can just react to incoming messages and use them to initiate service state machines.

The BMI API also allows multiple application or server components to use the same interface concurrently. BMI supports this foremost by being fully thread safe. Secondly, it introduces *contexts* to help differentiate between independent higher level callers. Each component that uses BMI will receive its own unique context, which is local to that host. This context can then be used to differentiate between operations posted by each component, both at post time and at test time. This allows components (or threads) to test for completion of any pending operations without the risk of receiving notification of completion for an operation posted by a different component.

Performance BMI implements several features that are intended to improve efficiency. Some of these are simply optimizations on the basic API model outlined earlier. One important optimization on the post and test model is that any post function may elect to indicate *immediate completion* at post time. Immediate completion means that the operation has successfully finished during the execution of the post call; therefore, no testing step is necessary. In some scenarios, such as very small

sends, or receives for which the data has already been buffered, this will avoid the overhead of calling an extra function to retrieve the status information.

Many modern user level network protocols as VIA [86] or GM [58] rely on the use of message buffers that are pinned into physical memory before transmission. BMI accommodates this by providing functions for allocating and releasing buffers that are optimized for use by BMI. The use of these functions is optional, however, and BMI will handle buffering internally if needed. This is important for client library usage in which there is no opportunity to register buffers in advance.

PVFS2 supports the use of arbitrary data types to describe patterns of offsets and sizes within a file. It also allows data to be striped across an arbitrary number of hosts. These two features lead to scenarios in which communication must be carried out from a set of many noncontiguous buffers. BMI allows sets of noncontiguous buffers to be sent or received in a single function call through the use of *list* operations. List operations are similar to their traditional send and receive counterparts, except that they operate on an array of memory offsets and sizes rather than a single buffer. This can cut down drastically on the number of messages necessary to transfer a complex data pattern between two hosts. Some BMI implementations may directly support list operations and use hardware-provided scatter/gather support to move the noncontiguous buffers.

Scalability The ability to handle a large number of concurrently posted operations is critical to file system scalability. The *user pointer* field associated with each operation is one feature designed to help in that regard. The user pointer is an opaque value that may be set by the caller at post time. It is returned unchanged to the caller when a test indicates completion. It provides a mechanism for the caller to map completed operations back to some higher level data structure outside of BMI after calling a test function. For example, on the server side it may map the operation back

to a state machine that tells the server what to do next. Thus, no matter how many operations are in flight, the originating function or data structure can be located for each completion with O(1) complexity.

It is also important for scalability to insure that the caller does not have to execute a test function separately for each pending operation. This would consume too much CPU time even with just a few posted operations. There are two variations on the test function that overcome this problem. One variation allows a single call to test for completion of any of a set of specified operations. Another variation allows a single call to test for completion of any previously posted operation without specifying the operation identifiers. This last function is significant because it prevents a busy server or library from having to construct a list of operations to test on; instead, it just checks for any possible operation that may have completed in a single function call.

A final key to BMI API scalability is that it is connectionless. There is no state to maintain for a given peer on the network, and no connection to set up or tear down in preparation. This simplifies communication and aids in scalability when communicating with thousands of hosts. Note that if BMI is built on top of a network that uses a connection oriented model, then BMI will manage the connections transparently underneath the API, likely by caching previously used connections in hope of later reuse.

Fault Tolerance Clean handling of file system and network faults is necessary for modern large scale file systems. BMI addresses this issue by working in concert with fault handling capability at higher levels of the file system. For example, BMI does not automatically retry transmission of failed network messages. This is impractical in the general case because network messages in parallel file systems are typically just a single part of a larger multistep operation. Automatic retransmission at the net-

work API level could lead to inconsistent requests if a server is restarted or fails over, or it could simply lead to duplicate operations. For this reason, BMI relies on the server or client libraries to determine the appropriate retransmission points. It accommodates this decision by preserving network address information and transparently reconnecting or utilizing secondary network interfaces as needed. BMI also improves fault tolerance by providing the ability to cancel network operations that have been posted but have not yet completed, therefore giving higher level components a clean interface to handle time out conditions.

3.3.2 Flows

The flow interface is the PVFS2 component responsible for buffering data between network and disk for bulk I/O transfers. In addition, it interprets the I/O description and distribution information to determine where to place data in memory or in local storage objects. The flow interface must therefore serve as the focal point of management between the BMI (network), Trove (disk), and request processing (I/O description and distribution) components.

Like the BMI implementation, the flow interface is designed in a modular manner to allow for multiple underlying implementations. The top level API is as abstract as possible; it simply describes where the data is coming from, where it should be placed, and what the datatype looks like. The source and destinations are described by *flow endpoints*, which contain either network addresses, storage object references, or memory addresses. It is up to an underlying *flow protocol* to actually carry out the work of moving data. This choice of architecture allows the potential for several underlying optimizations because the actual transfer mechanism between disk and network is hidden. For example, a flow protocol could perform explicit caching, or could use operating system features to make direct transfers from disks to network

cards. There may also be multiple flow protocols active simultaneously; they can be selected on a per operation basis.

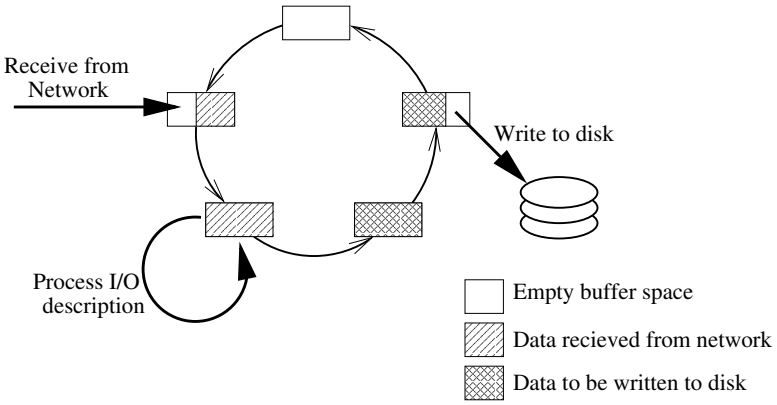
Flows are used on both the client and server side of all I/O operations. Each party must post a matching flow operation. For example, a client may post a memory to network transfer, while a server may post a network to disk transfer. This example describes the transfer of data in a file system write operation. The network is the common ground between the two client and server flows. Like the BMI interface, the flow interface uses message tags to properly match concurrent flows.

The flow interface is asynchronous and executes a user defined callback function upon completion of the transfer. In addition, the BMI and Trove interfaces that it leverages are asynchronous. Flow protocols must therefore be capable of managing threading and asynchronous completion. This includes proper cleanup of previously posted low level operations if a flow is canceled or reaches an error state.

Multiqueue Flow Protocol Although the flow interface allows for the possibility of multiple flow protocols, there is only one production mode implementation at this time. This flow protocol is known as the *multiqueue* protocol. It explicitly buffers data between network and disk on the server side by using a set of several fixed size buffers per flow. The number and size of these buffers is configurable. The choice of these parameters has significant impact on the amount of overlap and pipelining that will be achieved during the course of the transfer. The buffers can be thought of as being arranged in a ring formation. Each one continuously cycle between three states: filling from the source endpoint, I/O description processing, and emptying to the destination endpoint.

Figure 3.5 gives an example of this buffer progression for a network to disk flow running on a PVFS2 server. Each buffer receives data from the network, then checks the I/O description, and finally writes to disk. The I/O description may indicate that

Figure 3.5: Server flow example



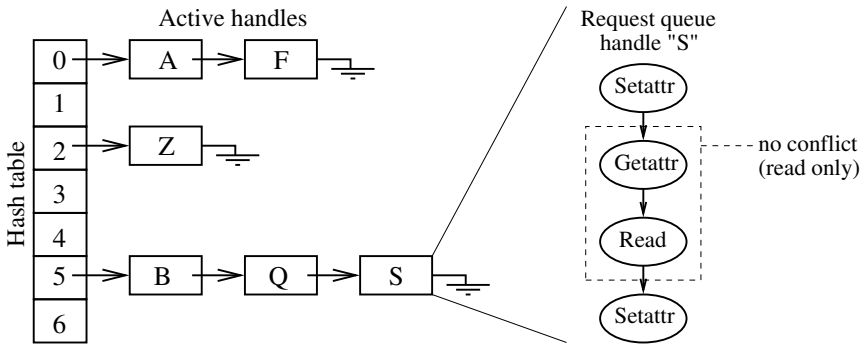
a single buffer should be written out as many discontiguous regions to disk as a result of the datatype description provided by the end user or the distribution of file data. The flow protocol avoids breaking apart buffers in this manner until as late as possible in order to maximize efficiency. Although it is not shown explicitly in this diagram, it is possible for multiple operations to each subsystem to be posted concurrently. In fact, this is the normal mode of operation and the reason that multiple buffers are used in the first place. The Trove storage subsystem in particular achieves improved throughput when many buffers are posted concurrently rather than posting them sequentially.

The multithreading aspect of the multiqueue flow protocol is actually handled by an external thread management component. This component accepts low level operations as input and issues callbacks upon completion. The callbacks in turn drive the state of the flow protocol and its buffer progression.

3.3.3 Request Scheduler

PVFS2 servers are designed to allow as many operations as possible to proceed concurrently. This is essential to achieving throughput for parallel applications which tend to generate many simultaneous requests. However, this situation results in pos-

Figure 3.6: Request scheduler



sibility of processing concurrent requests that perform conflicting operations. For example, two concurrent requests to set attributes could produce undefined results; the actually server processing is not atomic. We must therefore perform consistency checking before allowing concurrent operations to proceed.

The PVFS2 *request scheduler* is the component which performs the necessary consistency checking as well as scheduling on a per request basis. It operates strictly on the server side of the file system and has no dependencies across hosts. Like the I/O interfaces in PVFS2 (BMI, Trove, and flow interface), the request scheduler exports an asynchronous interface to callers. The caller in this case is a PVFS2 server state machine. The state machine posts a request to the scheduler as soon as it has been decoded. The scheduler asynchronously reports when the request is free to proceed. In other words, it suspends the request until all consistency constraints for the request have been met.

The architecture of the current request scheduler implementation is illustrated in figure 3.6. A hash table is maintained with an entry for each active handle, where an active handle is defined as a file system handle for which requests are pending. Each active handle is associated with an independent request queue. This example shows a detailed view of the queue for handle S. There are four requests pending which are preserved in FIFO order. Each time a new request is allowed to proceed,

the queue is scanned to find any adjacent operations which may also proceed. In this case, after the first set attributes operation is completed, then the next two operations (get attributes and read) will be allowed to proceed at the same time because they do not conflict. In general only read only operations are allowed to proceed concurrently for a given handle.

The PVFS2 request scheduler is in some ways just a proof of concept which insures sufficient request ordering to maintain PVFS2 file system semantics. A future implementation may use a rule based system to define more complex relationships between available operations. Example extensions include examining I/O datatypes for conflict, or throttling the number of simultaneous operations of a given type.

3.3.4 Jobs

The job interface is the PVFS2 framework responsible for coordinating concurrent activity between the BMI, Trove, flow, and request scheduling components of PVFS2. One aspect of this coordination is thread management. The job interface can be configured to operate with or without threads. In the former case one thread is assigned per underlying component to motivate progress, while in the latter case the underlying components must multiplex time spent in job function calls. Both modes present a consistent interchangeable API for callers. The threaded mode is most frequently used on PVFS2 servers.

The second purpose of the job interface is to provide a single unified point for checking for completion of operations from any I/O component. For example, a caller may wish to check for completion of both BMI operations and Trove operations at the same time. The job interface accomplishes this by integrating all operations into a single unified queue that can be queried by way of a single set of access functions. Like the underlying components that it coordinates, the job interface utilizes a nonblocking interface.

3.4 Performance Monitoring

PVFS2 includes substantial performance monitoring capabilities. These capabilities fall roughly into two categories: event logging and statistics monitoring. The event logging infrastructure uses instrumentation to record when various events occur, such as the start of a disk operation and the completion of a disk operation. These events are buffered into memory and may be retrieved at any time. This event logging mechanism provides a means by which to measure the impact of low level operations on file system behavior. The second category of performance monitoring deals with real time statistics monitoring. This again relies on internal instrumentation, but in this case the goal is to collect ongoing statistics about performance, such as how many bytes have been written to disk. These statistics can be collected by a client application in real time in order to monitor metrics such as throughput and number of metadata operations per second.

3.5 Operation Algorithms

In this section we will describe the algorithms used for a few example PVFS2 operations. The operations are: file creation, directory creation, file removal, directory removal, get attributes, and file system status. In all cases we will be outlining conventional algorithms; new versions that leverage intelligent servers and collective communication will be shown later in this study. All algorithms are illustrated from a relatively high level that emphasizes which servers are involved and what the communication pattern will be.

For example purposes we will assume that there is exactly one client and at least three servers, with each server performing a different role in the operation. The data server holds the actual file contents. The meta server holds information about files or directories, such as owner, permissions, and distribution. The parent directory

server holds a directory entry which ties the file system object into the file system name space. In practice, these same algorithms will work fine with just one server which performs all three roles. They will also work with an arbitrarily larger number of servers. We just choose three as an example that clearly separates where services are performed.

We will use the following notation in the algorithms of this chapter for brevity:

- C: represents a client process
- D: represents a data server
- M: represents a metadata server
- P: represents a parent directory server
- →: represents a request sent from a client to a server (response is implied)

For example, the statement "C → M create metadata object" indicates that a request was sent from the client to the metadata server in order to create a metadata object.

Note that we are omitting error cleanup steps for now and focusing on the normal case for each algorithm. Implications of error cleanup will be covered in a later discussion of file system consistency in section 6.4.

3.5.1 Create File

Figure 3.7: File creation algorithm

```
1 C → P get parent directory attributes
2 C → M create metadata object
3 for each D:
4     C → D create data objects
5 C → M set file attributes
6 C → P create directory entry
```

The algorithm for file creation is shown in figure 3.7. The parent directory attributes are retrieved first to verify permissions. The metadata and data objects are created next. Finally, attributes are set on the metadata object to associate it with the data objects and a new directory entry is added.

3.5.2 Get Attributes

Figure 3.8: Get attributes algorithm

```
1 C → M get basic attributes
2 if object is file and size requested:
3     for each D:
4         C → D get datafile attributes
5     C compute logical file size
```

The algorithm used to retrieve attributes for a file system object is given in figure 3.8. The first step is to retrieve the metadata attributes. If the object in question is a file, then each data server must be contacted to learn the size of the data objects. The client then computes the total size of the file by comparing the data object sizes to the distribution information for the file.

3.5.3 Remove File

Figure 3.9: File removal algorithm

```
1 C → P remove directory entry
2 C → M get file attributes
3 for each D:
4     C → D remove datafile object
5 C → M remove metafile object
```

The file removal algorithm is shown in figure 3.9. In this case the directory entry is removed first. The attributes from the metadata are then retrieve in order to

verify permissions and generate a list of data objects to remove. Finally, all metadata and data objects for the file are deleted.

3.5.4 Create Directory

Figure 3.10: Directory creation algorithm

```
1 C → P get parent directory attributes
2 C → M create new directory
3 C → P create directory entry
```

The directory creation algorithm is shown in figure 3.10. This operation consists of three single operations: retrieving attributes from the parent to verify permissions, creating the directory object, and creating a parent directory entry that refers to it.

3.5.5 Remove Directory

Figure 3.11: Directory removal algorithm

```
1 C → P remove directory entry
2 C → M get directory attributes
3 C → M remove directory
```

The directory removal algorithm shown in figure 3.11 is similar to that used for directory creation. It consists of three operations: removing the directory entry, retrieving attributes to verify permission, and removing the directory object itself.

3.5.6 File System Status

The file system status operation of figure 3.12 is the final algorithm we will show in this chapter. For a given file system, the client must contact every server (metadata or data) and retrieve statistics. The client then computes a summary of statistics for the file system as a whole.

Figure 3.12: File system status algorithm

```
1 for each D:
2     C → D get server statistics
3 for each M:
4     C → M get server statistics
5 C compute summary
```

CHAPTER 4
PARALLEL VIRTUAL FILE SYSTEM 2 EXTENSIONS

Chapter 1 introduced the concepts of intelligent servers and collective communication, while chapter 3 described our PVFS2 experimental platform. We will now outline specifically how PVFS2 has been extended to support the research presented in this study.

4.1 Server Intercommunication

Basic server intercommunication is the first building block necessary for the work presented in this text. It consists of three primary components. The first is configuration management and process identification. In other words, how to locate other servers and reference them properly. The second component is initiation of unexpected requests between servers within a system that was designed to be driven from the client's perspective. Finally, the communication framework had to be expanded beyond unexpected request messaging to include balanced peer to peer messaging as well.

PVFS2 clients typically operate in terms of object handles rather than servers. For example, to delete an object a client must map the object to a server and then issue a request. This mapping step is performed independently for each request to allow for changes in the association between handles and servers. We chose to extend this approach by porting the handle mapping interface to the server infrastructure. This provides an easy mechanism for servers to choose other servers for object requests. Object handles therefore essentially serve as a compact form of an address for each server. Handles are implemented as 64 bit integers. The configuration management

interface was also optimized in terms of time order complexity to limit the cost of resolving servers and handles in large scale file systems.

Some operations do not involve manipulating a particular object, and therefore do not reference a particular handle. To accommodate those cases, we extended the configuration management interface to query the handle mapping and provide *sample handles* for each server. A sample handle is simply a representative handle from the set controlled by any given server. Thus all PVFS2 server communications can operate in terms of handles and leverage a consistent routing and address lookup infrastructure.

The standard PVFS2 client orchestrates multiple concurrent requests through a construct known as a *message pair array*. Message pair arrays are implemented as nested state machines in the PVFS2 state machine infrastructure. These arrays perform several steps for each request: address resolution, buffer allocation, posting sends and receives, waiting for completion, and issuing callback functions for completed messages. Each request may proceed at an entirely different pace through these steps in order to allow for as much asynchronous concurrency as possible.

In the general case for server to server messaging, one server is essentially acting as a temporary client to another server. It therefore makes sense for servers to leverage the same message pair array functionality that clients do. We accomplish this by stripping client specific functionality from the nested state machine and modifying it to work in both server and client contexts.

The modifications outlined above were sufficient for most messaging patterns used in this text. However, we found that in some cases it was necessary for servers to exchange messages as peers, rather than with one temporarily acting as a client. The significance here is that both servers expect the incoming message, and wish to use it in an ongoing state machine. Normal server requests are transfered as unexpected messages, however, and trigger the launch of new state machine instances. In order to support peer to peer server messaging, we implemented a new construct known

as the *message exchange array*. Message exchange arrays are similar to message pair arrays, except that two parties exchange messages simultaneously rather than sending a request and waiting for a response. It was still necessary to leverage unexpected BMI messages for the communication, because a given server may not know in advance the BMI address of the server it will receive from. We therefore designate a special BMI message tag value for use by message exchange arrays. Incoming messages with this tag are queued until a matching message exchange requests the data.

4.2 Collective Communication

Having established basic server to server communication, the next step was to expand point to point primitives into structured communication patterns. The three most useful patterns in parallel file systems are the one-to-many, many-to-one, and all-to-all network constructs. We chose to use binary trees for the one-to-many and many-to-one patterns and recursive doubling for the all-to-all patterns. Both of these algorithms are described in the related work of section 2.1 and are known to work well for the types of switched networks commonly deployed in cluster computers.

We were not able to directly apply existing collective communication implementations. First of all, most mature collective implementations exist as part of a message passing library such as MPI, and it is not appropriate to force all PVFS2 clients and servers to act as MPI applications. Secondly, and most importantly, there are several distinctions between message passing and parallel file system environments that must be accounted for. We therefore created our own novel collective communication framework. The following list briefly summarizes the key differences:

- **Unexpected messages**: In most message passing libraries, each process participates in a collective by executing a blocking library call. Therefore each process is dedicated to the task and prepared to handle incoming messages. In contrast, file system servers do not know in advance when a collective will

occur. They must be able to process unexpected messages that are part of a structured communication and process them accordingly. This may also lead to increased delay at each communication step.

- **Adaptive routing**: Most message passing implementations either determine the ordering and structure of communications statically or with a fixed algorithmic model. File system server may instead wish to alter communication ordering in response to file system state or performance. For example, a server may restructure a binary tree at run-time in order to give more communication work to a less heavily loaded server. The PVFS2 binary tree algorithms allow for adaptive rerouting at each communication phase.

- **Process identification**: All message passing libraries include some standard for how to identify each process, usually with an integer, and assign an ordering to those processes. This is not the case in PVFS2, in which servers are referenced by a string identifier and clients may enter or leave the system at any time. At this time we do not allow clients to participate in collectives, thereby avoiding the latter problem. For servers, we use handles for identification as described in the previous section and rely on static configuration files to assign consistent ordering of servers in the system.

- **Fault tolerance**: Most mature message passing implementations do not provide fault tolerance. Failure of a process typically results in termination of the application as a whole. This is not acceptable in a production level parallel file system. We therefore integrate error detection at each stage of the operation and report error status back to the root of any collective operation. We will also describe later how the adaptive routing can be used to avoid failed links or servers.

The choice to implement our own collective communication framework was clear after reviewing the points listed above. There were two options for implementation, however: integrating collectives into the BMI network layer or implementing them on top of BMI at the server level. We chose the latter option based on three design issues. The first was our desire to allow dynamic routing, as listed above. Only servers have enough information to make these routing decisions unless the BMI interface were heavily augmented to allow definition of routing functions. The second issue was protocol representation. At the server level, we have the opportunity to decode, break apart, and reassemble request messages. This allows us to construct messages as compactly as possible at each stage by not duplicating data and only transmitting the minimum information needed. Finally, the BMI layer lacks any type of compact global process identification. Servers on the other hand can utilize object handles for this purpose in most cases.

Collective communications implemented at the server level consist of two primary components. The first is a set of functions that can be used to separate an array of handles into subsets based on the desired collective algorithm. For example, one function can break an array of handles into chunks to be sent to the next servers in a binary one-to-all operation. These functions include hooks for filtering the results and reordering based on load or locality. The second component is the actual communication progress engine, provided by either the message pair array or message exchange array constructs, both defined in the previous section.

4.3 Server Status Composition

We will define *server status composition* as the process by which a set of servers communicate with each other to share statistics and create a global summary of the file system state. This has been implemented as an extension to PVFS2 that periodically runs an all-to-all exchange among all servers in the file system. The

Table 4.1: Server status composition fields

file system identifier
available storage space
available RAM
available handles
total storage space
total RAM
total handles
load averages
server uptime

communication component is carried out using a recursive doubling algorithm. This composition is performed by a continuously running state machine that resets a timer for a predefined interval after each exchange.

At the end of each communication exchange, each server possesses a relatively up to date copy of the statistics from every other server. This information can then be used in a variety of ways. A simple application is to allow any server to quickly respond to statfs requests. More interesting applications are to use this information for load balancing decisions or health monitoring. Each of these potential uses will be covered in greater detail later on.

The initial implementation updates every 500 ms and includes all of the parameters given in table 4.1. Each server stores a copy in its local memory. A history of multiple time-steps is preserved for future algorithms that may wish analyze system trends over time.

4.4 State Machine Infrastructure

Multistep operations on both the client side and server side of PVFS2 are implemented using state machines. A state machine in this context is made up of a set of functions to be executed along with a dependency chain defining what order they will be executed in. State transitions occur at points in which low level I/O opera-

tions, such as disk or network access, are posted. When the low level I/O operation completes, its error code is used to determine which state in the dependency chain to transition to next.

