Control and Inspection of Distributed Process Groups at Extreme Scale via Group File Semantics

By

Michael Joseph Brim

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

2012

Date of final oral examination: 5/11/12

The dissertation is approved by the following members of the Final Oral Committee:

Barton P. Miller, Professor, Computer Sciences

Remzi H. Arpaci-Dusseau, Professor, Computer Sciences

Michael M. Swift, Assistant Professor, Computer Sciences

Benjamin R. Liblit, Associate Professor, Computer Sciences

Rafael Lazimy, Associate Professor, Operations and Information Management

To Lynn and Sage, my precious flowers.

Acknowledgments

A great number of people have inspired and encouraged me along this path. It is my pleasure to thank them for all they have done.

My advisor, Bart Miller, has guided my development throughout my graduate career. I owe to him my abilities to approach problems critically and soundly, and to relate ideas to others in both speech and prose. I am still in awe of his ability to know what I am thinking before I have the chance to think it. I am grateful for his trust in me to direct my research and mentor my peers, and for his support and advice in handling the stumbling blocks along the way.

I thank Dave Hudak, my undergraduate advisor at Ohio Northern University, for instilling the belief that nothing less than a doctorate would suffice given my talents. I also thank him for introducing me to Guinness, which never fails to reduce my stress level.

Stephen Scott, my supervisor during my first stint at ORNL, helped me gain confidence by sending me in his stead to my first conference and giving me the reigns to the OSCAR project. Stephen encouraged me to aim high in my graduate school applications, and I will never forget his generous recommendation letter.

My years as a member of the Paradyn group have been filled with great co-workers, far too many mention and all of whom I consider friends. I am particularly grateful to Phil Roth and Dorian Arnold for cooking up this wonderful piece of steak that is the basis for my research, and for letting me take over as they moved on to bigger and better things. Vic Zandy has always been an inspiration for his creativity in software and speaking, and was kind enough to humor me as I adapted his ideas for my own purposes. I wish him well on his difficult journey. Special thanks goes to all of my “MRNet minions”, who have helped me to become a better mentor and sounding board.

I thank my committee, Remzi Arpaci-Dusseau, Mike Swift, Ben Liblit, and Rafi Lazimy, for thoughtful questions and comments on my work. They all deserve a “good reader” badge for making their way through my lengthy tome.

I thank John DelSignore and Steve Lawrence for bouying me as I waded through the deep waters of TotalView.

Without the people and resources of the ORNL Leadership Computing Facility and the LLNL Open Computing Facility, my research would not have been possible. The U.S. Department of Energy (DOE) *Operating and Runtime Systems for Extreme Scale Scientific Computation (FASTOS)*[1] and *Software Development Tools for Improved Ease-of-Use of Petascale Systems*[2] programs that funded my work and gave me the opportunity to share my ideas with my peers in HPC, and the PEAC project within the DOE INCITE program[3] through which we have received access to leadership class machines have both been crucial to performing my work. I would like to thank Pat Worley for serving as PI of PEAC, and for encouraging me to use as many CPU hours as I needed (a mere 5.5 million hours in 2011 alone).

Last, but certainly not least, I thank my family and my wife’s family for continued love and support throughout this unexpectedly long process. Although it was hard for them to understand the reasons for the lengthy span, I never doubted they would be waiting to congratulate me at the end. I only hope I have made them proud. Lynn, your capacity for patience and understanding can never be doubted again. Sage, you are my sunshine and my joy. I love you both and will cherish you forever.

1. This work is supported in part by Department of Energy grant 08ER25842.
2. This work is supported in part by Department of Energy grants DE-SC0003922 and DE-SC0002153.
3. An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program. This research used resources of the Oak Ridge Leadership Computing Facility located in the Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

Contents

List of Figures

List of Tables

CONTROL AND INSPECTION OF DISTRIBUTED PROCESS GROUPS AT EXTREME SCALE VIA GROUP FILE SEMANTICS

MICHAEL JOSEPH BRIM

Under the supervision of Professor Barton P. Miller

At the University of Wisconsin–Madison

Tools and middleware are crucial to the effective use of large distributed systems. Middleware enables efficient utilization of resources, and tools help to diagnose and fix problems in distributed programs. A common requirement among tools and middleware is operating on groups of distributed processes and files, but prior work has failed to provide a solution that both addresses the key scalability barriers and is easy to use within new and existing software. In this dissertation, we present our four-part solution to these problems.

First, we introduce a new programming idiom, *group file operations*, that eliminates iteration and includes explicit data aggregation. The idiom extends familiar POSIX I/O operations and provides intuitive semantics, which eases its adoption. The idiom frees developers from considerations about how to access distributed resources or parallelize group operations. Thus, developers can focus on devising scalable data aggregations that reduce centralized analysis.

Second, we present a flexible and efficient way