State machines in PVFS2 are represented using a custom language that describes the functions to execute and how to transition between them. This state machine language is digested by a state machine compiler to generate standard C code that will be compiled into the file system.

State machines play a critical role in the intelligent server implementation described in this text. Two particular modifications were made to the state machine descriptions and processing framework to make this possible. The first modification was to simply reorganize existing state machines so that reusable parts could be separated into independent *nested* state machines. Nested machines are state machines that are executed as a single unit within a higher level machine. This is important because the extended PVFS2 operations are intended to be fully compatible with existing PVFS2 operations. We therefore reuse as many metadata access routines as possible rather than rebuilding them from scratch.

The second state machine modification was actually a change to the core processing ability of the state engine. In a conventional state machine, only one state is executed at a time. Thus all state machine steps are fundamentally sequential, although there is no restriction on the amount of work performed per step. This poses a problem for intelligent servers when they participate in collective operations. Each phase of the collective typically consists of both a local data access as well as a forwarding communication to the next participant in the collective. Since both of these tasks may be high latency operations dependent upon separate threads or I/O devices, it would be best if they were executed concurrently. We therefore added *forked* state machine capability to PVFS2.

Figure 4.1: Forked state machine example

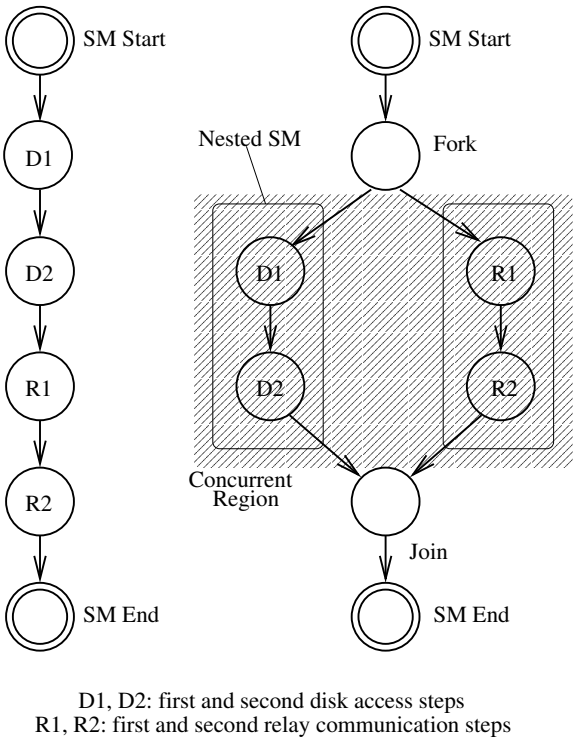


Figure 4.1 illustrates this concept. On the left side of the figure, we show an example of the primary path through a standard state machine. D1 and D2 represent two disk access steps while R1 and R2 represent the communication steps necessary to relay a request to the next participant in a collective operation. Notice that all of the disk accesses must complete before continuing to the communication. On the right hand side we have given an example of a forked state machine that performs the same work. The disk accesses have been grouped into a nested state machine, as have the communication relay steps. Those two nested machines can then be processed simultaneously to achieve as much overlap between communication and local work as possible. The concurrent region is shaded in this figure. Notice that we have added two additional steps, labeled fork and join in this case, to setup the concurrency. This additional overhead is outweighed by the concurrency gain that it enables.

4.5 Operation Algorithms

Several PVFS2 operations have been reimplemented using intelligent server and collective communication capability. However, in all cases we have preserved compatibility with the operations defined in section 3.5. The disk format has not changed and the basic semantics have not changed. For experimental purposes we have simply added a new set of client API functions that provide aggregate counterparts to the standard functions. Either set may be used interchangeably. In addition to the new API functions, we also created counterpart administrative tools and an alternative version of the pvfs2 kernel driver that utilize them.

In this section we will outline the algorithms used for each new API function. Where applicable, we have added additional steps to indicate where servers have introduced new "locking" points. It is important to note, however, that the term lock is just used for notational purposes to describe the semantics that are achieved. In practice these regions are protected by the request scheduler defined in section 3.3.3 which queues operations on objects according to a well defined set of consistency rules. In no cases is any underlying file system lock used, nor is any lock state shared across servers or clients.

The notation used here is the same as that used when describing the conventional operation algorithms in section 3.5. The only difference is that we will use the $\leftarrow$ symbol in most cases to denote the response to an aggregate request. This is done to point out when servers perform operation steps on behalf of the client before sending a response.

4.5.1 Create File

The aggregate file creation algorithm is shown in figure 4.2. The client first contacts a parent directory server that will act as an intelligent server for the remainder of the operation. This server locks the parent directory, creates all file objects (possibly at

Figure 4.2: Aggregate file creation algorithm

```
1 C → P aggregate request
2 P locks parent directory
3 P → M create metadata object
4 for each D:
5     P → D create data object (collective)
6 P → M set file attributes
7 P creates directory entry
8 P unlocks parent directory
9 C ← P aggregate response
```

different servers), and then sets the metadata attributes to tie the objects together. It then unlocks the parent directory and sends a response to the client.

4.5.2 Get Attributes

Figure 4.3: Aggregate get attributes algorithm

```
1 C → M aggregate request
2 M locks metadata object
3 M get metadata attributes
4 if object is file and size requested:
5     for each D:
6         M → D get datafile attributes (collective)
7     M computes logical file size
8 M unlocks metadata object
9 C ← M aggregate response
```

Figure 4.3 shows the aggregate algorithm for retrieving object attributes. The metadata server serves as an intelligent server in this case. It first locks the metadata object, then retrieves attributes for the object and any related data objects if needed. Finally it computes the total file size if needed, unlocks the metadata object, and sends a response to the client.

Figure 4.4: Aggregate file removal algorithm

```
1 C → P aggregate request
2 P locks parent directory
3 P → M get metadata attributes
4 for each D and M:
5     P → D remove data object (collective)
6     P → M remove data object (collective)
7 P remove directory entry
8 P unlock parent directory
9 C ← P aggregate response
```

4.5.3 Remove File

The aggregate file removal algorithm is shown in figure 4.4. As in the file creation algorithm, the parent directory server is chosen to be an intelligent server and keeps the parent directory locked during the course of the operation. In this case the parent directory server retrieves the metadata attributes, then deletes all objects, and then removes the associated directory entry. The algorithm concludes by unlocking the parent directory and sending a response to the client.

4.5.4 Create Directory

Figure 4.5: Aggregate directory creation algorithm

```
1 C → P aggregate request
2 P locks parent directory
3 P → M create metadata object
4 P create directory entry
5 P unlocks parent directory
6 C ← P aggregate response
```

Figure 4.5 shows the aggregated directory creation algorithm. The client contacts the parent directory server. The parent directory server then locks the parent directory, contacts another server to create the object, and adds a new entry to point

to it. The server finally unlocks the parent directory and sends a response to the client.

4.5.5 Remove Directory

Figure 4.6: Aggregate directory removal algorithm

```
1 C → P aggregate request
2 P locks parent directory
3 P → M get metadata object attributes
4 P → M remove directory
5 P remove directory entry
6 P unlocks parent directory
7 C ← P aggregate response
```

The aggregate directory removal algorithm is shown in figure 4.6. The parent directory server acts as an intelligent server and locks the parent directory over the course of the operation. It first retrieves the metadata attributes to verify permissions and determine if the directory is empty. It then removes the directory and the associated directory entry. Finally, the parent directory is unlocked and a response is sent to the client.

4.5.6 File System Status

Figure 4.7: Aggregate file system status algorithm

```
1 C → M get all server statistics
2 C computes summary
```

The aggregate file system status operation given in figure 4.7 is the final algorithm we will cover in this chapter. We assume that a server status composition operation (see section 4.3) has already distributed file system statistics to each server. The algorithm then simply consists of a single request to retrieve those statistics from

any server in the file system and a client computation to create a summary of those statistics.

CHAPTER 5
MODELS AND RAW PERFORMANCE

File system models can be used for comparing potential optimization alternatives, predicting application performance, or extrapolating behavior beyond the bounds of available hardware. In this work we are particularly interested in predicting file system performance at extreme scales and choosing appropriate algorithms for that environment. For this purpose have developed analytical models of file system operations that take into account system network, processor, and disk characteristics.

We outlined the algorithms used to implement various metadata operations in PVFS2 in section 3.5 and then expanded upon them using intelligent servers and collective communication in section 4.5. At this point we will describe analytical cost models which can be used to compare these algorithms and predict their behavior on large scale systems. The models will be contrasted with various empirical results gathered on real world file systems to verify their accuracy for known performance ranges.

This chapter will focus exclusively on model construction and verification. The models will later be used for performance prediction when evaluating file system efficiency in chapter 6

5.1 Experimental Platform and System Settings

The empirical measurements presented in this and the following chapters were gathered on two systems: the Adenine Linux cluster at Clemson University's Parallel Architecture Research Laboratory, and the Jazz Linux cluster at Argonne National Laboratory's Computing Resource Center.

As of this writing, Adenine is made up of 75 compute nodes. Each compute node consists of a dual processor Pentium III 1 GHz with one gigabyte of RAM. The nodes are connected by a 100 Mb/s Ethernet network with a single dedicated switch. The system software includes Linux kernel version 2.6.9 and GNU C library version 2.3.2. All test programs (and the PVFS2 file system) were compiled with gcc version 3.3.2 using "-O3" as the only compiler switch. The disk drives in Adenine are Maxtor 30 GB 5T030H3 IDE hard disks, operating in UDMA mode on a Serverworks OSB4 IDE controller. The local file systems are all of type EXT2.

Jazz is made up of 350 compute nodes, 250 of which were made available for experimentation in this dissertation. Each node contains a single Pentium Xeon 2.4 GHz processor and at least 1 gigabyte of RAM. The nodes are connected by both a 100 Mb/s Ethernet network and a 2 Gb/s Myrinet 2000 network. The system software includes Linux kernel version 2.4.26 and GNU C library version 2.2.4. All test programs (and the PVFS2 file system) were compiled with gcc version 2.96 using "-O3" as the only compiler switch. The disk drives are IBM 82 GB IC35L080AVVA07 IDE hard disks, operating in UDMA mode on an Intel 82801CA IDE controller. The local file systems are all of type EXT3.

All model parameters presented in this and following chapters will be given in terms of these two systems. Use of these models to predict performance on other systems will require a set of benchmarks and trace file analysis to calibrate the new parameters.

Several system parameters and caveats have an impact on the experiments presented in this study. These include Myrinet network behavior, file system synchronization, and client side caches. The Myrinet network has an important characteristic in addition to its low latency and high throughput. This network uses the GM protocol which bypasses operating system overhead and offloads as much protocol and memory transfer overhead as possible to the network interface cards. These fea-

tures allow servers using Myrinet to overlap more communication with computation or disk service than the TCP/IP network does, thereby complicating models which attempt to take this factor into account. This overlap is enhanced by the forked state machine optimizations employed in the aggregate operations, as described in section 4.4. In addition, the Myrinet network on Jazz was found to be unable to support the network patterns generated by PVFS2 metadata operation benchmarks on more than approximately 125 nodes. We therefore limit all Myrinet network results to that scale despite the availability of 250 nodes for experimentation, all of which were used successfully in Ethernet TCP/IP testing.

Underlying file system synchronization was also found to have a significant impact on performance. The PVFS2 servers accept configuration settings which control whether individual metadata or I/O operations are implicitly synchronized to the underlying storage. We disable this functionality because implicit synchronization hinders performance and is not the normal operating mode of production file systems. In addition, however, it was found that the underlying Berkeley DB [60] database used for storing metadata was implicitly synchronizing using the *fsync* system call during certain operations. This is most likely an artifact of the PVFS2 usage of the database API and would not normally be a notable issue. Unfortunately, these frequent fsync operations interact poorly with the EXT3 [84] file system used on Jazz. EXT3 by default uses an ordered data journaling mode. This means that any write synchronization forces all previous writes to be committed to disk in order first. The fsync call on EXT3 is therefore quite expensive on an active file system. To avoid this unintentional overhead, we entirely disable this function on all PVFS2 servers used in this study.

Finally, client side PVFS2 specific caching may have an impact on benchmark performance. In particular, PVFS2 has the ability to cache both name space information and simple attributes within the client library for short periods of time. We

disable both of these caches in order to better measure the file system level impact of the optimizations presented in this study.

5.2 Modeling Basics

Several general purpose network and server access patterns are reused frequently for different operations in PVFS2. It is therefore helpful to begin the case study discussion by developing models for these access patterns so that they can then be referenced later when analyzing specific PVFS2 operations. We will begin with the simplest network patterns and gradually expand the model to cover more complex cases.

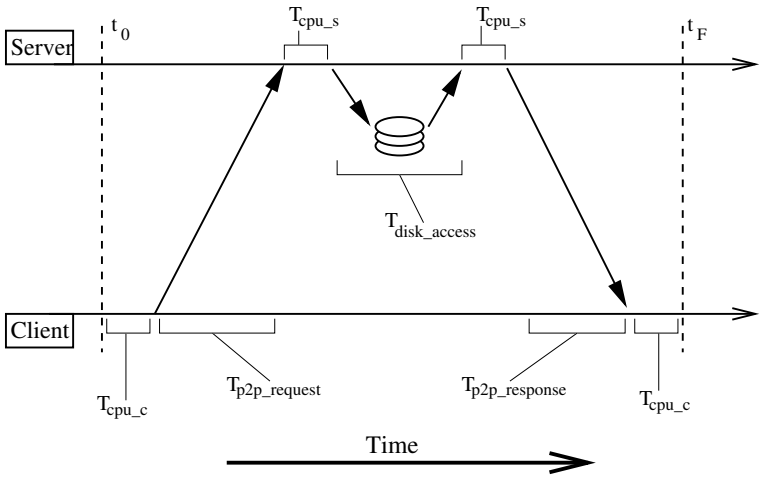
5.2.1 Single Metadata Operation

The first common PVFS2 access pattern occurs when a single client contacts a single server and requests that a service be performed. The server steps generally consist of a trivial computation followed by a read or write to a local database or file. The communication steps consist of two point-to-point network operations: one to send the request and one to receive the response. None of these steps overlap; they must be performed in series. We can therefore represent the cost (T) as:

$$T_{meta\_op} = T_{p2p\_request} + T_{cpu\_s} + T_{meta\_access} + T_{p2p\_response} + T_{cpu\_c} \qquad (5.1)$$

Figure 5.1 shows these costs in an operational diagram over time. T_{cpu_c} and T_{cpu_s} represent the computation overhead at the client and server, respectively. These values are generally small relative to the other terms and can be treated as a constant value which includes thread latency, state machine startup time, and function call overhead. The T_{meta_access} term refers to the cost of the actual service that the server will perform. The T_{p2p} terms can be modeled using the cost model from

Figure 5.1: Single metadata operation costs



Thakur and Gropp's work [79] on collective communication for switched networks, found in equation 2.1 of section 2.1. If we let n be the average size in bytes of the request and response messages, then equation 5.1 becomes:

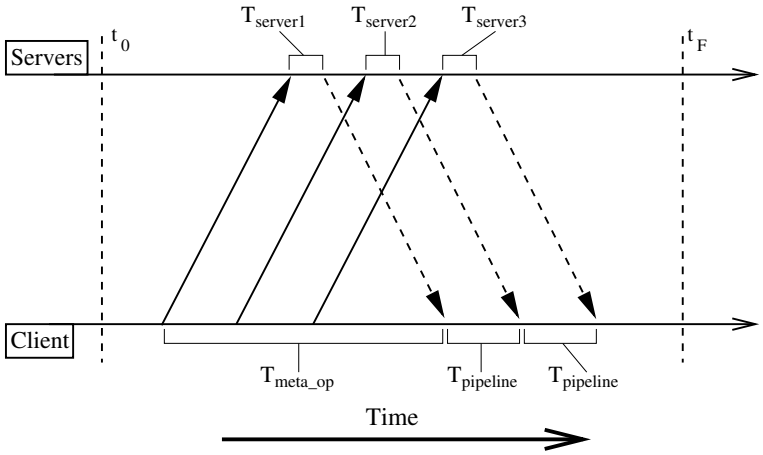
$$T_{meta\_op} = 2\alpha + 2n\beta + T_{cpu\_s} + T_{meta\_access} + T_{cpu\_c} \tag{5.2}$$

As in the collective communication literature, α and β respectively represent the message startup cost and message transfer cost per byte.

5.2.2 Concurrent Metadata Operations

Another common PVFS2 pattern occurs when multiple concurrent requests are issued from a single client. In most cases each request is sent to a different server. We can build a model for this example using the single metadata operation model from the previous subsection as a starting point. Figure 5.2 illustrates how the concurrent operations occur over time from the client's perspective. Note that after the first operation, subsequent requests are pipelined to achieve as much concurrency as possible.

Figure 5.2: Concurrent metadata operation costs



From a high level, we can represent the total cost as:

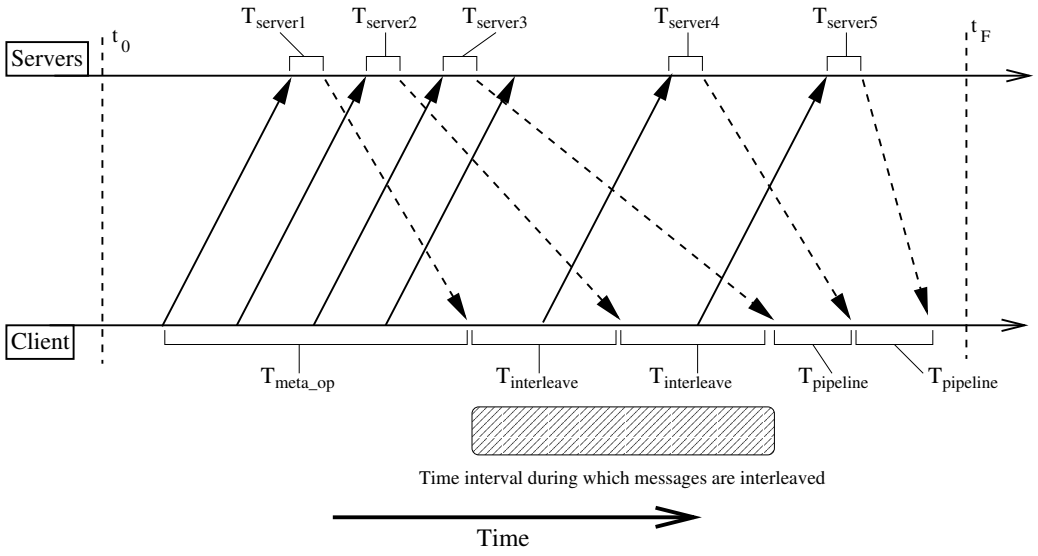
$$T_{meta\_concurrent} = T_{meta\_op} + (P-1)\,T_{pipeline} + T_{cpu\_c} \tag{5.3}$$

T_{meta_op} and T_{cpu_c} were defined in the preceding subsection. P represents the number of servers that the client is communicating with. At first glance, it appears that $T_{pipeline}$ would simply correspond to the message startup cost, α. However, it turns out that this is not the case; α does not take into account pipelining effects that occur when several messages are transmitted sequentially. In practice, the cost of α is not paid exclusively by the sending party. Roughly half of that cost is absorbed on the receive side, including overheads such as interrupt processing and application task switching. Thus, the $T_{pipeline}$ is more closely approximated by $\alpha/2$.

If we take these elements into account, and substitute in the single metadata operation model, we arrive at the following equation:

$$T_{meta\_concurrent} = 2\alpha + 2n\beta + T_{cpu\_s} + T_{cpu\_c} + T_{meta\_access} + (P-1)\left(\frac{\alpha}{2}\right) \tag{5.4}$$

Figure 5.3: Concurrent metadata operation costs with interleaving

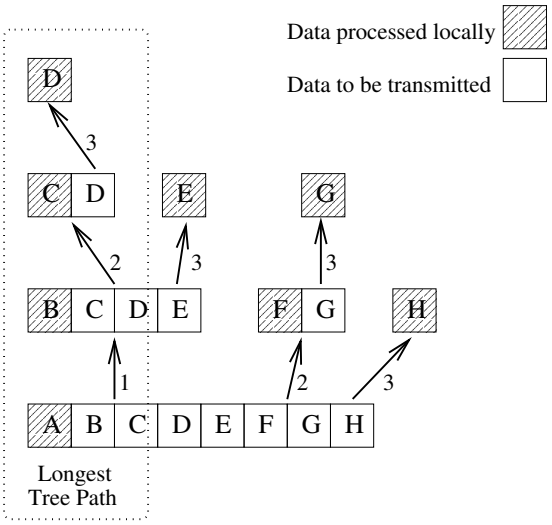


Equation 5.4 correctly models the communication costs for the pattern seen in figure 5.2. However, there were only a relatively small number of messages involved in that example. As the number of messages increases, the sending of requests begins to interleave with the the reception of responses, as shown in figure 5.3. Despite the fact that most modern networks are full-duplex, the startup cost per message, whether sending or receiving, must still be serialized. The messaging rate is therefore effectively cut in half during the time in which the messages are interleaved. We represent the message interval at this point as $T_{interleave}$, which corresponds directly to the value of α. Equations 5.5 and 5.6 can be used calculate the number of messages that are interleaved verses the number of messages that are simply pipelined:

$$N_{pipelined} = \min\left(\left(\frac{T_{meta\_op}}{T_{msg\_overhead} + \frac{\alpha}{2}} + 1\right), (P-1)\right) \tag{5.5}$$

$$N_{interleaved} = P - N_{pipelined} - 1 \tag{5.6}$$

Figure 5.4: Aggregate message pattern



Thus our final cost model for concurrent metadata operations is shown in equation 5.7. The terms of the equation are left in their most general form here for the sake of brevity:

$$T_{meta\_concurrent} = T_{meta\_op} + N_{pipelined}\,T_{pipeline} + N_{interleaved}\,T_{interleave} + T_{cpu\_c} \quad (5.7)$$

5.2.3 Aggregate Concurrent Metadata Operations

In section 5.2.2 we outlined a cost model for performing concurrent requests when each of the requests are sent from the same client. An alternative to this pattern is to use a collective algorithm such as the binary tree outlined in section 2.1. In this case, the total amount of work to be performed by servers is the same, but the communication pattern and cost is quite different.

Figure 5.4 shows an example communication pattern for an aggregate concurrent operation involving eight servers. Each server services its own portion of the work (the shaded unit) while simultaneously forwarding the next set of work to another

server. The arrows represent communication and are labeled with a step number. The work units are labeled with letters that represent where the work should be performed. In this example, three total steps are necessary to complete the pattern. Also notice that the size of the messages to be transmitted is smaller in the later steps, because those messages contain less information about where the remaining data must be forwarded.

The number of steps necessary to complete the operation can be represented as $\lfloor \log_2(P) \rfloor$, where P is the number of hosts involved. This expression along with the communication models from previous sections forms the basis for the aggregate concurrent operation cost. We only need to model the cost of the longest path through the communication tree because it places an upper bound on the cost of all concurrent paths. However, there are a two new cost components to take into account. The first is the computation cost of routing the messages. Each server must interpret an incoming request, decide how to route the remaining messages, and construct new requests based on that decision. We will call this cost T_{route}; its value will depend upon the processing speed of the system. The second additional cost is the CPU time required to reduce responses at each stage of the communication, assuming that a binary tree is likewise used to gather the results. We will refer to this cost as γ; it will be dependent upon the CPU speed of the system and the complexity of the operation in question.

Unfortunately, γ is not a fixed value per server or per step of the communication. It depends upon the number of units of data that must be reduced at each step. We could therefore choose a coefficient expression for γ that reflects the total number of data units transfered. However, we will take a slightly more thorough approach and break the CPU cost per communication step out into a summation that essentially results in the creation of a separate term for each step. This is significant because we

will later show how this summation can be reused in a more sophisticated network cost model as well.

If we let k be the communication step number, we can represent the number of message units at each communication step with the following expression:

$$\sum_{k=1}^{\lfloor \log_2(P) \rfloor} \left(\left\lceil \frac{P - 2^k + 1}{2^k} \right\rceil \right) \tag{5.8}$$

At this point we can construct a complete model for an aggregate metadata operation that combines the following components: network cost, routing overhead, reduction costs, metadata access, and the network pattern.

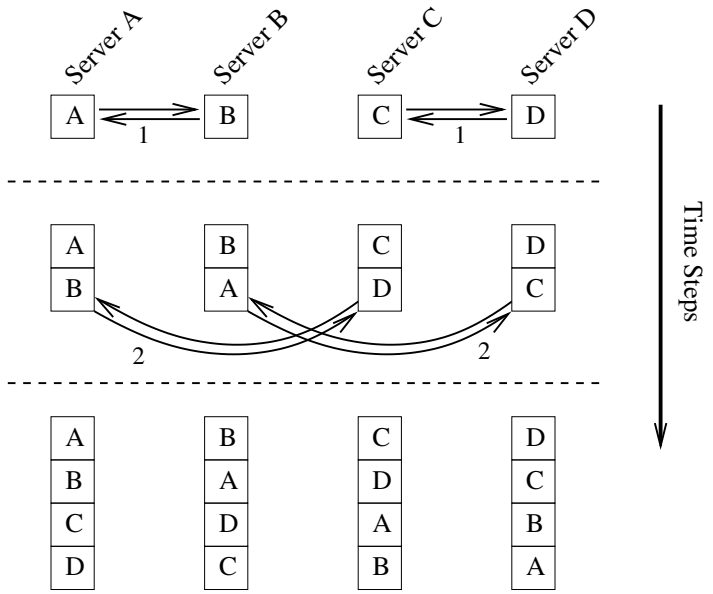
$$\begin{aligned} T_{meta\_agg} = (\lfloor \log_2(P) \rfloor)(2\alpha + 2n\beta + T_{cpu\_c} + T_{cpu\_s}) + \\ P(T_{route}) + T_{meta\_access} + \\ \sum_{k=1}^{\lfloor \log_2(P) \rfloor} \left(\left\lceil \frac{P - 2^k + 1}{2^k} \right\rceil \times \gamma \right) \end{aligned} \tag{5.9}$$

Note that the computational overhead associated with client operations in previous models is now factored into the server cost at each step of this model. Each forwarding server is acting as a client to another server and must therefore absorb the same messaging setup cost that a client process would bear in a conventional algorithm.

5.2.4 Server Status Composition

In section 4.3 we defined a *server status composition* operation in which servers perform an all to all communication operation to efficiently exchange global state information. This state information has a direct impact on the cost of the *statfs* file system operation, but is perhaps most interesting as an enabling technology for more advanced file system features such as load balancing, fault tolerance, or health mon-

Figure 5.5: Server status composition network pattern



itoring. However, the server status composition is never triggered as a direct result of an individual file system operation. Instead, it runs autonomously as a periodic background service for active servers and never factors directly into the time cost of a file system operation. In this subsection we will therefore focus on the network utilization cost of the server status composition rather than the time cost. Network bandwidth is the most significant resource that will be consumed by this operation.

Figure 5.5 shows the network pattern resulting from an example four server status composition operation. As in previous examples, the data from each server is designated with a letter. Arrows representing communication are labeled with numbers that correspond to the operational step in which they occur. The total number of steps is bounded by $\lceil \log_2(P) \rceil$. Note that the messages become larger at each step and are exchanged bidirectionally between peers. The payload of each message consists of a fixed sized portion of size n and a variable sized portion in increments of n_1. In this case, n_1 corresponds to the size of the status information for a single server plus the size of a handle that identifies it. We can calculate the

maximum amount of data transmitted (sent or received) per server using the following equation:

$$n_{max\_per\_server} = n(2\lceil \log_2(P) \rceil) + n_1(2^{\lceil \log_2(P) \rceil}) \tag{5.10}$$

This equation also holds for cases in which P is not an exact power of two. In those cases, the amount of data exchanged by some servers will actually be reduced but the upper bound will not change. We will eventually revisit this equation to aid in evaluation of the impact of operations that rely on the server status composition.

5.3 Model Extensions

The previous section outlined the principle components of models that will be used when describing a variety of file system operations. However, several subtle extensions can be made to these models. The importance of these extensions was not evident in some cases until investigation of the differences between the basic component models and real world behavior. We will now describe some of these extensions, with examples where appropriate, in order to clarify complete operation models that will be outlined later in this study.

5.3.1 Additional Message Startup Cost for Myrinet

Early on in experimentation, it was discovered that the *alpha* value measured for Myrinet networks in stand alone benchmarks did not translate well into use with PVFS2 operation models. Myrinet has a much lower message startup cost than TCP/IP. This message startup cost is low enough that it is dominated by the user level preparation required to send a message in PVFS2. For example, sending a request from a PVFS2 client actually consists of four steps: allocating a communication buffer, encoding the request, preposting a receive for the acknowledgment, and then

actually sending the message. Raw benchmarks do not induce this much overhead per message. Some of the overhead is accounted for by way of the T_{cpu_c} term in the models previously defined. However, the slower speed of the Ethernet network allows more of this cost to be overlapped and amortized with adjacent communication than can be done with the Myrinet network. We must therefore compensate by adding an additional term to Myrinet startup costs.

We define α_{prep} as the message preparation overhead associated with each communication. For Ethernet networks this term will be set to zero. For Myrinet or similar networks it will be set to a value that represent the message preparation cost that is not hidden by the network latency.

5.3.2 Network Transfer Time

Thus far in PVFS2 cost model development we have assumed that the communication transfer cost per byte (β) is a fixed value. However, this is not the case in practice. Most networks gradually ramp up to a maximum bandwidth figure as the size of the transmitted messages is increased. As a result, smaller messages are more expensive to transfer *per byte* than large messages are, even after taking into account the message startup time (α). This trend will play a significant roll in collective operations where a variety of message sizes may be used depending on the communication step.

To account for this effect, we need to characterize the bandwidth verses message size performance of the system network and incorporate it into the PVFS2 models. One way to do this would be to measure the transfer cost per byte for every potential message size, and then use the data as a lookup table for use in the cost models. Instead, we have chosen to take a large number of sample measurements and fit a more convenient formula to the data.

Figure 5.6 shows the result of this process for the Myrinet network on Jazz. 512 bandwidth samples were gathered using a point to point benchmark. A quadratic

Figure 5.6: GM/Myrinet bandwidth and model

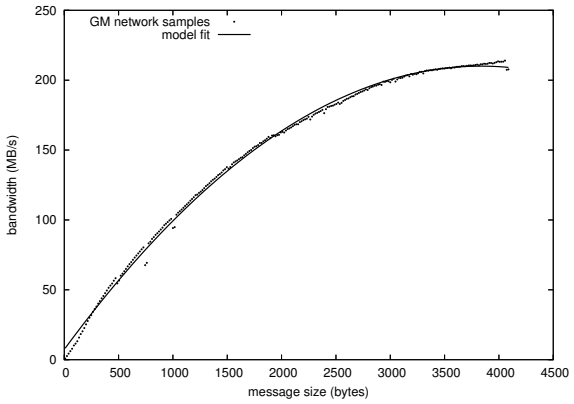
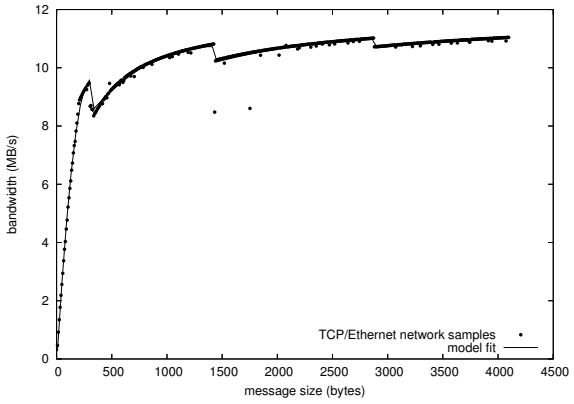


Figure 5.7: TCP/Ethernet bandwidth and model



equation was then fit to these samples using a nonlinear least-squares (NLLS) Marquardt-Levenberg algorithm [31]. This provides us with the following model for the Myrinet bandwidth:

$$BW_{Myrinet}(x) = 3.75 \times 10^{-05} x^2 + 0.109x - 0.0275 \tag{5.11}$$

The TCP/IP protocol over an Ethernet network posed a more challenging problem. The bandwidth samples and quadratic fit for this network are shown in figure 5.7. As before, a nonlinear least-squares (NLLS) Marquardt-Levenberg algorithm was used to fit the samples. However, the bandwidth curve is not smooth, due to

factors such as MTU size boundaries [28]. We have therefore broken the model into different formulas depending upon the message size range:

$$BW_{Ethernet}(x) = \begin{cases} .468 & : \quad x < 50 \\ -0.000105x^2 + 0.0640x - 0.233 & : \quad 50 \leq x < 300 \\ -1.96 \times 10^{-6}x^2 + 0.00541x + 6.99 & : \quad 300 \leq x < 1428 \\ -2.88 \times 10^{-7}x^2 + 0.00180x + 8.20 & : \quad 1428 \leq x < 2880 \\ -5.46 \times 10^{-8}x^2 + 0.000640x + 9.33 & : \quad 2880 \leq x < 4096 \\ 11.21 & : \quad x \geq 4096 \end{cases} \tag{5.12}$$

With the network bandwidth for Myrinet and Ethernet approximated by these models, we can now compute β using equation 5.13. Note that the extra 1024^2 factor is added to compensate for the fact that the bandwidth has been modeled in MB/s rather than bytes/s. Equation 5.13 can be substituted as appropriate in any of the preceding cost models where a β term has been used.

$$\beta(x) = x\frac{1}{BW(x) \times 1024^2} \tag{5.13}$$

Now that we have a more precise means to quantify the cost per byte of a communication step, we can revisit the aggregate metadata operation model of section 5.2.3. In that model we expressed the β component as shown below, with n representing the average message size:

$$\lfloor \log_2(P) \rfloor (2n\beta) \tag{5.14}$$

However, we should now take the exact size of each message into account rather than using an average n value. Most request or responses in an aggregate operation actually consist of both a fixed size component (header information) plus

some variable sized component. We will represent the fixed sized components of requests and responses as n_{req} and n_{resp} respectively, and the corresponding increments of variable data as n_1 and n_2. If we leverage equation 5.8 for the number of work units included with each message, we can express the complete message transmission cost per byte for a binary request and response as follows:

$$\sum_{k=1}^{\lfloor \log_2(P) \rfloor} \beta \left(\left\lceil \frac{P - 2^k + 1}{2^k} \right\rceil \times n_1 + n_{req} \right) + \sum_{k=1}^{\lfloor \log_2(P) \rfloor} \beta \left(\left\lceil \frac{P - 2^k + 1}{2^k} \right\rceil \times n_2 + n_{resp} \right) \tag{5.15}$$

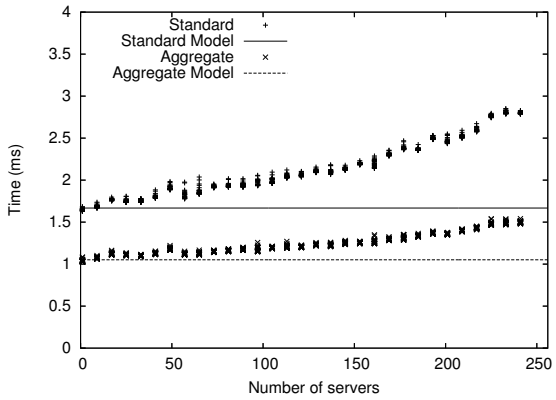
The use of a summation is critical here to account for the fact that the value of $\beta(n)$ as described above is a function of the message size at each step. We will incorporate this full expression in future PVFS2 operation models.

5.3.3 Inactive Connection Overhead

All benchmarks presented in this chapter begin with a setup phase in which each server contacts every other server in the file system with a test message. The purpose of this communication is twofold: to verify all-to-all connectivity, and to initiate any underlying network connections so that connection startup cost will not impact later measurements.

An unexpected side effect of this practice is that TCP/IP messaging was significantly impacted by the number of open socket connections at each server, regardless of whether the sockets were actually in use or not. Figure 5.8 gives a clear example of this phenomenon. The x axis shows the number of servers in the system while the y axis shows the time consumed by a PVFS2 directory creation operation. The file system in this example has only one metadata server. Thus only one server is contacted at each data point regardless of the size of the file system. The simple

Figure 5.8: Inactive connection impact on mkdir: Jazz TCP/Ethernet



models shown in this case assume that the performance will stay the same as more data servers are added. However, we can see that this is clearly not the case. The details of the base model used here will be covered in depth later.

By measuring the slope of the line and comparing it to the number of network messages required for each operation, we can estimate how much overhead has been added to each round trip message due to inactive network connections. We will refer to this value as $T_{inactive}$ and incorporate it into future models. This trend is not present on Myrinet networks, which are not connection oriented. We will therefore set $T_{inactive}$ to zero in Myrinet models.

5.3.4 Active Connection Overhead

In addition to the inactive connection overhead previously described, TCP/IP also exhibits more serious scalability problems as the number of *active* connections is increased. This is evident on large scale concurrent metadata operations in PVFS2 in which a single client must simultaneously communicate with many servers. The overhead becomes significant once approximately 48 connections are active but does not scale linearly with the number of connections.

Figure 5.9: Active connection impact on create: Jazz TCP/Ethernet

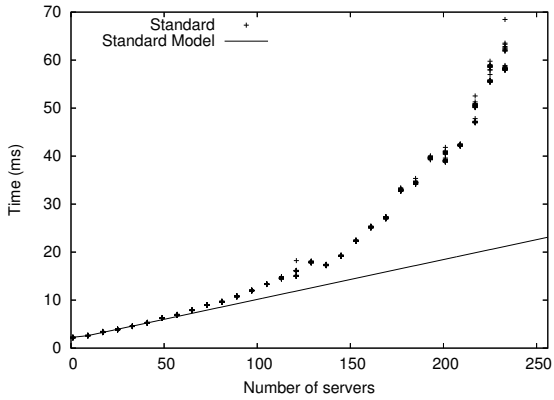


Figure 5.10: Poll() system call scalability: Jazz

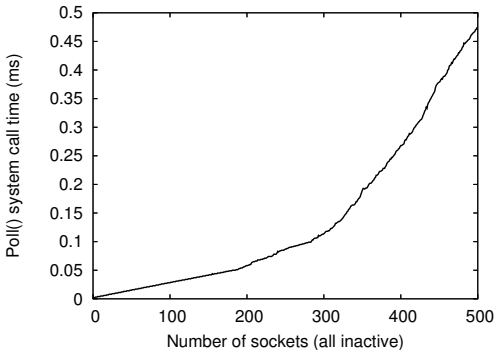


Figure 5.11: Create model residual and curve fit: Jazz TCP/Ethernet

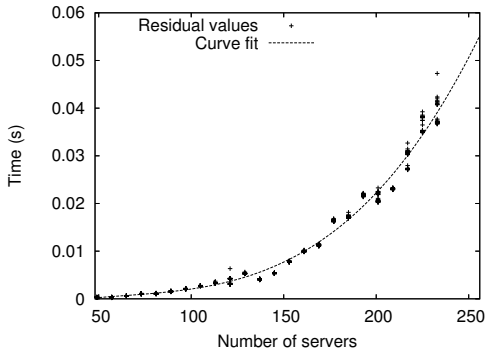


Figure 5.9 shows an example of this behavior. In this case, a file creation model that does not take the active connection overhead into account is shown in contrast to experimental samples for file creation time. There is a clear exponential trend present once more than 48 servers are active.

The source of this overhead is the poor scalability of the poll() system call [45] [17]. Poll() is used to scan a set of active sockets in order to wait for events which must be processed [41]. We can isolate this problem using an artificial benchmark which opens a large number of sockets and then measures the amount of time needed to complete a poll() operation on those sockets. Figure 5.10 shows the result of this test. None of the sockets were connected; they were simply allocated and added to

the poll set. Even with the sockets being completely idle, the call gets progressively more expensive as more sockets are added and eventually shows a nonlinear scalability trend. However, we cannot easily translate the behavior of this benchmark into model parameters. The impact of poll() scalability depends upon the duration of the test and message activity.

In order to model the impact of this scalability trend we will apply a nonlinear least-squares (NLLS) Marquardt-Levenberg fit [31] to the residual values gathered from the results in figure 5.9. The result of this process is shown plotted against the residuals themselves in figure 5.11. The corresponding polynomial function is given below:

$$T_{poll}(P) = 6.68 \times 10^{-9} P^3 - 1.23 \times 10^{-6} P^2 + 1.04 \times 10^{-4} P - 2.68 \times 10^{-3} \quad (5.16)$$

We will show later that this equation does a reasonable job of approximating the poll() overhead in concurrent PVFS2 operations over Ethernet. However, the equation we have given above is specific to file creation. We must scale the result of $T_{poll}(P)$ to an appropriate level depending on the relative runtime of the operation that we wish to apply it to. This can be done by multiplying $T_{poll}(P)$ by a scaling factor, F_{pscale}. F_{pscale} can be computed as the ratio of the projected runtime of the target operation divided by the runtime of the create operation.

$T_{poll}(P)$ is the only model parameter used in this study that cannot be determined through small scale benchmarking or trace examination. Further investigation would be required to develop a method for generating this model component directly. Newer Linux systems will most likely not suffer from this component, however, due to PVFS2's ability to utilize the epoll() interface available in modern Linux kernels [51].

This interface is meant to be a non-portable replacement for poll() which does not suffer from scalability difficulty as the number of monitored sockets is increased.

The Myrinet module suffers from a scalability problem as the number of concurrent messages is increased as well, though the effect here is much milder. This may be due the GM interface, or may be an artifact of the BMI implementation. It begins to appear when 110 or more servers are used. Since only 128 servers were available for the Myrinet experiments in this chapter, we do not have enough data points to determine the source of the overhead or model it with any certainty. Its presence will be noted in later experiments, however.

5.3.5 Metadata Access Costs

We have thus far described T_{meta_access} as the cost of the service performed by a server during the course of a file system operation. This cost will vary greatly depending on the operation. In the general case, we will always approximate this cost using a fixed value determined from an average of event log timings. T_{meta_access} will include any storage device (trove) accesses, as well as operation specific processing overhead on the server.

In all experiments we have configured the file system to avoid file synchronization as much as possible. For small metadata workloads, the servers will therefore seldom need to access the disk and the fixed parameter assumption approximation works well. Modeling cases in which the disk is plays a significant factor would require application of the disk modeling techniques discussed in section 2.5.

5.3.6 GM Computation Overlap Ratio

In section 5.2.3 we discussed the overhead involved in routing and reduction of collective communication in aggregate operations. These values are relatively easy to measure and incorporate into the models for TCP/IP Ethernet networks. The behav-

ior on GM Myrinet networks is more complicated, however. GM offloads much more communication work to the network cards than TCP/IP does. This in turn frees up more CPU time for other tasks. The trend becomes evident in aggregate operations in which a server is simultaneously processing messages, forwarding messages, and servicing local operations. In these cases, significant portions of the computation overhead can be overlapped with communication. This behavior is difficult to quantify directly and is dependent upon factors that are unique to each aggregate operation. This problem is exacerbated by the forked state machine optimizations described in section 4.4.

In Myrinet based models which include aggregate server processing (create, remove, and getattr), we will reduce the T_{route} and γ values by a ratio relative to the TCP/IP case: $F_{gm\_overlap}$. $F_{gm\_overlap}$ is not as accurate as we would like, and there is no heuristic way to determine its value for a given operation other than by observing scalability trends. We will nevertheless incorporate it into operation models until this GM/Myrinet behavior is better understood and a more advanced model can be constructed.

5.4 Model Parameters

In order to successfully apply the models developed in this chapter, we must first collect component parameters from real world systems to use as input. This section outlines each of the parameters that we intend to use along with a brief discussion of how they were acquired. The measurements presented in this section were gathered on the Jazz Linux cluster. All subsequent models in this chapter will use these values verbatim unless otherwise noted.

Table 5.1 shows the cost of various server side metadata operations. Each of these values was determined by analysis of trace files generated by the PVFS2 event logging facility, and includes both storage access and CPU costs.

Table 5.1: Jazz: metadata access costs

Parameter	Description	Value
T_GETATTR	get attributes	.000053
T_REMOVE	remove object	.000300
T_CREATE	create object	.000090
T_CRDIRENT	create directory entry	.000216
T_SETATTR	set attributes	.000577
T_RMDIRENT	remove directory entry	.000217
T_MKDIR	create directory	.000660
T_CHECK_DIR	check directory status	.000072
T_AGG_CRDIRENT	aggregate version of T_CRDIRENT	.000073
T_RMDIR	remove directory	.000575
T_CHECK_DIR2	check directory contents	.000190
T_STATFS	stat file system	.000030

Table 5.2: Jazz: network costs

Parameter	Network	Description	Value
α	GM	communication startup cost	10.8 μs
α	TCP	communication startup cost	82.3 μs
α_{prep}	GM	message preparation cost	20 μs
α_{prep}	TCP	message preparation cost	0 μs
$T_{inactive}$	GM	inactive connection overhead	0 μs
$T_{inactive}$	TCP	inactive connection overhead	1.4 μs

Table 5.3: Jazz: CPU costs

Parameter	Description	Value
T_{cpu_c}	client CPU overhead	27 μs
T_{cpu_s}	server CPU overhead	18 μs
T_{route}	aggregate routing cost	5.5 μs

Table 5.4: Jazz: operation specific costs

Parameter	Operation	Description	Value
γ	create	aggregate reduction cost	5 μs
γ	getattr	aggregate reduction cost	3 μs
F_{pscale}	create	poll overhead scaling factor	1
F_{pscale}	remove	poll overhead scaling factor	.929
F_{pscale}	getattr	poll overhead scaling factor	.85
$F_{gm\_overlap}$	create	Myrinet CPU overlap factor	.55
$F_{gm\_overlap}$	remove	Myrinet CPU overlap factor	.32
$F_{gm\_overlap}$	getattr	Myrinet CPU overlap factor	.32

Table 5.2 shows the model parameters related to network performance. In addition to these values, the $\beta()$ function defined in 5.3.2 will be used to characterize the network as well. The α values here were determined by way of independent BMI benchmarks. The α_{prep} value was determined through event logging. $T_{inactive}$ is described in section 5.3.3.

Table 5.3 gives generic parameters related to raw CPU performance of the system. Each of these was determined by analyzing PVFS2 event logs. Table 5.4 provides parameters that are unique to specific operations. The γ values were determined from event logs. The F values are described in sections 5.3.4 and 5.3.6.

Table 5.5 shows the disk parameters for Jazz, following the Hennessy and Patterson disk model given in the related work of section 2.5. T_{seek}, R, and $T_{c\_overhead}$ were all provided by manufacturer specification sheets [38] [55]. B was determined using the *hdparm* benchmarking tool [53]. Note that the hard disk data-sheet indicates a transfer rate of 48 MB/s for the IBM drives but this is apparently not achievable with the IDE controllers used on the experimental compute nodes. The average disk

Table 5.5: Jazz: disk access costs

Parameter	Description	Value
F_{seek_local}	seek locality factor	.27
B	disk transfer rate	29.91 MB/s
T_{seek}	average disk seek time	8.5 ms
R	disk rotational speed	7200 RPM
$T_{c\_overhead}$	disk controller overhead	4.17 ms
$size$	average disk access size	4 KB

accesses are normally much smaller than 4 KB for the metadata operations studied in this work. However, 4KB is the minimum block size that is normally read or written at one time with the combination of operating system and disk software used on Adenine. F_{seek_local} could not be directly measured with event logging, but was derived by solving for it in models for which all other terms were known. F_{seek_local} was found to coincide with the 25% to 30% value projected by Hennessy and Patterson [65].

5.4.1 Experimental Samples

We will compare the PVFS2 operation models to experimental samples from available systems in order to validate the model behavior. The experimental samples were generated using synthetic benchmarks that measure the elapsed time for execution of client side system interface functions. The number of servers (P) was used as the independent variable and varied from one to the maximum possible on each test system. We gathered 35 samples at each value of P. The first two samples in each set were discarded to avoid experimental noise due to startup costs. The remaining 33 samples were still sufficient to satisfy the Central Limit Theorem [52] and obtain an approximately normal shape for the distribution of the sample mean and standard deviation, assuming that we can approximate the experimental error using a normal distribution.

Examination of the data from these experiments reveals the appearance of outliers in almost all tests. These outliers occur due to system variability beyond benchmark control rather than from experimental error. Example sources of this type of variability include task switching from other system processes or disk activity from the virtual memory subsystem. We therefore choose to discard any extreme outliers from our statistical analysis. Extreme outliers are defined as sample values that are less than $(Q_1 - 3(IQR))$ or greater than $(Q_3 + 3(IQR))$ [61]. Q_1 and Q_3 represent the first and third quartile of the sample set respectively, while IQR represents the interquartile range. In some cases it was found that the computed value of IQR was too small relative to the magnitude of the measurements to be of any practical use. In those cases we set IQR to a minimum of 5μs for each sample set.

In the following sections we will compare the the modeled and measured data to evaluate the quality of the models. We will employ a variety of techniques to interpret the models depending on the situation. The first option is to simply plot the data and model so that they can be compared visually. This also gives some indication of the variability of the experimental data if the measurements are shown as a scatter plot. Secondly, we can compute the *coefficient of determination* (r^2) for the models [61]. The coefficient of determination indicates the proportionate reduction in error of the cost estimate attributed to the model, and can be defined as follows:

$$r^2 = \frac{SS(Total) - SS(Residual)}{SS(Total)} \tag{5.17}$$

A large r^2 value indicates that the model does a good job of predicting behavior. A small r^2 value indicates that the model is not relevant. SS(Residual) is the sum of the squared residuals, while SS(Total) is the sum of the total error squared. They can be computed as follows, with y_i representing the sample measurements, $\hat{y}_i$ the predicted values, and $\bar{y}$ the mean of all measurements:

$$SS(Total) = \sum (y_i - \bar{y})^2 \tag{5.18}$$

$$SS(Residual) = \sum (y_i - \hat{y}_i)^2 \tag{5.19}$$

In cases in which the output of the model does not vary greatly in response to the independent variable P, r^2 analysis will not be appropriate. In those cases we will simply calculate the percentage difference between the mean of the samples and the model output in regions of interest.

The final technique we will use in analyzing the models is to graph the residual values $(y_i - \hat{y}_i)$ themselves when appropriate. If a model does not fit the data well, then the residual plot can give an indication of the trend of the factor that is causing the discrepancy. This technique has already been demonstrated in the discussion of poll overhead in section 5.3.4.

5.5 Create Directory

The process of creating a new directory in PVFS2 is relatively straightforward. In the test environment used on Jazz, the number of data servers P is varied from 1 to the maximum number available (approximately 250 on Ethernet, 125 on Myrinet). However, in all cases only one of these servers acts as a metadata server.

The standard PVFS2 mkdir algorithm is described in section 3.5.4 and consists of three serialized steps: retrieving attributes for the parent directory, creating the new directory, and creating a new directory entry. We can therefore construct a model by summing the cost of these three round trip network operations plus the cost of each server component. The size of the requests and responses involved are given in table 5.6 and the resulting model is given below:

Table 5.6: Jazz: additional mkdir model parameters

Parameter	Description	Value
n_{get_req}	size of get attribute request	40
n_{get_resp}	size of get attribute response	64
n_{mkdir_req}	size of mkdir request	92
n_{mkdir_resp}	size of mkdir response	24
n_{dirent_req}	size of create directory entry request	76
n_{dirent_resp}	size of create directory entry response	16
n_{agg_req}	size of aggregate mkdir request	92
n_{agg_resp}	size of aggregate mkdir response	24

$$T_{mkdir} = 6(\alpha + \alpha_{prep}) + 3(T_{cpu\_c} + T_{cpu\_s} + T_{inactive}P) + \beta(n_{get\_req}) + \beta(n_{get\_resp}) + \beta(n_{mkdir\_req}) + \beta(n_{mkdir\_resp}) + \beta(n_{dirent\_req}) + \beta(n_{dirent\_resp}) + T\_GETATTR + T\_MKDIR + T\_CRDIRENT \quad (5.20)$$

The aggregate version of this operation is actually simpler in this case. By using intelligent servers we reduce the communication cost to a single network message. The server the carries out all steps of the operation and contacts other servers if necessary (though there is only one metadata server in this example). We can therefore construct a model for aggregate directory creation as follows:

$$T_{agg\_mkdir} = 2(\alpha + \alpha_{prep}) + T_{cpu\_c} + T_{cpu\_s} + T_{inactive}P + \beta(n_{agg\_req}) + \beta(n_{agg\_resp}) + T\_MKDIR + T\_AGG\_CRDIRENT + T\_CHECK\_DIR \quad (5.21)$$

Notice that in the aggregate case we have replaced $T\_GETATTR$ and $T\_CRDIRENT$ with T_CHECK_DIR and $T\_AGG\_CRDIRENT$ respectively. The intelligent server implementation actually performs fewer overall metadata steps to service the opera-

Figure 5.12: Mkdir performance: Jazz TCP/Ethernet

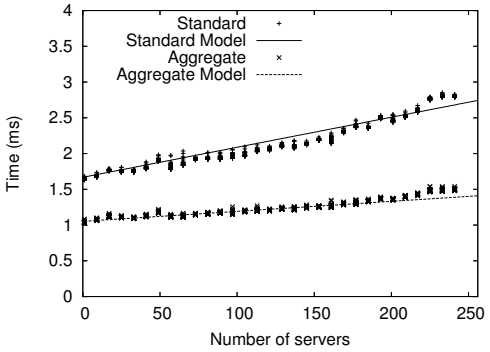
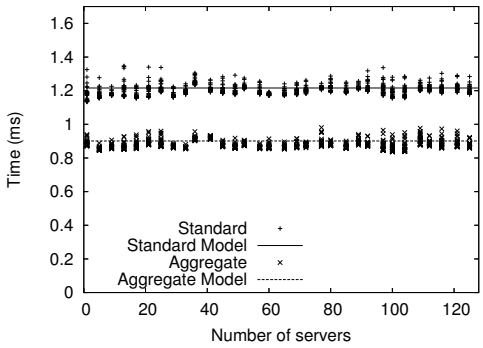


Figure 5.13: Mkdir performance: Jazz GM/Myrinet



tion. For example, the directory entry creation normally includes the cost of retrieving attributes in order to determine where to store the entries. In the aggregate case, the server can save the attributes retrieved in the first step and reuse them when creating the directory entry, therefore reducing overall metadata cost.

Figures 5.12 and 5.13 show a comparison of these models to experimental samples on the Jazz Linux cluster. As expected for such a simple operation, the model is very accurate. We cannot perform r^2 analysis because the models do not change significantly with respect to P. The TCP/IP performance only varies due to socket overhead as discussed in section 5.3.3. The models appear to be quite accurate however, and never deviate by more than approximately 6% from the sample mean value at any one point.

5.6 Remove Directory

The directory removal algorithms in PVFS2 are very similar to the directory creation algorithms given in the previous section. The only modification is that the steps performed by the server have changed in order to essentially "undo" the steps of the directory creation process.

Table 5.7 shows the additional parameters that will be employed to model directory removal cost. The model for the standard PVFS2 algorithm is given below:

Table 5.7: Jazz: additional rmdir model parameters

Parameter	Description	Value
n_{get_req}	size of get attribute request	40
n_{get_resp}	size of get attribute response	60
n_{rmdir_req}	size of rmdir request	36
n_{rmdir_resp}	size of rmdir response	16
$n_{rmdirent\_req}$	size of remove directory entry request	72
$n_{rmdirent\_resp}$	size of remove directory entry response	24
n_{agg_req}	size of aggregate rmdir request	72
n_{agg_resp}	size of aggregate rmdir response	16

Figure 5.14: Rmdir performance: Jazz TCP/Ethernet

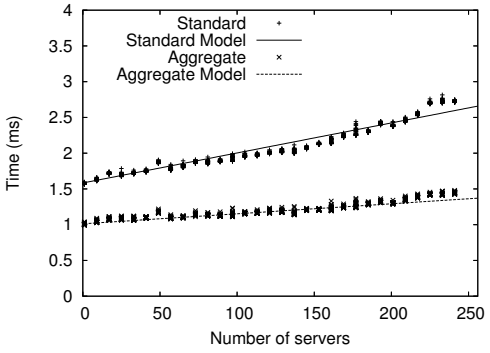
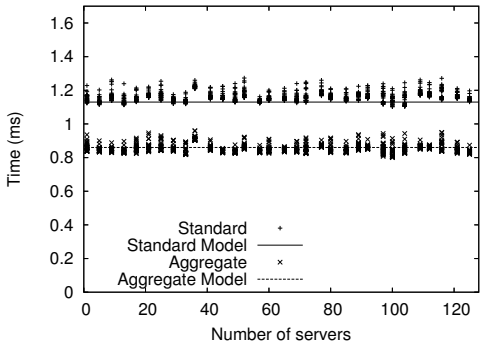


Figure 5.15: Rmdir performance: Jazz GM/Myrinet



$$T_{rmdir} = 6(\alpha + \alpha_{prep}) + 3(T_{cpu\_c} + T_{cpu\_s} + T_{inactive}P) + \beta(n_{get\_req}) + \beta(n_{get\_resp}) + \beta(n_{rmdir\_req}) + \beta(n_{rmdir\_resp}) + \beta(n_{rmdirent\_req}) + \beta(n_{rmdirent\_resp}) + T\_GETATTR + T\_RMDIR + T\_RMDIRENT \quad (5.22)$$

Similar modifications to the aggregate model result in the following equation:

$$T_{agg\_rmdir} = 2(\alpha + \alpha_{prep}) + T_{cpu\_c} + T_{cpu\_s} + T_{inactive}P + \beta(n_{agg\_req}) + \beta(n_{agg\_resp}) + T\_RMDIR + T\_RMDIRENT + T\_CHECK\_DIR2 \quad (5.23)$$

Table 5.8: Jazz: additional create model parameters

Parameter	Description	Value
n_{get_req}	size of getattr request	40
n_{get_resp}	size of getattr response	60
n_{create_req}	size of create request	52
n_{create_resp}	size of create response	24
n_{set_req}	size of setattr request	120
n_{set_resp}	size of setattr response	16
n_{dirent_req}	size of crdirent request	76
n_{dirent_resp}	size of crdierent response	16
n_{agg_req}	size of aggregate create request	124
n_{agg_resp}	size of aggregate create response	24
n_{rel_req}	size of relay create request	8
n_{rel_resp}	size of relay create response	42
n_1	incremental request size	8
n_2	incremental request size	24

Comparisons of the modeled and actual behavior on both Ethernet and Myrinet networks are given in figures 5.14 and 5.15. As in the directory creation case, the models track quite well, exhibiting no more than approximately 7% deviation from the mean value for any given data point.

5.7 Create File

Table 5.8 shows the request sizes of various messages involved in both the standard and aggregate create algorithm. The n_{rel_req} and n_{rel_resp} terms represent the server to server messages exchanged during the collective phase of the aggregate create.

The standard PVFS2 file creation algorithm is described in section 3.5.1. It consists of four serialized steps in addition to a concurrent metadata operation phase in which each of the datafiles is created. We can therefore construct a model for this operation by combining the cost of multiple serial operations (as in the mkdir and rmdir examples) with the cost of a concurrent metadata operation as given in section 5.2.2. The resulting file creation model is described by the following equation:

$$T_{create} = 10(\alpha + \alpha_{prep}) + 5(T_{cpu\_c} + T_{cpu\_s} + T_{inactive}P) + \beta(n_{get\_req}) + \beta(n_{get\_resp}) + 2\beta(n_{create\_req}) + 2\beta(n_{create\_resp}) + \beta(n_{set\_req} + Pn_1) + \beta(n_{set\_resp}) + \beta(n_{dirent\_req}) + \beta(n_{dirent\_resp}) + T\_GETATTR + 2\,T\_CREATE + T\_SETATTR + T\_CRDIRENT + N_{pipelined}\left(\frac{\alpha}{2} + \alpha_{prep}\right) + N_{interleaved}(\alpha + \alpha_{prep}) + F_{pscale}\,T_{poll}(P) \quad (5.24)$$

The create request costs are included twice to account for the separate and sequential metadata and data object creation phases. The set attribute request size is dependent upon the number of objects in the file and changes in increments of n_1 with respect to P.

The corresponding aggregate create model consists of a single client to server round trip communication, server metadata costs, and a collective request to create the file objects. The model can be constructed using the following two equations, separated for legibility:

$$T_{agg\_create} = 2(\alpha + \alpha_{prep}) + T_{cpu\_c} + T_{cpu\_s} + T_{inactive}P + \beta(n_{agg\_req}) + \beta(n_{agg\_resp}) + F_{gm\_overlap}\,T_{route}P + T\_CREATE + T\_CRDIRENT + T\_SETATTR + \lfloor log_2(P)\rfloor(2(\alpha + \alpha\_prep) + T_{cpu\_c} + T_{cpu\_c}) + \sum_{k=1}^{\lfloor \log_2(P)\rfloor}(T_{\beta\gamma}(k)) \quad (5.25)$$

$$T_{\beta\gamma}(k) = \beta\left(\left\lceil \frac{P - 2^k + 1}{2^k} \right\rceil n_2 + n_{rel\_req}\right) + \beta\left(\left\lceil \frac{P - 2^k + 1}{2^k} \right\rceil n_1 + n_{rel\_resp}\right)$$

Figure 5.16: Create performance: Jazz TCP/Ethernet

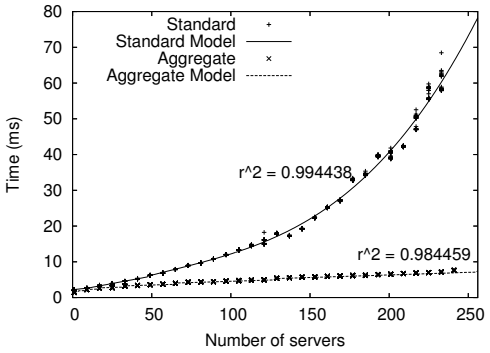
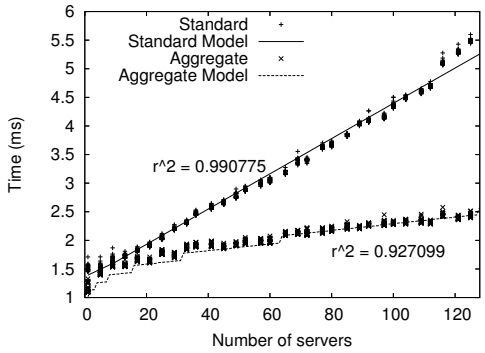


Figure 5.17: Create performance: Jazz GM/Myrinet



$$+\left\lceil \frac{P-2^k+1}{2^k} \right\rceil (F_{gm\_overlap}\, T_{reduce}) \quad (5.26)$$

The logarithmic network costs which replace linear terms from the standard model account for most of the efficiency improvement in this operation. The final summation terms are used to calculate the β costs which vary at each stage of the collective communication. An unexpected discovery was that when using TCP/IP, the PVFS2 servers completely failed to overlap computation with communication during the aggregate relay phase. We can compensate for this behavior by adding a $log_2(P)\,T\_CREATE$ term when modeling behavior on that network.

Figures 5.16 and 5.17 show the results of comparing our analytical model for file creation with empirical results on Jazz. In three of the four cases shown by these graphs, the models track extremely well with a coefficient of determination in excess of .99. The exception is the aggregate GM model, which does not match well until a large number of servers are involved in the operation. This suggests that the GM overlap interaction (described in section 5.3.6) requires further refinement at small scales. Also note that the GM model for the standard operation begins to skew at around 115 servers due to a BMI/GM scalability effect which has not been incorporated into this model. In both cases the aggregate algorithm does not show improvement unless greater than 16 servers are involved in the operation.

Table 5.9: Jazz: additional remove model parameters

Parameter	Description	Value
n_{get_req}	size of getattr request	40
n_{get_resp}	size of getattr response	64
n_{rem_req}	size of remove request	36
n_{rem_resp}	size of remove response	16
$n_{rmdirent\_req}$	size of rmdirent request	72
$n_{rmdirent\_resp}$	size of rmdirent response	24
n_1	incremental request size	8
n_{agg_req}	size of aggregate remove request	72
n_{agg_resp}	size of aggregate remove response	16
n_{rel_req}	size of relay remove request	36
n_{rel_resp}	size of relay remove response	24

5.8 Remove File

PVFS2 file removal must essentially "undo" the work of the file creation operation, and therefore has a similar number of steps. The algorithm is described fully in section 3.5.3. The message size parameters and equation for the standard file removal algorithm are given in table 5.9 and equation 5.27, respectively.

$$\begin{aligned} T_{remove} = 8(\alpha + \alpha_{prep}) + 4(\, T_{cpu\_c} + T_{cpu\_s} + T_{inactive} P) + \beta(n_{get\_req}) + \\ \beta(n_{get\_resp}) + 2\beta(n_{rem\_req}) + 2\beta(n_{rem\_resp}) + \beta(n_{rmdirent\_req}) + \\ \beta(n_{rmdirent\_resp}) + T\_GETATTR + 2\,T\_REMOVE + T\_RMDIRENT + \\ N_{pipelined}\left(\frac{\alpha}{2} + \alpha_{prep}\right) + N_{interleaved}(\alpha + \alpha_{prep}) + F_{pscale}\, T_{poll}(P) \end{aligned} \quad (5.27)$$

The model for the aggregate version of the remove option is shown in the next two equations. We have again provided the summation terms as a separate function for clarity.

Figure 5.18: Remove performance: Jazz TCP/Ethernet

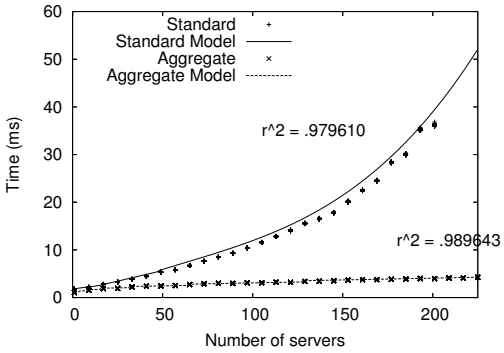
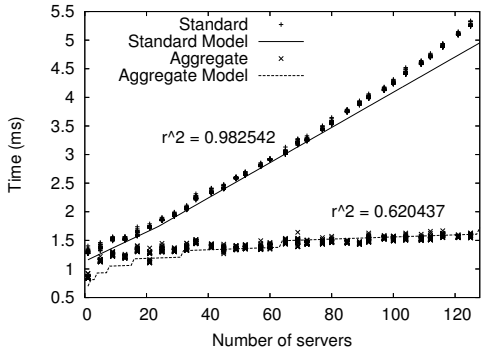


Figure 5.19: Remove performance: Jazz GM/Myrinet



$$T_{agg\_remove} = 2(\alpha + \alpha_{prep}) + T_{cpu\_c} + T_{cpu\_s} + T_{inactive}P +$$
$$\beta(n_{agg\_req}) + \beta(n_{agg\_resp}) +$$
$$F_{gm\_overlap}\,T_{route}P + T\_REMOVE + T\_RMDIRENT + T\_GETATTR +$$
$$\lfloor log_2(P) \rfloor (2(\alpha + \alpha\_prep) + T_{cpu\_c} + T_{cpu\_c}) + \sum_{k=1}^{\lfloor \log_2(P) \rfloor} (T_{\beta\gamma}(k)) \quad (5.28)$$

$$T_{\beta\gamma}(k) = \beta\left(\left\lceil \frac{P - 2^k + 1}{2^k} \right\rceil n_1 + n_{rel\_req}\right) + \beta\left(n_{rel\_resp}\right) \quad (5.29)$$

A noticeable difference in this case is that the relay responses do not grow larger at each step of the operation because there is no extra information per response to return as long as all operations are successful. This also means that there are no extra reduction costs. These factors combine to make $T_{\beta\gamma}$ simpler in this example than in the create example.

Figures 5.18 and 5.19 show the results of the file removal benchmark compared to the model for both Ethernet and Myrinet networks. In the Ethernet case, the largest data points for the standard algorithm have been discarded. Some system

Table 5.10: Jazz: additional getattr parameters

Parameter	Description	Value
n_{req}	size of getattr request	40
n_{resp_m}	size of getattr response (meta)	92
n_{resp_d}	size of getattr response (data)	68
$n_{agg_r eq}$	size aggregate getattr request	40
$n_{agg_r esp}$	size aggregate getattr response	104
$n_{rel_r eq}$	size of relay getattr request	24
$n_{rel_r esp}$	size of relay getattr response	32
n_1	incremental response size	16
n_2	incremental response size	8

variability not related to the algorithm itself caused the cost of several samples in that range to jump to as much as 15 times their normal value. Resource constraints prevent recreation of the experiment to replace the measurements at this time.

As in the create results, we see the most significant modeling problems at small scale with aggregate operations over GM. We again attribute this to poor modeling of GM's overlap of communication with computation. The standard GM model again points out the uncaptured scalability effects of BMI/GM with large numbers of servers. The TCP/IP standard model slightly overestimates operation cost. Both figures demonstrate the logarithmic increase in cost for the aggregate operation in contrast to the linear increase in cost for the standard operation.

5.9 Get Attributes

The models for the PVFS2 getattr algorithms follow the precedent set by the create and remove models. In this case, however, there are fewer sequential steps. In the standard algorithm there is a single get attribute request for the metadata object and then a concurrent request phase to retrieve attributes from the data objects. Although each phase uses the same request type, the response sizes are different due

to the discrepancy in attribute sizes between meta and data objects. The model for the standard getattr algorithm is given below:

$$T_{getattr} = 4(\alpha + \alpha_{prep}) + 2(T_{cpu\_c} + T_{cpu\_s} + T_{inactive}P) + \beta(n_{req}) + \beta(n_{resp\_m}) + \beta(n_{resp\_d}) + 2\,T\_GETATTR + N_{pipelined}\left(\frac{\alpha}{2} + \alpha_{prep}\right) + N_{interleaved}(\alpha + \alpha_{prep}) + F_{pscale}\,T_{poll}(P) \quad (5.30)$$

The corresponding aggregate algorithm model is given in the following two equations:

$$T_{agg\_getattr} = 2(\alpha + \alpha_{prep}) + T_{cpu\_c} + T_{cpu\_s} + T_{inactive}P + \beta(n_{agg\_req}) + \beta(n_{agg\_resp}) + F_{gm\_overlap}\,T_{route}P + T\_GETATTR + \lfloor log_2(P)\rfloor(2(\alpha + \alpha\_prep) + T_{cpu\_c} + T_{cpu\_c}) + \sum_{k=1}^{\lfloor \log_2(P)\rfloor}(T_{\beta\gamma}(k)) \quad (5.31)$$

$$T_{\beta\gamma}(k) = \beta\left(\left\lceil\frac{P - 2^k + 1}{2^k}\right\rceil n_1 + n_{rel\_req}\right) + \beta\left(\left\lceil\frac{P - 2^k + 1}{2^k}\right\rceil n_2 + n_{rel\_resp}\right) + \left\lceil\frac{P - 2^k + 1}{2^k}\right\rceil (F_{gm\_overlap}\,T_{reduce}) \quad (5.32)$$

As in the create modeling case, we discovered that servers using TCP/IP were unable to overlap communication and computation cost effectively. We compensate for this behavior by adding a $log_2(P)\,T\_CREATE$ term when modeling behavior over TCP/IP.

Figure 5.20: Getattr performance: Jazz TCP/Ethernet

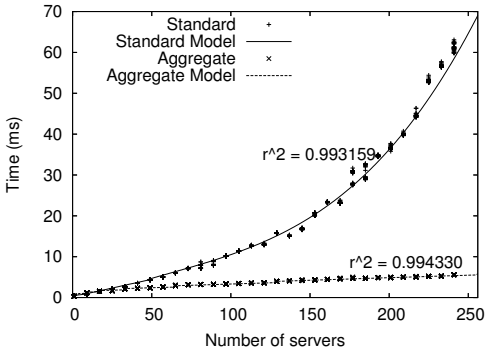
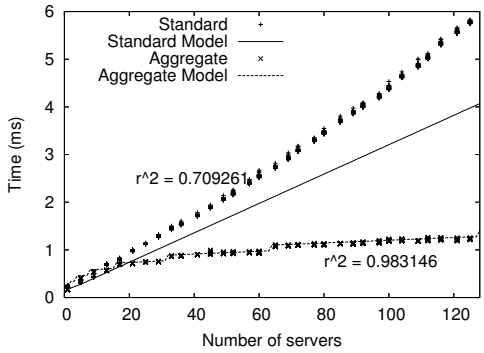


Figure 5.21: Getattr performance: Jazz GM/Myrinet



Figures 5.20 and 5.21 show the results of applying this model to the Jazz Linux cluster over both Ethernet and Myrinet. Both exhibit logarithmic increase in cost in the aggregate case compared to linear cost for the conventional case. As in previous cases, the coefficient of determination is quite high for TCP/IP examples. The surprising result in this case is that the GM model is accurate for the aggregate operation but underestimates the standard algorithm, which is contrary to the trend seen create and remove. The most significant difference in getattr compared to create and remove is that the server operation performed at each data server is very short and is composed almost exclusively of CPU overhead because of the write through attribute cache used at each server. This combination of brief server activity and lack of time spent waiting on peripherals most likely interferes with GM's ability to overlap communication. More work will be required to develop a model for this aspect of the system.

5.10 File System Status

Table 5.11 shows the request sizes of the protocol messages for both the standard and aggregate statfs operation. The parameters associated with the server to server communication (server status composition) that makes the aggregate statfs possible

Table 5.11: Jazz: additional statfs model parameters

Parameter	Description	Value
n_{statfs_req}	size of statfs request	28
n_{statfs_resp}	size of statfs response	100
n_{agg_req}	size of aggregate statfs request	28
n_{agg_resp}	size of aggregate statfs response	20
n_1	incremental request size	92

will be covered in the following section. For now we are only concerned with the immediate costs of the client to server operations.

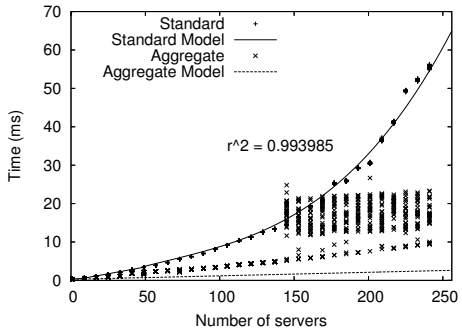
$$T_{statfs} = 2(\alpha + \alpha_{prep}) + T_{cpu\_c} + T_{cpu\_s} + T_{inactive}P + \beta(n_{req}) + \beta(n_{resp}) + T\_STATFS + N_{pipelined}\left(\frac{\alpha}{2} + \alpha_{prep}\right) + N_{interleaved}(\alpha + \alpha_{prep}) + F_{pscale}T_{poll}(P) \quad (5.33)$$

$$T_{agg\_statfs} = 2(\alpha + \alpha_{prep}) + T_{cpu\_c} + T_{cpu\_s} + T_{inactive}P + \beta(n_{agg\_req}) + \beta(n_{agg\_resp} + Pn_1) + T\_STATFS \quad (5.34)$$

Equations 5.33 and 5.34 show the cost models for the standard and aggregate statfs operations, respectively. The standard model is very similar to previous models for concurrent metadata operations, except that there are no serialized requests prior to the aggregate phase. The client begins immediately by issuing statfs requests to the relevant servers.

The aggregate statfs model is somewhat unique, because it consists of only a single request and response. This is made possible by the server status composition operation which insures that each server possesses a cached copy of the global statfs

Figure 5.22: Statfs performance: Jazz TCP/Ethernet



information in advance. There is no need to perform a collective or contact multiple servers to perform an aggregate statfs operation.

Several unexpected results were obtained from the statfs performance measurements on Jazz. First of all, the Myrinet network on Jazz was found to be unable to perform the server status composition operation at scale. The network pattern generated by the server status composition is much more demanding than that generated by the other algorithms used in this study, due to both the volume of concurrent activity and the bidirectional transfers between individual hosts. Several servers reported network failures in all test runs. Due to time constraints we were unable to investigate further. A simpler BMI based benchmark or experimentation with newer system software and drivers may be necessary to determine the problem. We have elected not to show any Myrinet results for statfs until the problem is resolved.

The Ethernet network did perform the server status composition operation successfully, however, and the results of the experiment and model output are shown in figure 5.22. Both cases exhibit linear increase in cost, but the aggregate version increases at a much slower rate. The standard operation and model agree quite well, with a coefficient of determination in excess of 99%. The aggregate results were not successful. There are in fact two trends of interest. The first, and most important, is that the aggregate cost as the number of servers increases grows much more quickly than expected (approximately an extra 20 μs per server). Further investigation of

Table 5.12: Server status composition parameters

Parameter	Description	Value
n_{req}	size of status composition message	36
n_1	incremental size of status information	92

trace files shows that the servers are maintaining consistent performance, but the client library is taking longer to receive and process the responses as they grow in size. We believe that this excess cost is due to an implementation performance flaw which can be corrected in future PVFS2 versions. We therefore will not account for it in the model.

The second unexpected trend in figure 5.22 is the extreme variability and higher average cost in the statfs operation beyond 140 servers. The unusual but consistent pattern of samples in this region suggests either the impact in this range of either a network topology characteristic or a programmatic error in the implementation. Unfortunately, the behavior cannot be replicated at small scales, and we do not at this time have trace files for the largest scale runs. Further investigation will be required with a similar allocation of processors on Jazz in order to determine if it is a network or implementation characteristic.

5.10.1 Network Utilization

The aggregate version of the statfs operation is made possible by a server status composition performed continuously by intelligent servers. As noted in section 4.3, the most significant cost of a server status composition is the network utilization.

Table 5.12 shows the sizes of the network messages involved in a server status composition iteration. If we combine this information with equation 5.10 from section 5.2.4, then we can compute the maximum amount of data transfered per server, per composition. The following table shows the result for a sampling of file system sizes:

Table 5.13: Server status composition network usage

Number of servers	Bytes transfered per server
1	0
16	1760
64	6320
256	24128
1024	94928

For example, if we were to perform a status composition twice per second on a 256 node system, then each server would sacrifice about 47 KB/s of network bandwidth. This is well less than one percent of the network capacity of even a 100 Mbit Ethernet link. We can therefore assume that this network load will not be a burden on the system as long as the update rate is not too aggressive.

5.11 Summary

We have analyzed and developed models for six key metadata operations in PVFS2. In the majority of cases the models were quite accurate. The most notable exceptions occurred on Myrinet networks where it is not fully understood yet how to model the overlap of communication with computation. However, these models provide a strong foundation for the principles necessary to model complex file system operations. In all cases the models would at least be sufficient to predict general trends and guide the choice of file system algorithms. We will later use these models to predict file system efficiency at scales much larger than those where PVFS2 is currently deployed.

CHAPTER 6
EVALUATION

Thus far we have identified five key obstacles to achieving scalability in next generation parallel file systems: efficiency, complexity, management, consistency, and fault tolerance. We have also described the implementation of intelligent servers and collective communication as a means to overcome those challenges. We will now evaluate how well each of the five challenges has been addressed by way of experimental results and case studies in a real world parallel file system.

6.1 Efficiency

Several factors contribute to the efficiency of a parallel file systems. The most significant factor is network overhead, which is simply the amount of time spent sending and receiving data. A secondary factor is the protocol efficiency, or how concisely the file system protocol describes the operations which servers must perform. Finally, computation overhead can play a role, particularly when results from many servers must be interpreted to determine the outcome of a file manipulation. These factors have been addressed somewhat by the general concept of parallel file systems and the optimizations afforded by I/O libraries, but they fail to address the metadata performance which becomes a limiting factor in usability at extreme scales. We will focus on metadata performance in this section.

The most significant of the three efficiency problems that we will focus on is the network overhead. This the limiting factor in performance for a variety of workloads, especially small latency bound operations. The network overhead can be reduced using a variety of techniques. One approach is to leverage more expensive special purpose networking hardware, such as Myrinet [7] or Quadrics [66]. Another

approach is to use lighter weight network protocols, such as Active Messages [87] or Gamma [18]. Further optimizations can even be applied at the abstraction level that allows the network to interface with the file system, regardless of the protocol. Many of these abstraction level optimizations are covered in greater detail in section 3.3.1 and in [12].

The above techniques are all useful for improving point-to-point messaging performance within a parallel file system. However, they do not take into account the *concurrent* network patterns that often occur within a parallel file system. In particular, the inherent parallelism of the file system resources typically results in a client or server communicating with several other hosts just to carry out an individual file system function. In a naive implementation, these network access patterns result in the sequential transmission and reception of a large number of messages. This sequential behavior at a single network hot-spot can result in a performance penalty regardless of the speed of the underlying network, due to factors such as startup costs, interrupt processing, and hardware limits on concurrent messaging. This problem is exacerbated in large scale systems as more and more hosts must be contacted to carry out each operation.

Figure 6.1 shows an example of a client which must contact seven servers in order to carry out a single file system operation. The network messages are shown as arrows which are numbered. This aggregate operation requires seven sequential network steps to complete. The number of sequential steps increases as more servers are added to the system; thus, performance will decrease as the file system grows.

This type of pattern can easily be optimized using collective communications, however. A binary tree algorithm as noted in related work in section 2.1 is one example of how this can be achieved. A tree algorithm divides the work of transmitting messages recursively amongst all hosts involved in communication. Figure 6.2 shows how the operation from the previous example could be performed using a binary

Figure 6.1: Example communication pattern for client requests (conventional)

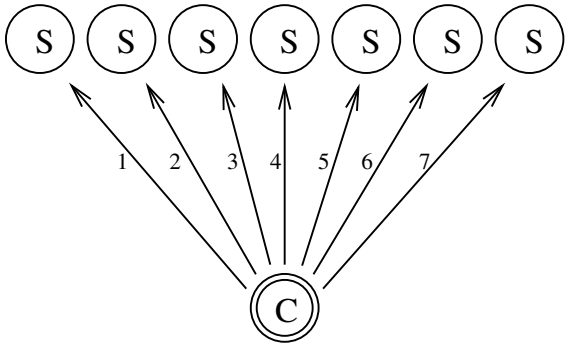
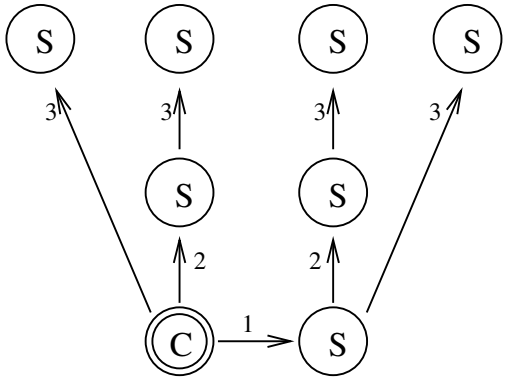


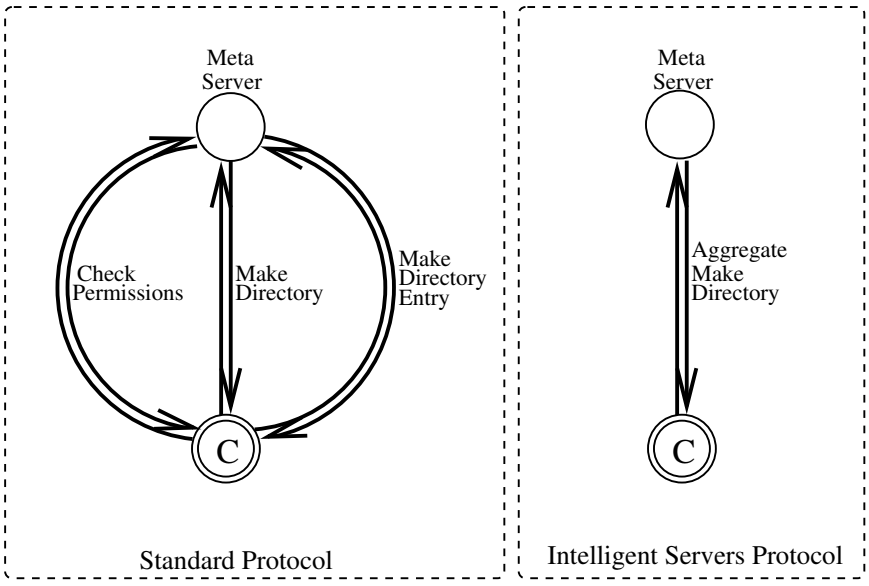
Figure 6.2: Example communication pattern for client requests (collective)



tree. The network steps are labeled once again, though this time only three steps are required because several communication stages occur in parallel. The number of network steps needed for this pattern is $O(\log_2 N)$, where N is the number of servers, rather than $O(N)$ for the naive approach. This will result in greater scalability for network bound parallel operations. The binary tree algorithm is not the only way to optimize this case; it is simply given for illustrative purposes.

In addition to the raw network performance, protocol efficiency plays a role in overall file system efficiency as well. With standard servers, several protocol messages may be required to carry out a single high level operation, even if only one server is involved. This is because the server does not understand the relationship between file

Figure 6.3: Intelligent servers protocol comparison



system objects or how to contact other servers should the need arise. It must instead rely on the client to independently orchestrate each step of the operation. With intelligent servers, we have the opportunity to use a much more concise protocol. The client can issue a single high level request to take the place of multiple small scale requests, thereby reducing excess protocol traffic. The server can interpret this high level request and carry out multiple local tasks to service it, or it can farm out tasks to other servers as needed. This is even more relevant in wide area or grid environments in which the inherent network delay exacerbates the impact of protocol efficiency.

Figure 6.3 shows an example of the protocol improvements that can be achieved by using intelligent servers. In this case we show the traffic between a single client and single metadata server as a directory is created. The standard protocol would require three requests: get attributes, make directory, and create directory entry. An intelligent server can carry out this entire multi-step operation with only a single protocol request.

Computational overhead is the final aspect of file system efficiency that has been addressed through the use of intelligent servers and collective communication. Computational costs incurred by the file system include composition of protocol messages, mapping operations to servers, and interpreting or summarizing results from servers. By using intelligent servers and distributing tasks throughout the file system, we have increased the amount or parallelism that can be achieved for this type of computation, and insured that the burden is not handled exclusively by the client.

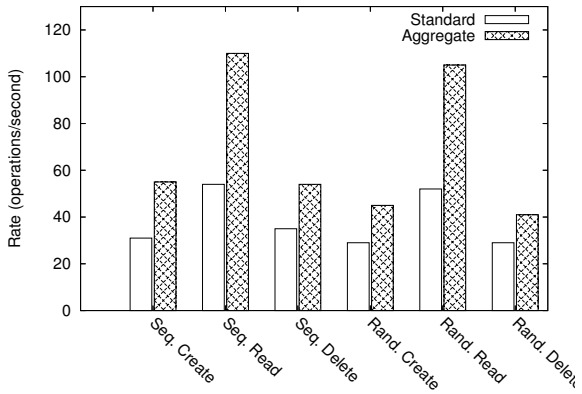
In PVFS2 we have implemented optimizations for six existing system operations: make directory, get attributes, create file, remove directory, stat file system, and remove file. Three of these utilize a binary tree to improve network efficiency, while one utilizes a recursive doubling algorithm to avoid network costs at operation time. Each one (and particularly the directory operations) benefits from a more concise protocol. Each one also benefits by varying degrees from a better distribution of computational cost. In the following subsections we will quantify the overall impact at the file system level resulting from these improvements in efficiency.

6.1.1 Benchmarks and Applications

This section will focus on the evaluation of various metadata intensive applications and benchmarks. All tests were carried out on the Adenine Linux cluster described in 5.1. The goal of this dissertation is to determine if intelligent servers and collective communication can be used to achieved scalability on much larger systems. However, only 74 servers were available for testing on Adenine. These examples should still provide some indication of the efficiency improvement that can be obtained even with relatively limited resources.

The first benchmark that we will examine is Bonnie++, by Russell Coker [19]. Bonnie++ consists of two phases; the first measures I/O throughput for various access patterns, while the second emphasizes metadata performance. We will only

Figure 6.4: Bonnie++ results: Adenine 74 servers



consider the latter phase at this time because the I/O path was not modified when implementing the PVFS2 extensions used in this text. Bonnie++ tests the creation, reading, and deletion of small files. It first creates a series of files in sequential numerical order, stats each file, then deletes each file. The tests are then repeated using random file names and a random ordering of each step.

Figure 6.4 shows the results of the Bonnie++ benchmark on PVFS2, both with and without aggregate optimizations. The y axis shows the number of operations per second achieved for each type of operation. The aggregate version achieves higher throughput in all cases, particularly with over 100% improvement in the file accesses which are dominated by stat() operation costs. The random delete operation shows the least improvement, though still significant at about 40%. The random delete operations are slowed by readdir() costs which have not been optimized in this study.

The second benchmark that we examine is Postmark, from Network Appliance, Inc [44]. Postmark is intended to measure the type of access patterns common to large scale Internet services such as mail, news, and web commerce. Despite widespread demand, parallel file systems have historically not excelled in these problem domains. The biggest impediment is is the latency associated with distributed metadata access. However, the use of intelligent servers and collective communication may enhance the

Table 6.1: Postmark results: Adenine 74 servers

10K Total transactions	PVFS2	PVFS2 Aggregate
Transactions per second	16	29
Data read (KB/s)	48.23	85.82
Data written (KB/s)	57.22	101.82

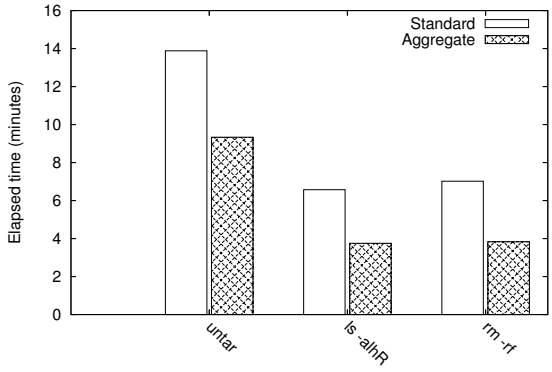
applicability of parallel file systems to a wider range of tasks such as these. Table 6.1 summarizes the results of the Postmark benchmark on Adenine. In this case, 10,000 transactions were performed on a set of 1,000 sample files. The transactions shown in this table consist of pairs of operations, such as create, delete, read, or append. We show results in terms of both the number of transactions per second and the amount of data read or written per second. The aggregate PVFS2 operations achieve at least a 75% improvement in all categories.

The final experiments that we will investigate center on the manipulation of the Linux kernel source package. Unpacking the kernel source (using the GNU tar command) is a familiar task for many Linux users. It often serves as an indicator of the interactive responsiveness of a file system. Unpacking the kernel source also happens to be a task that parallel file systems typically perform poorly, particularly in the absence of client side caching. In figure 6.5 we show the amount of time elapsed in minutes for unpacking, listing, then deleting the Linux kernel source on a 74 server system. Version 2.6.9 of the Linux kernel was used for this test. The source includes over 1,000 subdirectories and 16,000 files. If the costs of these three commands are added together, then we find that the total time is reduced by over 10 minutes in the aggregate case.

6.1.2 Projected Operation Performance

Although the preceding experiments have demonstrated promising efficiency even on relatively small file systems, we are ultimately most concerned with how well the

Figure 6.5: Kernel source tree manipulation: Adenine 74 servers



performance will scale for larger file systems. In particular we want to enable the use of parallel file systems on the next generation of massively parallel systems.

In chapter 5, we developed analytical models for several example parallel file system operations in PVFS2, and verified their accuracy for file systems with hundreds of servers. Four of the optimized operations exhibited significant improvement in scalability as the number of servers in the file system were increased: create, remove, get attributes, and file system status.

We will now utilize those models to predict how the example operations would perform in a hypothetical file system comprised of 1000 servers. Table 6.2 summarizes the results of this projection. Each row shows the estimated time to complete the example operation, both with and without aggregate optimizations. The final column indicates the percentage reduction in cost brought about by the optimizations. In each case we have chosen input parameters for the models based on the performance characteristics of the Jazz Linux cluster.

Each aggregate optimization is projected to result in at least a 75% reduction in individual operation cost. The most striking improvement is evident on the commodity TCP/IP over Ethernet network. In this environment, each operation is projected to grow to the unreasonable time cost of approximately 5 seconds per in-

Table 6.2: Projected PVFS2 performance: 1000 servers

Operation	Network	Standard	Aggregate	% Reduction
Create	TCP	5.634955	0.018645	99%
Remove	TCP	5.084923	0.010606	99%
Getattr	TCP	4.803568	0.015489	99%
Statfs	TCP	4.788267	.009486	99%
Create	GM	0.032144	0.007944	75%
Remove	GM	0.031838	0.003511	89%
Getattr	GM	0.030955	0.004107	87%
Statfs	GM	0.030897	0.000561	98%

dividual metadata operation. However, aggregate optimizations can reduce this by over 99%. We should also note that there is a possibility that the standard operation numbers are overly optimistic. The GM experiments in particular suggested an additional scalability penalty beyond the 100 server point which has not been captured by the model.

6.1.3 Wide Area Performance

As mentioned in section 6.1, improvements in protocol and network efficiency can also have a pronounced impact in wide area or grid environments. This can be quantified by duplicating the benchmark and application experiments in a grid environment. In particular, we will evaluate a scenario in which the file system and the client that accesses it are separated by a wide area network link. Unfortunately, such an environment was not available at the time of testing, largely due to security concerns on the test systems. However, it is possible to emulate the behavior of a wide area network with a simulator. We chose to use the Netem network simulator which is included in recent stock Linux kernel releases [37]. It builds on many ideas of the earlier NIST Net simulation tool [15] and allows artificial inducement of effects such as packet loss, delay, duplication, and reordering on any active Linux network interface.

Figure 6.6: Wide area simulation latency

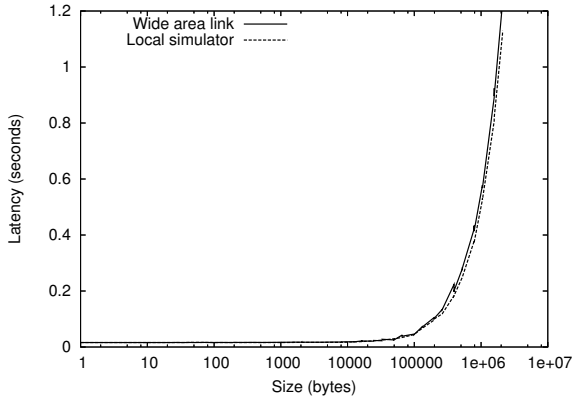


Figure 6.7: Wide area simulation bandwidth

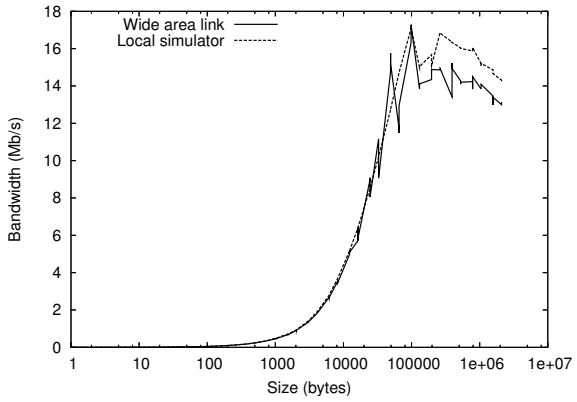
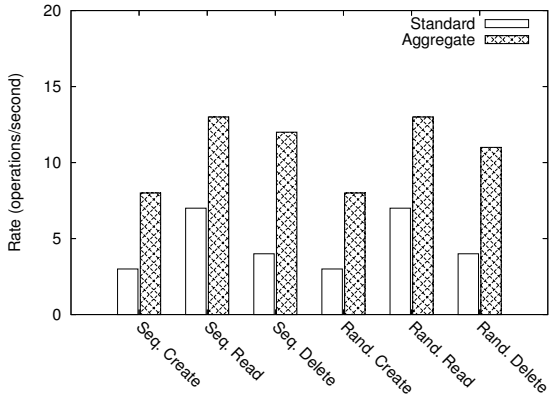


Figure 6.8: Bonnie++ WAN results: Adenine 72 servers



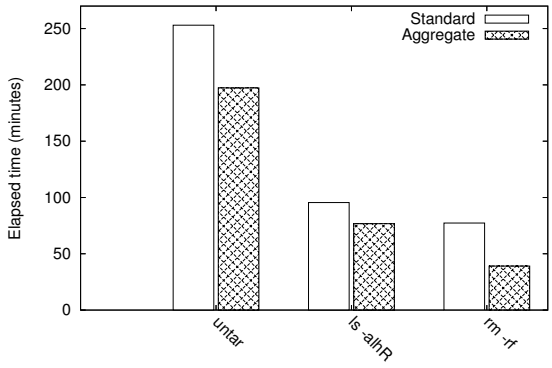
We chose to simulate the network connection between Clemson University and Argonne National Laboratory for these tests. Clemson and Argonne are linked primarily by the Abilene Internet2 network [29] which provides a high bandwidth connection for research purposes between 220 university and government research centers. We used the NetPipe network analysis tool [76] to characterize the network connection and choose appropriate simulator parameters. Figures 6.6 and 6.7 show a comparison of the actual network latency and bandwidth compared to the simulated network latency and bandwidth. The final settings consisted of a bitrate of 16.1 Mb/s with a buffer size of 7000 bytes and a maximum instantaneous burst buffer of 4000 bytes. The network delay was modeled as a 31.36 ms delay using a normal distribution with a variance of 50 μs. The simulated behavior closely resembles that of the real world network link, particularly in the latency domain. The bandwidth characteristics are overestimated slightly in the peak performance range.

Figure 6.8 shows the result of repeating the Bonnie++ experiment from the previous section in a simulated wide area environment. In this case, 72 servers were accessed by one client using the Netem tool. The aggregate version of PVFS2 demonstrates an improvement ranging from 70% to 300% in this environment. The delete and create operations are the most significant categories in this case. This is because

Table 6.3: Postmark WAN results: Adenine 72 servers

10K Total transactions	PVFS2	PVFS2 Aggregate
Transactions per second	1	2
Data read (KB/s)	4.47	7.88
Data written (KB/s)	5.31	9.34

Figure 6.9: Kernel source tree manipulation WAN: Adenine 72 servers



the standard delete and create operations consist of several sequential steps from the client, and therefore must traverse the high latency wide area link several times. The aggregate extensions use a more abstract protocol format that requires fewer messages.

Table 6.3 shows the result of repeating the Postmark benchmark of the previous section over a wide area network. The performance of this benchmark in all cases drops off more drastically than Bonnie++ when executed over a WAN. The aggregate optimizations still allow a significant advantage in data throughput. However, the transaction throughput has dropped to a point that is too slow in either case to warrant comparison.

Finally, figure 6.9 shows the results of repeating the kernel source manipulations of the previous section over a wide are network. This environment is clearly not conducive to interactive manipulation with PVFS2. However, the aggregate optimizations do offer improvement, reducing the total run time by nearly two hours.

Further ways in which intelligent servers and collective communication could enhance grid performance will be discussed in section 7.2.

6.2 Complexity

Parallel file systems are inherently complex due to the scope of services that they provide. This complexity is exacerbated in larger scale systems in which there are more resources to coordinate. Unfortunately, this trend towards increased complexity is difficult to mitigate at a system wide level. However, we can make design decisions that allow us to simplify specific components of the file system. In particular, the use of intelligent cooperative servers has the potentially to drastically reduce complexity of the client library implementation. Figure 6.10 shows an example of a hypothetical file system function that requires first communicating with a metadata server and then communicating with each of several associated I/O servers. In the case without intelligent servers, the client library carries the burden of orchestrating all of these steps. However, if an intelligent server carries out the operation on behalf of the client, then the client library is simplified to sending a single request across the network and waiting for a single acknowledgment upon completion.

The intelligent cooperative server approach obviously does not eliminate the complexity of the operation; it simply moves it to the servers instead of the clients. However, there are still substantial benefits to this approach. One important benefit is reduction in indirect CPU load on client applications. This is especially significant for applications or application specific libraries that are capable of overlapping computation with I/O, or compute nodes executing more than one application. Reduced client complexity also enables the client library to be more easily ported to new environments. It also becomes simpler for tools to interact with the file system at a protocol level if they wish to bypass use of the standard library. The steps involved from a client perspective are greatly simplified.

Figure 6.10: Example request exchange scenarios from client perspective

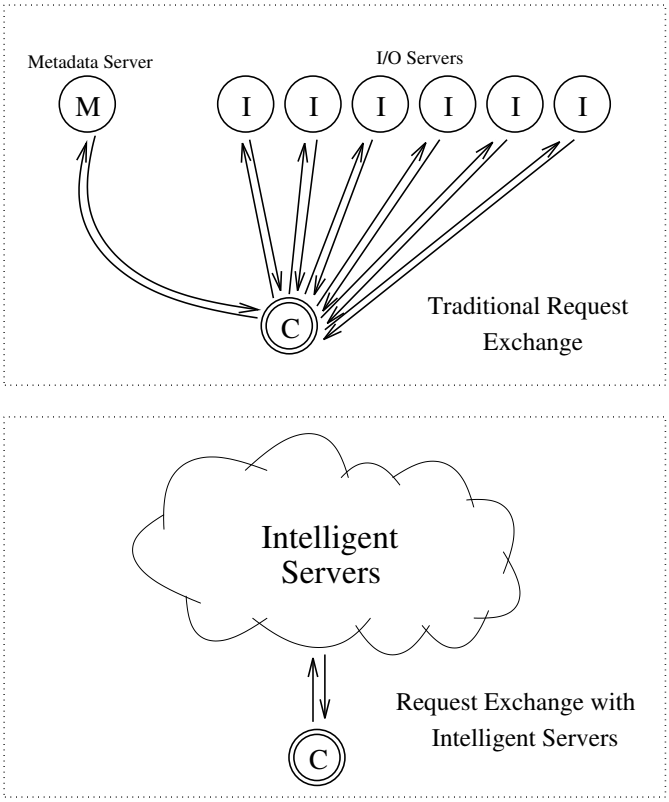
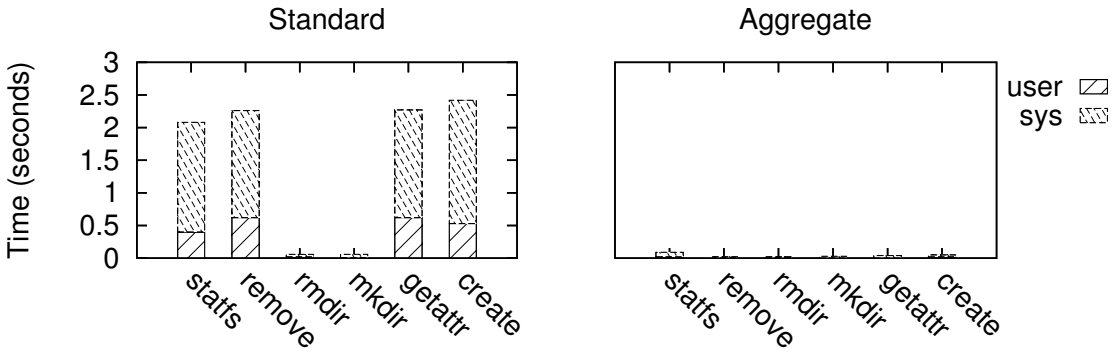


Figure 6.11: Client CPU time: TCP/Ethernet 249 servers



6.2.1 Client CPU Utilization

We can quantify client utilization by measuring the elapsed *CPU time* for various operations. CPU time is the amount of time a process expends on the system processor; it does not count time spent waiting for peripherals or time spent waiting in the CPU scheduler. It can be measured using system calls such as *getrusage()* or command line tools such as the *time* command built into tcsh. CPU time is normally divided into two categories. The first is user time, which represents CPU time spent directly by the user level application. The second is system time, which represents CPU time spent by the operating system to service the application.

Figure 6.11 shows the amount of CPU time spent during 35 iterations for each of six example operations. These tests were conducted on the Jazz Linux cluster with 249 servers and one client connected via Ethernet, both with standard PVFS2 and version of PVFS2 modified using the extensions from chapter 4. The bar graphs are stacked to show combined system and user time. In some cases the conventional approach consumes over 10 times as much CPU time as the optimized version. The difference is most pronounced on operations that require simultaneous interaction with multiple servers.

Figure 6.12: Client CPU time: GM/Myrinet 128 servers

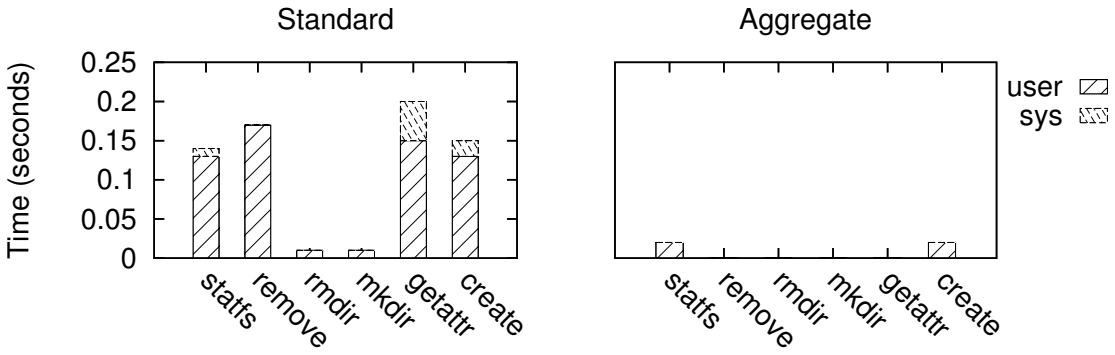


Figure 6.13: Interactive CPU time: Adenine 74 servers

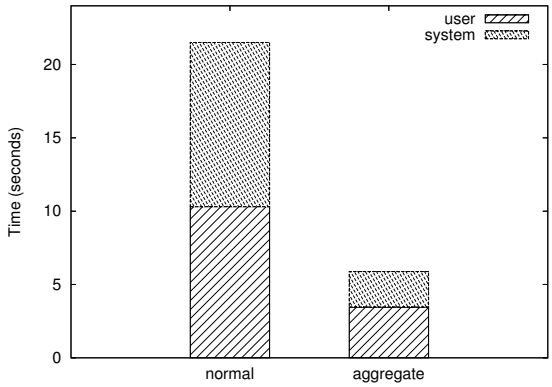


Figure 6.12 shows the same results as the previous graph except that a Myrinet network and only half the number of servers were used. Aside from the lower number of servers, the CPU time is much lower in this case largely due to the improved efficiency of the Myrinet network which offloads communication work to the network interface card. However, the aggregate optimizations still serve to reduce the CPU time by a factor of 5 in some cases.

Figure 6.13 shows the total CPU time consumed by the client during the untar, ls, and rm experiments from section 6.1.1. In this case, most of the CPU time was consumed by the pvfs2-client process (see section 3.2.2) rather than the

Table 6.4: SLOC for system interface functions

	create	remove	mkdir
Standard	818	777	447
Aggregate	240	150	188

application. We sum both together for completeness and find that CPU utilization has been reduced by nearly a factor of four through use of aggregate optimizations. These results were gathered using the Adenine Linux cluster with 74 servers.

6.2.2 Client Code Complexity

Several metrics have been developed to analyze code complexity [89]. Physical source lines of code (physical SLOC) is one example. Physical SLOC is a measure for code size that does not include comments or white space. Table 6.4 shows SLOC metrics for three example system interface functions, both with and without optimizations. These results were generated using David A. Wheeler's "SLOCCount" utility [88]. It is important to emphasize that this complexity has not been simply removed; it has been replaced by additional work on the server side. However, this configuration does allow for easier porting of client side libraries and enables easier protocol level interaction for tools that do not wish to use standard file system interfaces.

The use of Linux based file servers with custom operating systems on compute nodes is not uncommon for modern parallel computers. Notable examples of this configuration include the IBM Blue Gene/L and Earth Simulator systems, which rank 1st and 3rd in the top 500 list of supercomputers in the world as of November 2004 [27]. Systems such as these would require porting of the client side libraries but not the servers in order to use PVFS2.

6.3 Management

Ease of management can be an important factor in determining whether a file system is useful in production deployment or not. This is not only true for system administrators, but also for end users who wish to optimize applications for better I/O performance. Examples of common management tasks include:

- adding servers or storage devices
- improving utilization of existing storage
- monitoring performance
- modifying configuration settings
- tuning performance
- repairing damaged file systems

These activities not only insure correct operation, but can ultimately impact performance as well. Many parallel file systems lack tools for these tasks and instead rely on ad-hoc solutions implemented by system administrators. In addition, management tasks such as performance monitoring or event logging, even if implemented well, may become too costly at scale to be of practical use.

The use of intelligent servers and collective communication can help to address these management challenges in parallel file systems, both by improving the efficiency of management task and by enabling functionality that otherwise would not be possible. In this section we will investigate two specific management enhancements that have been implemented as proofs of concept in PVFS2. First we will examine automatic load balancing as a tool for resource management. We will then investigate the use of intelligent servers and collective communication to enhance performance monitoring.

6.3.1 Load Balancing Results

Load balancing is one approach to improving resource utilization. Load balancing may be based on several different metrics, including server activity, storage space, or network saturation. Regardless of which metric is used, the servers must have some mechanism for communicating and exchanging system wide statistics. In this section we will show how the server status composition operation outlined in section 4.3 can be leveraged to meet this requirement.

As a proof of concept, we have implemented load balancing based on the amount of storage space available at each server. The storage space available at each server can become unbalanced for any number of reasons, ranging from application access patterns and distribution configuration to the use of heterogeneous storage devices. The standard PVFS2 implementation makes no attempt to balance load other than to choose a random starting server for each set of data files. The data is then distributed round robin to each server, using a striping algorithm by default. If any one of the servers exhausts its storage capacity, then the file system as a whole will run out of space.

We chose to implement static load balancing by selecting the least loaded data servers first when deciding where to place data objects for newly created files. The server status composition information provides the necessary global statistics, and the intelligent server file creation algorithm allows servers to transparently make distribution decisions independent of the client library. The same number of servers (all servers by default) will still be used, but sorted in an optimal manner in response to load.

A more advanced algorithm would perform dynamic balancing by moving existing data at run time in response to system parameters. Intelligent servers could communicate with each other and synchronize to provide this functionality. How-

Figure 6.14: Untar: homogeneous servers

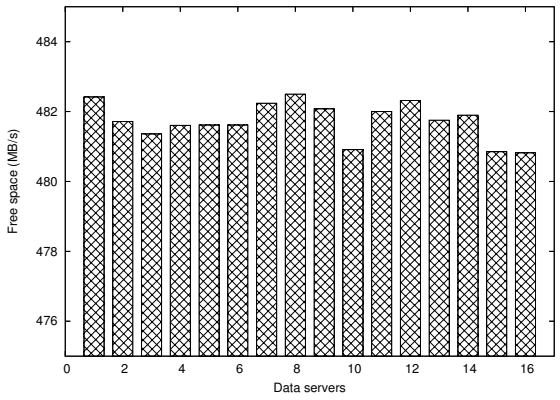
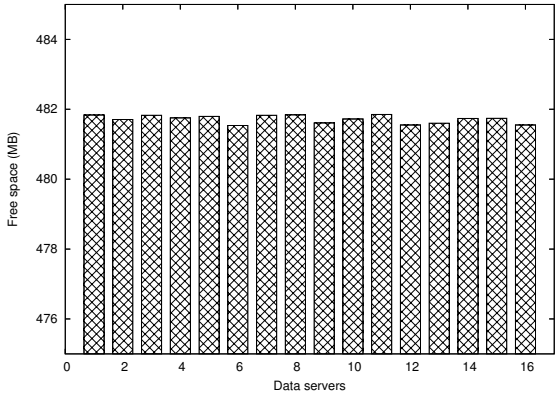


Figure 6.15: Untar with load balancing: homogeneous servers



ever, the static algorithm is much simpler and will serve as a building block for more complicated algorithms.

Figure 6.14 shows the resulting distribution of free space among 16 homogeneous servers after untarring the Linux kernel source on a standard PVFS2 file system. The Linux kernel source, as noted in section 6.1.1, contains over 16,000 files and takes up 227 MB of storage space. In this figure all servers started with 500 MB of free space. After completion of the untar command, we see that the random starting server with round robin distribution employed by PVFS2 results in a reasonable distribution of free space.

Figure 6.16: Untar: heterogeneous servers

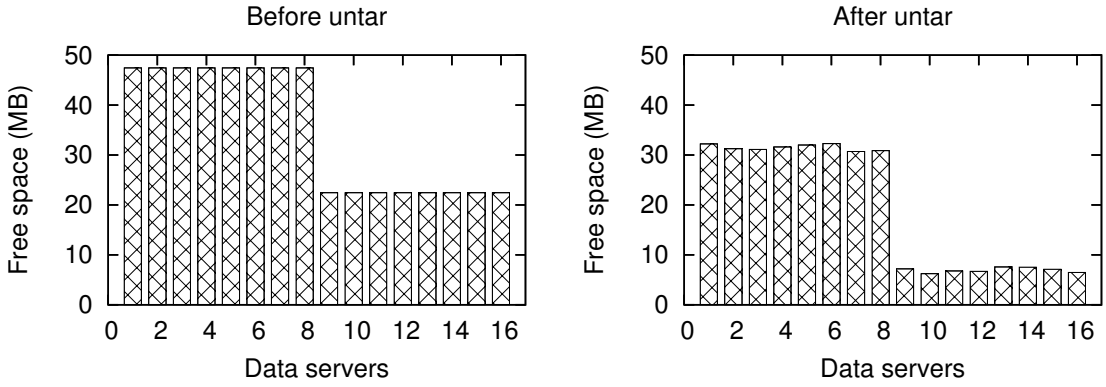


Figure 6.17: Untar with load balancing: heterogeneous servers

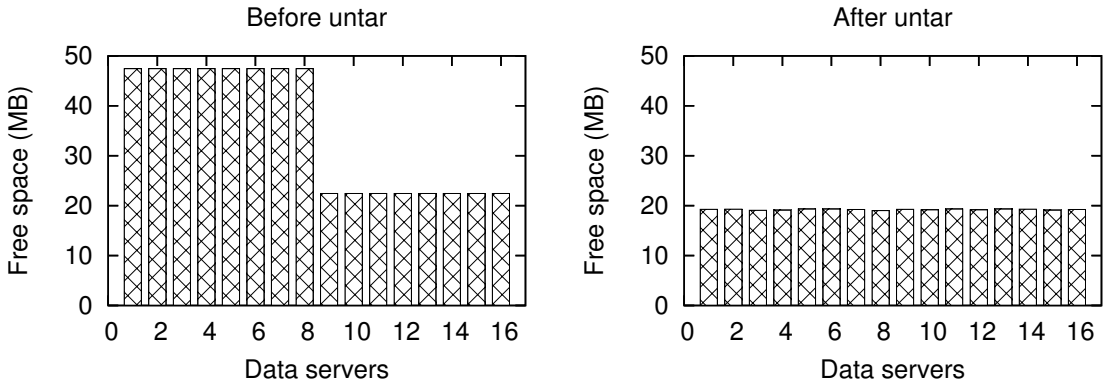


Figure 6.15 shows the results of untarring the Linux kernel on a homogeneous set of servers using PVFS2 with static load balancing. It results in slightly improved distribution of data over the standard algorithm employed in figure 6.14, although neither approach can really be said to perform poorly on this workload.

Figure 6.16 shows the results of the same test from figure 6.14, except that the servers begin with a heterogeneous configuration. Half of the servers started with 50 MB of free space each, while the other half started with 25 MB each. After untarring the Linux kernel we see that the server utilization remains unbalanced, and that half of the servers have now almost entirely exhausted their storage capacity.

Figure 6.18: Performance monitor latency: Jazz TCP/Ethernet

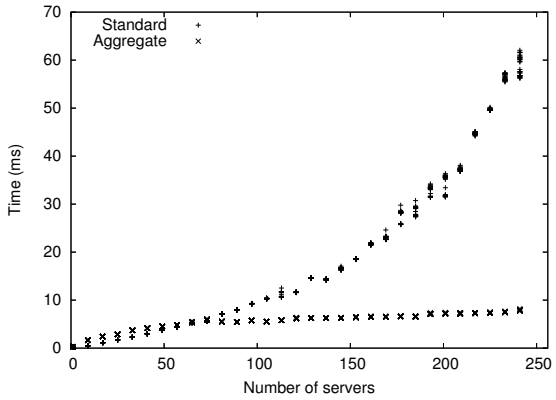


Figure 6.17 shows the results of untarring the Linux kernel on a heterogeneous system as in figure 6.16 using static load balancing. This time we find that the servers have balanced storage utilization quite well during the course of the experiment, and that overall file system capacity has been maximized. All server retain approximately 20 MB of storage capacity. The load balancing algorithm has therefore improved overall utilization in this case.

6.3.2 Performance and Event Monitoring

The performance and event monitoring facilities in PVFS2 were described briefly in section 3.4. In order to achieve better efficiency, we can employ the same intelligent server and collective communication techniques that were used for traditional file system operations in section 4.5.

Figures 6.18 and 6.19 show the results of timing performance monitoring operations on the Jazz Linux cluster as the number of servers as increased. The first figure shows Ethernet results while the second one shows Myrinet results. In both cases the intelligent server and collective communication techniques have greatly lowered the cost of the operation. In the Ethernet case, however, we do not see an improvement

Figure 6.19: Performance monitor latency: Jazz GM/Myrinet

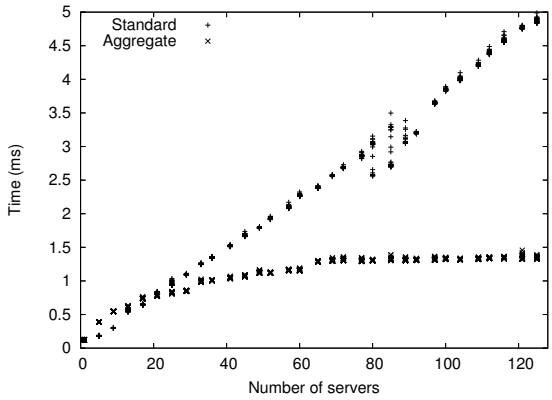


Table 6.5: Performance monitoring rate: Adenine 64 servers

	samples/minute
Standard	10,679
Aggregate	12,478

until at least 60 servers are involved in the operation. The biggest gains occur at much larger file system sizes.

Table 6.5 shows the performance monitoring sample rate that can be achieved on Adenine with 64 servers, both with and without aggregate optimizations. All servers were idle in this case. With 64 servers we see a 17% improvement in sample rate. This test is shown for benchmarking purposes only. This sample rate is impractical in production mode using either algorithm due to the load that it would induce on servers.

A more reasonable application of the performance monitoring framework would be to leverage it to instrument an MPI-IO implementation. The MPI-IO implementation could then query performance statistics after critical operations or at specified intervals in order to correlate application and server behavior. The intelligent server and collective communication optimizations insure that the cost of this instrumentation would not grow unreasonably as the size of the file system is increased.

6.4 Consistency

Consistency problems arise in parallel file systems any time that more than one process has the ability to concurrently access and modify files. For example, one process may attempt to read a file while another process simultaneously attempts to delete it. There is no way to control global timing within the cluster to inherently prevent this type of scenario. It may be possible to avoid the problem with distributed locking, but distributed locks are difficult to implement in a manner that is both scalable and fault tolerant [6] [42] [43]. These problems are exacerbated in large systems because the large number of resources to coordinate offer more opportunities for skewed file system state.

Consistency issues are most evident in name space operations, such as those that add, remove, or rename files and directories. Most name space operation actually consists of multiple small steps and are therefore never truly atomic. If they are orchestrated by the client, then there is an opportunity for inconsistent state to arise if a client crashes before completing the operation. In some cases there is no danger of inconsistent state, but there is instead a risk of stranding unreachable objects in the file system. These unreachable objects will not impact semantics. However, they can only be recovered and deleted by way of an expensive file system check (fsck) operation.

Other consistency problems arise from race conditions during concurrent operations. The most serious cases generally occur due to programmer error, such as multiple processes creating or deleting the same file simultaneously. These race conditions, once detected, will result in expensive cleanup operations orchestrated by a client library.

Intelligent servers can address consistency problems in two ways. First of all, intelligent servers take responsibility for multi-step operations away from the client. This means that client failures, especially due to application errors, will not

impact the progress of the operation. File system servers also tend to benefit from more robust hardware configurations than client processors do. Secondly, intelligent servers can utilize a request scheduler to prevent race conditions while multi-step operations take place. This can be accomplished by contacting the parent directory server first so that it can restrict directory access while name space changes take place. This type of access restriction is not possible from a client library without distributed locking. Directory servers, however, only need to lock local resources in order to protect consistency. As a result the intelligent server approach has much less complicated recovery implications.

In the following subsections we will outline some of the most challenging PVFS2 file system operations from a consistency standpoint. For each case we will describe how intelligent servers have either eliminated the problem or reduced its potential impact.

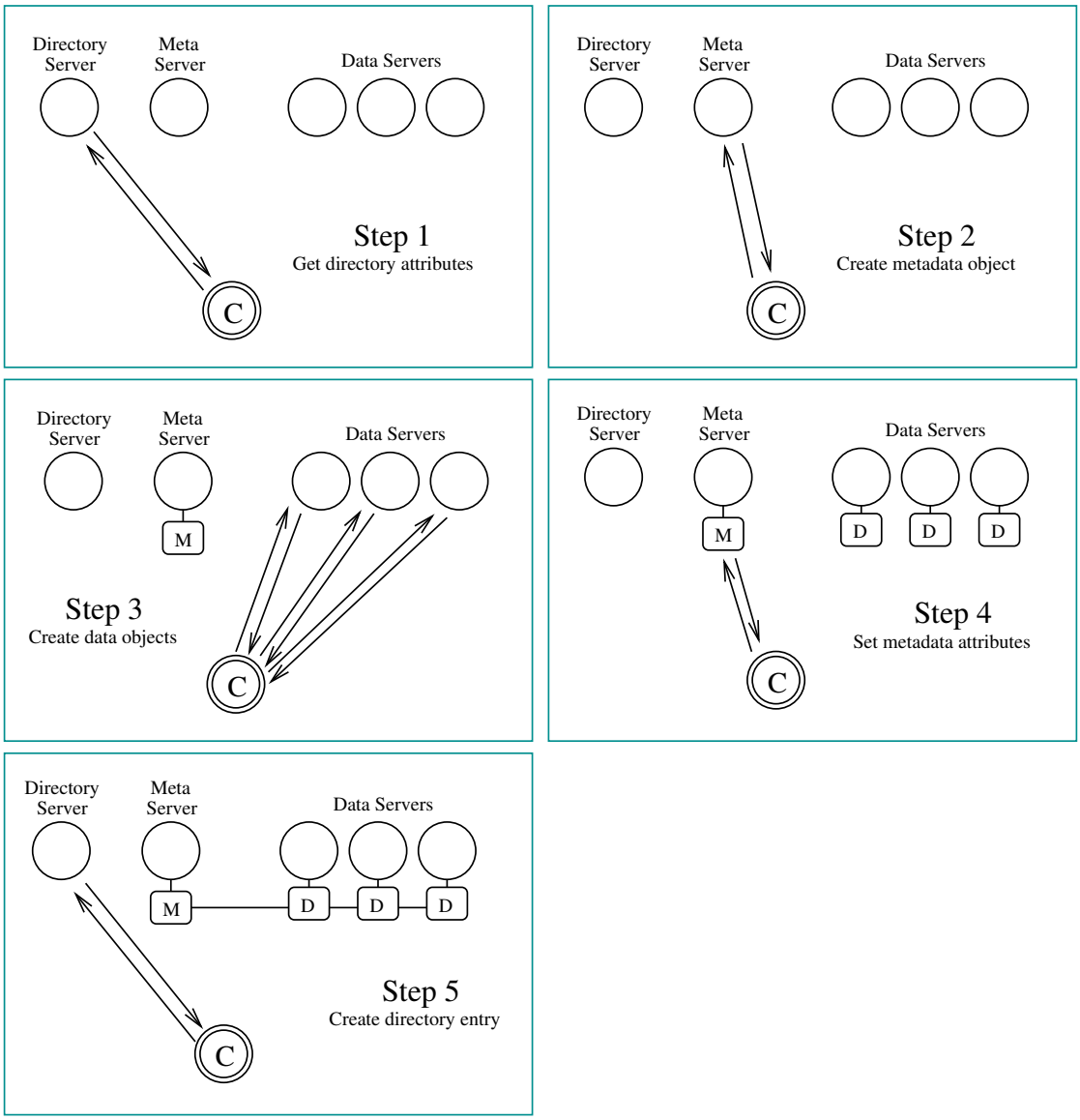
6.4.1 Create File

File creation is the first case study that we will analyze from a consistency standpoint. File creation is notable for two reasons: it modifies the file system name space and it is composed of many small steps. These two factors result in a challenging consistency situation.

Figure 6.20 shows the principle steps of the standard PVFS2 create algorithm from a network perspective. At a high level, the algorithm verifies permissions, then creates the file objects, and then links the objects into the name space by creating a directory entry. The directory entry must be created last in order to avoid the potential for a partially created object to be reachable in the file system if the client crashes before the operation is completed.

There are two potential consistency problems with this operation. First of all, the client may crash either due to hardware failure or application failure before

Figure 6.20: PVFS2 conventional create algorithm



the operation is completed. The file system is not damaged in this case because the directory entry will not have been created. However, it may strand multiple objects in the file system which cannot be recovered without a file system check operation. The second potential problem arises from race conditions. Examples include simultaneous file creation or modification of directory permissions during creation. In either case, a file creation operation will fail in its final step when it attempts to insert the directory entry. As in the previous example, this will not result in a damaged file system but it does result in a complex cleanup operation necessary to delete the metadata and data objects which have already been created. This cleanup phase must be coordinated by the client.

In contrast, figure 6.21 shows the principle steps of an aggregate PVFS2 create algorithm from a network perspective. In this case, the directory server acts as an intelligent server that orchestrates the operation on behalf of the client. By adopting this algorithm, we can eliminate both of the previously outlined consistency problems. First of all, client failure will have no impact on the operation; it will be completed by the directory server regardless of the client status. Note that it is still possible for the server itself to fail, but this is substantially less likely than client application failure. The race condition problem is solved by the fact that the directory server can restrict access to the parent directory while the operation is in progress. Competing clients therefore cannot initiate a conflicting create or permission modification until the create has completed. Subsequent conflicting creates will fail quickly with no expensive cleanup because the directory server will recognize that the file already exists before proceeding.

6.4.2 Create Directory

The steps for creating a new directory in PVFS2 are mostly the same as the file creation steps, except that there are no file data objects to manipulate. It therefore is

Figure 6.21: PVFS2 aggregate create algorithm

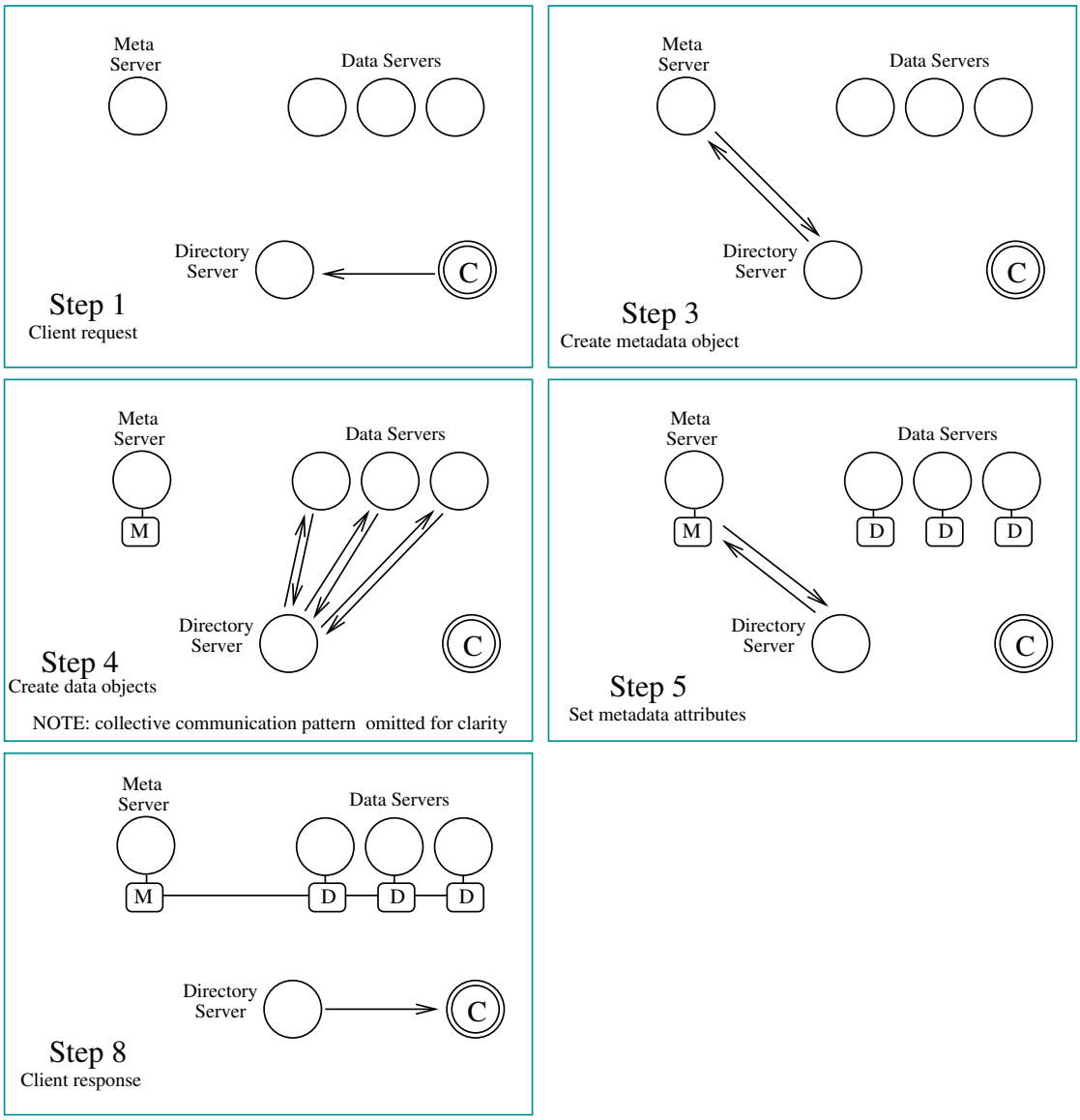
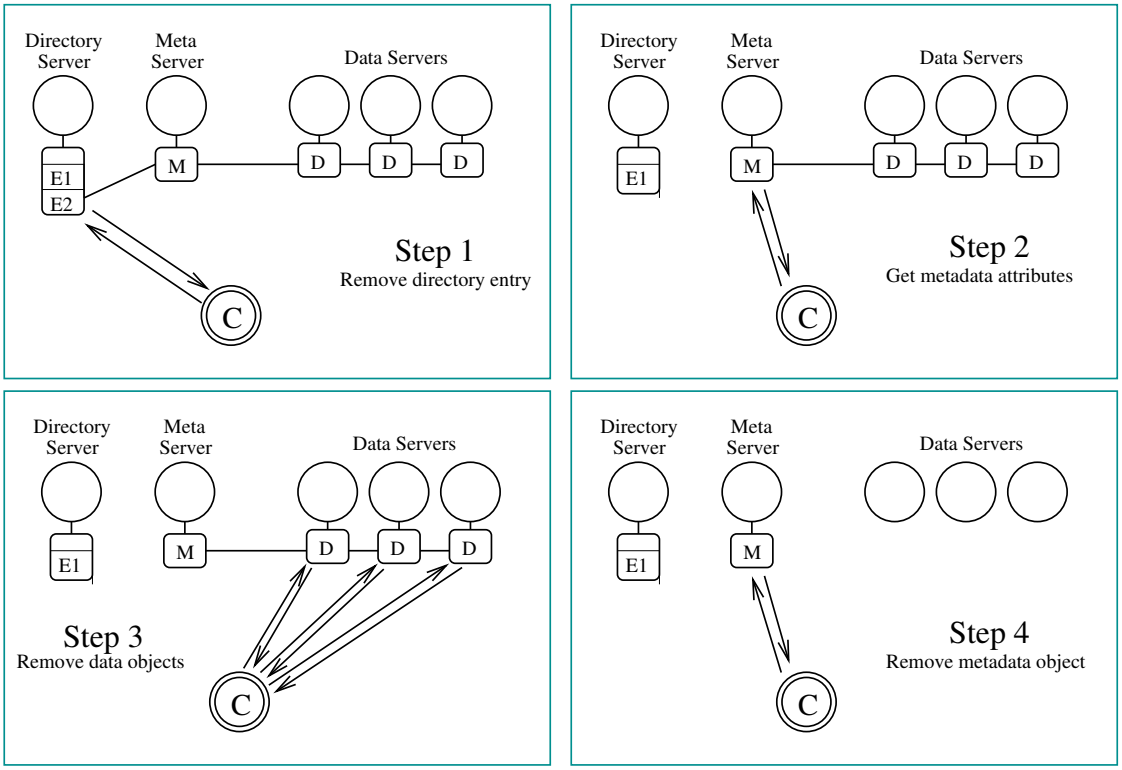


Figure 6.22: PVFS2 conventional remove algorithm



subject to the same consistency challenges and can be addressed by way of the same intelligent server algorithm.

6.4.3 Remove File

The PVFS2 remove case is similar to create in terms of the number of steps that must be performed, though the consistency challenges are different. Figure 6.22 shows the principle steps of the standard remove operation from a network perspective. Unlike the create algorithm, remove begins by modifying the parent directory. The directory entry is removed first to insure that the file becomes unreachable even if the client crashes midway through the operation. However, this still allows the possibility of stranded file system objects. This is a more serious problem in the remove case than in the create case, because the file data objects can be arbitrarily large at the time a

file removal operation begins. Client failure can therefore result in a large amount of wasted file system space until a file system check operation is performed.

The second consistency challenge arises if one of the late removal steps fails. One example would be if the server refuses to remove the metadata object due to a lack of permission. At this point it is clear that the remove should fail with a permission error from the application's point of view. However, the directory entry has already been removed, and the client must now attempt to replace it. Even if successful, a window of time has passed in which the file was not reachable from other processes.

In contrast, figure 6.23 shows the intelligent server file removal algorithm. As in the create case, the directory server acts as an intelligent server to perform the component steps on behalf of the client. This eliminates risk of stranded data objects as a result of client failure. The directory server can also restrict access and perform entry removal as the final step. This eliminates any window of time in which the directory entry may be incorrectly removed and insures that it is never possible to access an incomplete file in the name space.

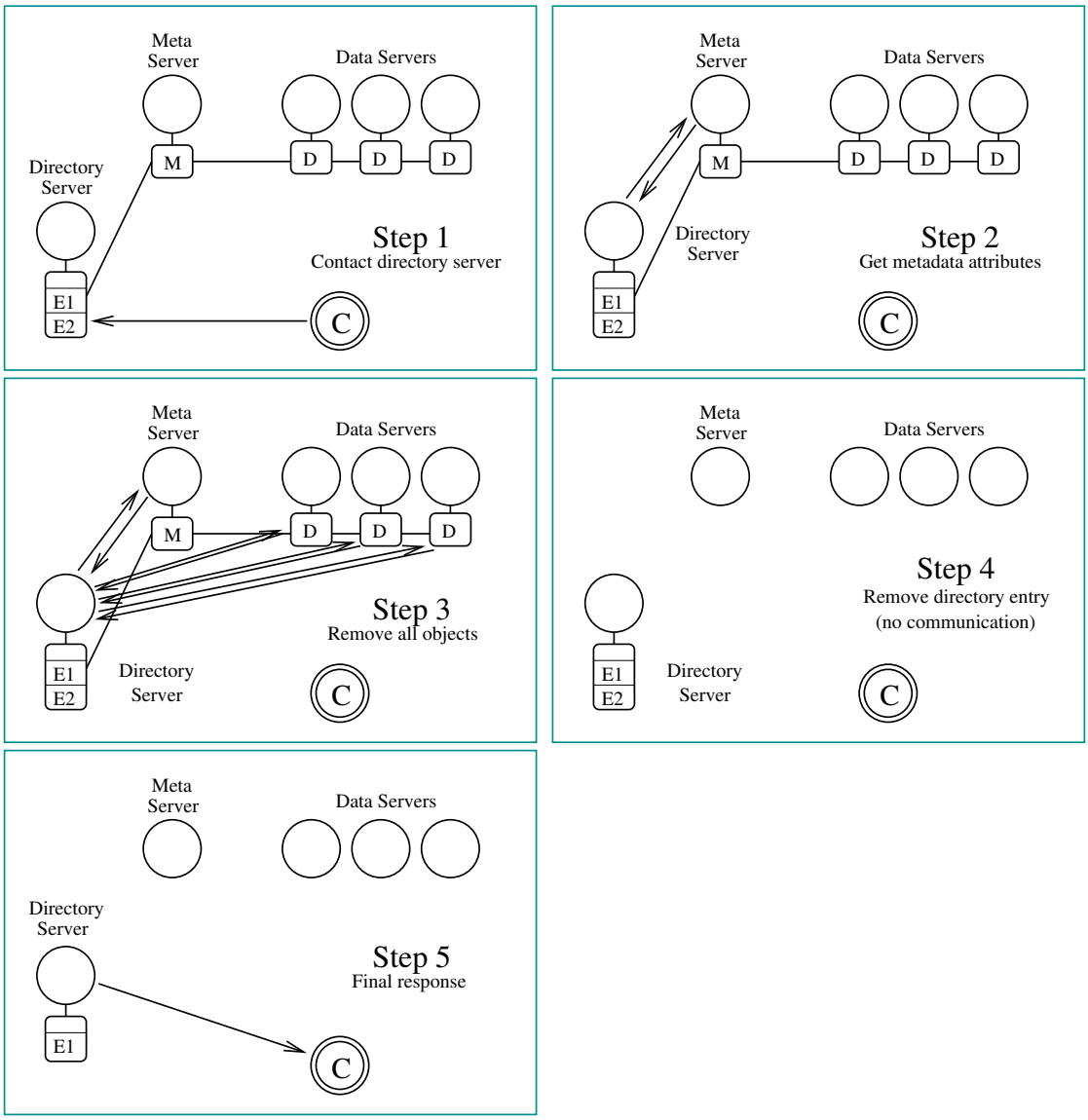
6.4.4 Remove Directory

The process of removing directories is very similar to that of removing files, except that no datafiles are involved. Rmdir consistency can be improved via the same intelligent server techniques.

6.4.5 Other Cases

The four examples of remove, create, rmdir, and mkdir have all been implemented using intelligent servers and collective communication, as described in chapter 4.5. However, there are several other metadata operations that either modify the file system name space or rely on file system name space consistency for correct operation.

Figure 6.23: PVFS2 aggregate remove algorithm



Examples include the lookup, symlink, readdir, and rename operations. Rename in particular is one of the most challenging operations due to the fact that it must make two name space changes (addition of one filename and removal of another) while simultaneously preserving file contents. Each of these operations could be improved through variations of the intelligent server algorithms already discussed.

6.5 Fault Tolerance

Fault tolerance is essential to productivity in large computing environments, especially for storage subsystems. File systems store data which may impact many applications and many users simultaneously. Therefore, their fault tolerance expectations are much different than that of other system software such as message passage libraries. To make matters worse, the mechanical properties of hard disks makes them one of the most likely components to fail in any computer system. This problem is exacerbated in large file systems in which the number of disks leads to a decrease in the average mean time between failure for the system as a whole.

As new functionality is added to a parallel file system, we must take care to insure that all subsystems allow fault tolerant behavior. We must also take advantage of any opportunity to leverage software technology to address fault tolerance problems. The techniques of intelligent servers and collective communication as outlined in this dissertation have not yet been used to directly address fault tolerance issues. However, in the following subsections we will discuss cases in which these techniques offer advantages over competing technologies in terms of fault tolerance. We will also identify ways in which intelligent servers and collective communication could serve as building blocks for specific fault tolerant features.

6.5.1 Shared Lock Avoidance

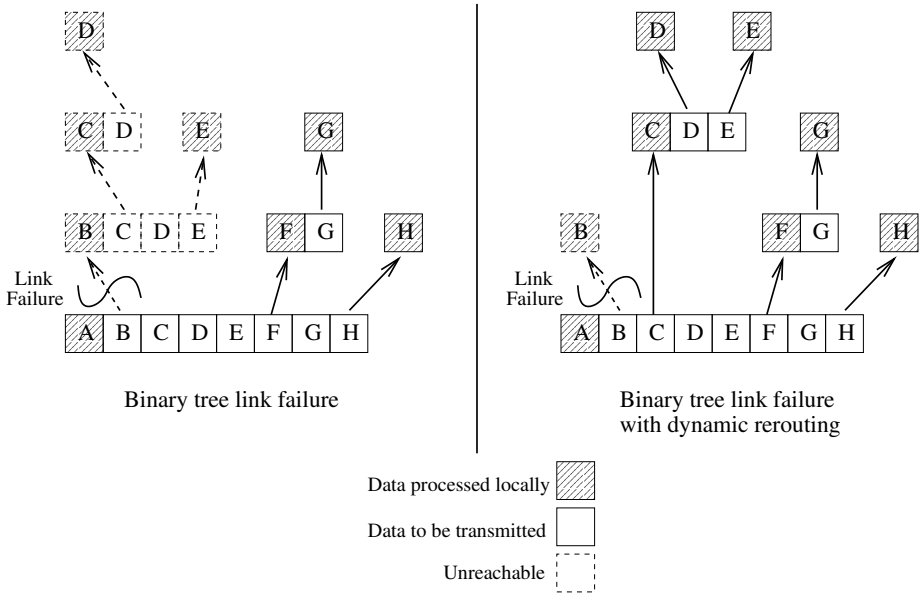
Section 6.4 outlined how intelligent servers can be used to improve consistency in a parallel file system environment. One important implication is that intelligent servers can help to provide well defined and consistent semantics without using distributed locks. This is in contrast to several other parallel file systems [10] [71] [67]. The complexity of distributed locking systems is well known [6] [42] [43], and can become one of the most elaborate components of a parallel file system once adopted. Distributed locking relies on sharing of state information between clients and servers and impedes the process of recovering from a file system fault. In particular, if a client fails while holding a lock, then there must be a timeout mechanism and/or failure verification before the lock can be reacquired by another process. If these recovery steps are not performed in an efficient manner, then individual file system faults may propagate into a system wide problem. Error propagation of this magnitude is not acceptable when faced with the fault tolerance expectations of production file systems.

Section 6.4 described eight common file system operations which can be implemented in a consistent manner by way of intelligent server algorithms without resorting to the use of distributed locks.

6.5.2 Fault Tolerant Collective Communication

In section 3.3.1, we outlined how the point to point network transfer layer for PVFS2 has been designed to allow for fault tolerant operation. The collective communication primitives built on top of it must follow through with this same philosophy. Implementation of fault tolerant collective communication is beyond the scope of this dissertation. In this section we will, however, describe how the collective communication infrastructure could be augmented to provide this behavior. We will assume, as in the point to point case, that the decision on when to retry or resume failed operations rests at a higher level than that of the communication infrastructure.

Figure 6.24: Binary tree with dynamic rerouting

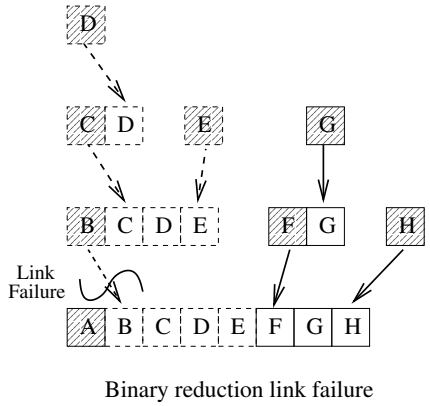


The one to many binary tree is the most common collective communication algorithm employed in this work. The key to achieving fault tolerance in a binary tree is to insure that failure at any one node of the tree does not result in failed communication to any other nodes in the tree. In other words, we wish to limit the propagation of the failure. In order to achieve this we must rely on the dynamic nature of the binary tree as described in section 4.2.

At each stage in the one to many communication, the current server disassembles the incoming message, breaks apart the data it contains, and composes new messages to subsequent servers. Since this is done at a high level, the servers have an opportunity to dynamically redirect where messages are routed. This is normally done to reflect locality or load, but it can also be leveraged to route around network faults.

For example, suppose that the next server that a message should be routed to is unreachable. This may be determined either because of a point to point messaging failure or due to information from a heartbeat monitor. If we simply accepted the

Figure 6.25: Binary tree reduction failure



failure at that point, then communication with an entire subset of servers might have been aborted. However, we can instead reconstruct the messages to give the remaining work to a different server that is reachable. Figure 6.24 shows an example of this behavior.

When reducing the response to this type of collective communication using a many to one collective pattern, we would like to employ a similar technique. Unfortunately, the steps in this process do not inherently possess information about other servers involved in communication as in the one to many case. Each server only knows two things: which hosts to receive incoming data from, and which host to forward the combined data to. If the latter host fails then communication cannot continue. Figure 6.25 shows an example of this scenario. In this case, host B cannot forward its results to host A. In order to overcome this problem, we would need to make sure that requests carry additional alternative routing information. In the simplest form, each response would have a choice of two hosts to relay results to. We could then insure that at each step of the operation there are multiple options for the return path of the data.

The fault handling algorithms described thus far do not account for all failure conditions. In particular we have not addressed the Byzantine failure cases described

in section 2.6. However, these algorithms would encompass a significant range of practical failure conditions and serve as building blocks for more sophisticated techniques.

All to all collective communications are significantly more complex from a fault tolerance perspective. Unlike the binary tree algorithms, the recursive doubling algorithm employed for all to all communications relies on all participants to possess a consistent ordering of hosts. If one fails, then it will impact the algorithm for all parties, not just immediate peers. Fault tolerance in this algorithm would therefore require a complete fault detection (or heartbeat) system as described in section 2.6 to insure that all participants are informed of the fault consistently and in a timely manner.

6.5.3 Quorum and Heartbeat Applications

Though we have not focused on implementation ramifications yet, quorum and heartbeat systems are clear applications of intelligent servers and collective communication infrastructure. We described the usefulness of quorums and heartbeats for data redundancy in the related work in section 2.6. Even as early as 1979, Robert Thomas identified that various collective patterns such as daisy chaining or broadcasting could be employed for use in quorum voting with various trade-offs in latency and safety [83]. Implementation of such a system in PVFS2 would require intelligent servers: servers with awareness of each other who can make autonomous communication decisions.

Heartbeat systems would likewise benefit from intelligent servers and collective communication. In particular, scalable heartbeat systems [33] would rely on servers using more efficient communication than simply expecting each server to repeatedly send a heartbeat message to every other server. In addition, efficiency could be improved by using scalable collectives to notify peer when an individual host identifies

a failure condition, rather than waiting for a complete fault detection algorithm to converge on its own.

CHAPTER 7
CONCLUSIONS

Parallel file systems have proven to be useful tools for providing high performance data throughput to parallel applications on many architectures. However, trends in parallel computing have lead to an increase in the deployment of large scale systems with thousands of processors. In order to achieve peak performance, these systems demand I/O scalability that exceeds the current state of file system technology. The solution presented here is to augment parallel file systems by integrating intelligent server and collective communication features. Intelligent servers are a wholly original concept, while collective communication relies on novel application of research from distributed shared memory and message passing research. These features enhance the core functionality of the file system and limit network overhead to address five key obstacles to scalability: efficiency, complexity, management, consistency, and fault tolerance.

We began by presenting the Parallel Virtual File System, version 2. PVFS2 is a modern parallel file system designed from the ground up to embody the principles of scalability and flexibility and to serve as a platform for high performance I/O research. PVFS2 is also a production level tool actively deployed at sites around the world. We outlined several components of the PVFS2 design which have enabled the implementation and evaluation of the research described in this text.

We next presented extensions to PVFS2 which incorporated intelligent server and collective communication capability into the file system. This work included an inter-server messaging framework as well as novel application of well known structured communication concepts to efficiently leverage it. Servers were enhanced to autonomously gather system state and interpret system characteristics in order to

make decisions regarding distribution of work and preservation of file system consistency. Several example file system functions were implemented using these new extensions.

Analytical models were constructed to parameterize and predict the performance in PVFS2 of both conventional algorithms and aggregate algorithms that utilize intelligent servers and collective communication. These models were verified through comparison to empirical results using multiple interconnection networks and hundreds of processors. Some of the models were accurate enough to account for as much as 99% of the system behavior not attributed to measurement variance, while others pointed out avenues of future work. These models were used to predict the behavior of file systems comprised of thousands of servers and suggest that operation efficiency could be improved by a factor of over 400 in some cases.

The experimental implementation was evaluated in terms of each of the five key obstacles to file system scalability. Efficiency improvements were demonstrated by doubling the throughput of several practical metadata benchmarks on a 75 node cluster. Wide area network efficiency was improved as well. A reduction in client code complexity and a ten-fold decrease in client CPU overhead were presented to evaluate the success of complexity reduction in the PVFS2 client library. Load balancing based on disk usage and enhanced performance monitoring demonstrated improvements in file system management. Consistency weaknesses in four specific case studies were eliminated by use of intelligent servers. Finally, intelligent servers were shown to be a valid means to improve fault tolerant consistency when compared to conventional alternatives. The preliminary implementation was also suggested as a building block for fault tolerant collective communication and comprehensive fault detection systems.

Aspects of all five obstacles to scalability have been successfully addressed in this study. We therefore conclude that the concepts of intelligent servers and collective communication can be applied within the framework of parallel I/O to

enable file systems to scale effectively to the next generation of parallel systems with thousands of processors.

7.1 Contributions

This work has yielded five primary contributions to the field of parallel I/O:

- Several building blocks for a modern production level parallel file system were implemented, including network abstractions, thread management, and request scheduling.
- The current state of file system technology was evaluated to determine the five most pressing obstacles to file system scalability.
- An analytical framework was developed for comparing file system algorithms and predicting performance.
- An implementation of intelligent server and collective communication extensions to the PVFS2 file system was completed.
- Several file system operations were optimized to take advantage of the intelligent server and collective communication extensions. The resulting implementation was evaluated with regard to each of the five obstacles to scalability and shown to enable file system scalability for the next generation of parallel systems with thousands of processors.

7.2 Future Work

Extending PVFS2 using intelligent servers and collective communication has improved many aspects of file system scalability. However, this technology could also serve as a platform for new research directions in parallel I/O, several of which would not be possible otherwise. In this section we will outline how intelligent servers and collective communication can enable advancement the areas of autonomous storage,

performance optimization, redundancy, performance modeling, and domain specific optimizations.

The concept of autonomous storage has gained popularity in recent years as high performance storage devices have become more diverse and difficult to administer. The goal of autonomous storage is to create storage subsystems that can protect data integrity, recover from faults, and optimize performance with minimal or no input from a system administrator. We have provided a clear first step in this direction with servers that are capable of taking an active role in algorithm decisions and communication with each other. For example, servers could balance load dynamically by migrating data objects based on application locality or resource utilization. Servers could likewise make autonomous tuning decisions by altering parameters such as the file distribution, cache strategy, or operation algorithm based on file access history and the number of servers involved. Finally, intelligent servers could perform file system integrity checks and automatically repair or adapt to file system damage.

There are also opportunities for performance enhancements beyond the direct gains already shown in large scale metadata operations. First of all, collective communication could be applied to the I/O path. In particular, optimizations such as two phase I/O that are normally implemented at the application library layer could be performed with assistance from servers that understand the underlying file distribution and inherently cooperate with other servers and clients. Distributed caching frameworks could also take advantage of file system level collectives as an efficient signaling or invalidation mechanism. Finally, intelligent servers could incorporate topology awareness to aggregate operations or route tasks in ways that take advantage of the fastest network links. The recently added capability of BMI to support multi-homed configurations means that topology aware collectives do not even necessarily need to use the same type of network. For example, a client could contact

a server via TCP/IP and the server could in turn contact its peers by way of a fast storage network. This is particularly relevant to wide area and grid deployment.

Redundancy is an active area of PVFS2 development that will benefit from the work of this dissertation. At a basic level, server intercommunication provides a way to transfer replicated data and negotiate consistency. More advanced applications of this technology could lead to fault detectors and quorum voting systems for recognizing server failures and recovering from inconsistent state.

Chapter 5 presented a foundation for analytical modeling of file system operations. However, there are several opportunities for improvement. One notable area of research pointed out in this study is a more formal investigation of the impact of overlap between network and disk activity. Modern file servers such as PVFS2 allow for pervasive concurrency which has not yet been quantified sufficiently. Further refinements and wider applicability of the analytical models could also be achieved by applying them in a simulation environment that can emulate sophisticated access patterns or full application behavior.

Lastly, intelligent servers and collective communication open up the possibility for advanced domain specific optimizations. For example, programmable server operations could offload computation from clients to perform tasks that avoid network overhead and operate directly on local data. If these programmable operations incorporated distribution awareness, then servers could communicate with each other to manipulate data in parallel. Internal server functionality could likewise benefit from similar operations such as distributed checksum, parity, or file version calculations.

BIBLIOGRAPHY

[1] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, and D. M. Dias. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[2] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Workshop on Distributed Algorithms*, pages 126–140, 1997.

[3] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, 1999.

[4] Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, and Michael K. Reiter. Fault detection for byzantine quorum systems. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):996–1007, 2001.

[5] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Interprocessor collective communication library (InterCom). In *Proceedings of the Scalable High Performance Computing Conference*, 1994.

[6] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[7] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet a gigabit per second local area network. *IEEE Micro*, 15, February 1995.

[8] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, Portland, OR, 1993. IEEE Computer Society Press.

[9] P. J. Braam. The Coda distributed file system. *Linux Journal*, (50), June 1998.

[10] Peter J. Braam. The lustre storage architecture. Cluster File Systems Inc. Architecture, design, and manual for Lustre, November 2002. http://www.lustre.org/docs/lustre.pdf.

[11] P. Carns, W. Ligon III, S. McMillan, and R. Ross. An evaluation of message passing implementations on beowulf workstations. In *Proceedings of the 1999 IEEE Aerospace Conference*, 1999.

[12] P. Carns, W. Ligon, R. Ross, and P. Wyckoff. BMI: A network abstraction layer for parallel I/O. In ***Under review*** *for The Workshop on Communication Architecture for Clusters in Conjunction with the International Parallel and Distributed Processing Symposium 2005*, December 2004.

[13] Philip H. Carns. Design and analysis of a network transfer layer for parallel file systems. Master's thesis, Clemson University, Clemson, SC, December 2001.

[14] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[15] Mark Carson and Darrin Santay. NIST Net: a Linux-based network emulation tool. *SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[16] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, 1995.

[17] Abhishek Chandra and David Mosberger. Scalability of Linux event-dispatch mechanisms. In *Proceedings of the USENIX Annual Technical Conference*, pages 231–244, 2001.

[18] G. Chiola and G. Ciaccio. GAMMA: a low-cost network of workstations based on active messages. In *Proceedings of PDP'97 (5th EUROMICRO workshop on Parallel and Distributed Processing)*, London, UK, January 1997.

[19] Russell Coker. Bonnie++ benchmark suite. `http://www.coker.com.au/bonnie++/`.

[20] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 32, pages 477–487. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[21] Standard Performance Evaluation Corporation. SPEC CPU benchmark results. http://www.spec.org.

[22] D.A.Patterson and J.L.Hennessy. *Computer architecture: A quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, third edition, 2002.

[23] Rogerio Luis de Carvalho Costa and Sergio Lifschitz. Database allocation strategies for parallel BLAST evaluation on clusters. *Distributed Parallel Databases*, 13(1):99–127, 2003.

[24] Nathan A. DeBardeleben, Walter B. Ligon III, and Dan C. Stanzione Jr. The Component-based Environment for Remote Sensing. In *Proceedings of the 2002 IEEE Aerospace Conference*, pages 6–2661–6–2670, March 2002.

[25] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, Newport Beach, CA, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[26] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[27] Jack Dongarra, Hans Meuer, and Erich Strohmaier. Top500 supercomputer sites. http://www.top500.org.

[28] P.A. Farrell and Hong Ong. Communication performance over a gigabit ethernet network. In *Performance, Computing, and Communications Conference, 2000. IPCCC '00. Conference Proceeding of the IEEE*, pages 181–189, 2000.

[29] Dennis Fowler. The next internet. *netWorker*, 3(3):20–29, 1999.

[30] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 150–162, 1979.

[31] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press, 1981.

[32] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message-passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[33] Indranil Gupta, Tushar D. Chandra, and German S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 170–179. ACM Press, 2001.

[34] M. Halem, F. Shaffer, N. Palm, E. Salmon, S. Raghavan, and L. Kempster. Can we avoid a data survivability crisis? *Science Information Systems Newsletter*, (51), 1999.

[35] Seungjae Han and Kang G. Shin. Experimental evaluation of behavior-based failure-detection schemes in real-time communication networks. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):613–626, 1999.

[36] HDF5 scientific file format. http://hdf.ncsa.uiuc.edu/HDF5/.

[37] Stephen Hemminger. Netem network emulator. `http://developer.osdl.org/shemminger/netem/`.

[38] Ibm deskstar GXP datasheet. http://www.hitachigst.com/hdd/desk/ds120gxp.htm. Hitachi Corporation.

[39] IBM Corporation. IBM Blue Gene project overview. http://www.research.ibm.com/bluegene/.

[40] IEEE. IEEE standard portable operating system interface for computer environments (POSIX) 1003.1, 1990.

[41] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–part 1: System application program interface (API) [C language], 1996 edition.

[42] Paris C. Kanellakis and Christos H. Papadimitriou. Is distributed locking harder? In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 98–107. ACM Press, 1982.

[43] Paris C. Kanellakis and Christos H. Papadimitriou. The complexity of distributed concurrency control. *SIAM Journal of Computing*, 14(1):52–74, 1985.

[44] J. Katcher. Postmark: A new file system benchmark, 1997.

[45] Eiji Kawai, Youki Kadobayashi, and Suguru Yamaguchi. Improving scalability of processor utilization on heavily-loaded servers with real-time scheduling. In *Proceedings of International Conference on Parallel and Distributed Computing and Networks (PDCN2004), IASTED*, 2004.

[46] Sandeep S. Kulkarni and Ali Ebnenasir. The complexity of adding failsafe fault-tolerance. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July 2002.

[47] Sandia National Laboratory. ASCI Red Storm fact sheet. http://www.sandia.gov/ASCI/phys_infrastruc.html.

[48] R. Latham, R. Ross, and R. Thakur. The impact of file systems on MPI-IO scalability. Technical Report Preprint ANL/MCS-P1182-0604, Mathematics and Computer Science Division, Argonne National Laboratory, June 2004.

[49] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, 1991.

[50] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, 2003.

[51] D. Libenzi. Linux epoll patch. `http://www.xmailserver.org/linux-patches/nio-improve.html`.

[52] David Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2000.

[53] Mark Lord. hdparm - utility for manipulating hard drive parameters. http://www.gnu.org/directory/hdparm.html.

[54] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 569–578. ACM Press, 1997.

[55] Diamondmax Plus datasheet. http://www.maxtor.com/en/documentation/data_sheets/dm_plus_60_ultra_ata_datasheet.pdf. Maxtor Corporation.

[56] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[57] Prasenjit Mitra, David Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts. Fast collective communication libraries, please. In *Proceedings of the Intel Supercomputing Users' Group Meeting*, 1995.

[58] Myricom, Inc. The GM message passing system. `http://www.myri.com`.

[59] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[60] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[61] R. Lyman Ott and Michael Longnecker. *An Introduction to Statistical Methods and Data Analysis (5th Edition)*. Duxbury, Pacific Grove, CA, 2001.

[62] Parallel Virtual File System 2. `http://www.pvfs.org/pvfs2`.

[63] Parallel Virtual File System 2 User's Guide. `http://www.pvfs.org/pvfs2/pvfs2-guide.html`.

[64] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 1, pages 3–14. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[65] David A. Patterson and John L. Hennessy. *Computer Architecture; A Quantitative Approach (2nd Edition)*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

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

[67] Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steven R. Soltis, David C. Teigland, and Matthew T. O'Keefe. A 64-bit, shared disk file system for Linux. In *Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems*, pages 22–41, San Diego, CA, March 1999. IEEE Computer Society Press.

[68] Rolf Rabenseifner. New optimized mpi reduce algorithm. `http://www.hlrs.de/organization/par/services/models/mpi/myreduce.html`.

[69] David F. Robinson, Philip K. McKinley, and Betty H. C. Cheng. Optimal multicast communication in wormhole-routed torus networks. *IEEE Transactions on Parallel and Distributed Systems*, 6(10):1029–1042, 1995.

[70] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[71] Frank B. Schmuck and Roger L. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies*, pages 231–244. USENIX Association, 2002.

[72] Elizabeth A. M. Shriver, Bruce Hillyer, and Abraham Silberschatz. Performance analysis of storage systems. In *Performance Evaluation*, pages 33–50, 2000.

[73] Elizabeth A. M. Shriver, Arif Merchant, and John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. In *Measurement and Modeling of Computer Systems*, pages 182–191, 1998.

[74] Mohak Shroff and Robert A. van de Geijn. Collmark: MPI collective communication benchmark. Technical report, 2000.

[75] A. Sivasubramaniam, M. Vilayannur, P. H. Carns, R. B. Ross, and R. Thakur. On the performance of the POSIX I/O interface to PVFS. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network based Processing*, February 2004.

[76] Q. Snell, A. Mikler, and J. Gustafson. Netpipe: A network protocol independent performance evaluator. In *Proceedings of IASTED International Conference on Intelligent Management and Systems*, 1996.

[77] Thomas Sterling, Donald J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 1995 International Conference on Parallel Processing*, 1995.

[78] Rajeev Thakur, Alok Choudhary, Rajesh Bordawekar, Sachin More, and Sivaramakrishna Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[79] Rajeev Thakur and William Gropp. Improving the performance of collective operations in MPICH. In *Proceedings of the 10th European PVM/MPI Users' Group Meeting*, September 2003.

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

[81] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.

[82] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, January 2002.

[83] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[84] Stephen Tweedie. Ext3, journaling file system. In *Ottawa Linux Symposium*, 2000.

[85] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 3. IEEE Computer Society, 2000.

[86] VI architecture specification revision 1.0. `http://www.viarch.org`, December 1997.

[87] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[88] David A. Wheeler. Sloccount. `http://www.dwheeler.com/sloccount`.

[89] Horst Zuse. *Software complexity: measures and methods*. Walter de Gruyter & Co., 1991.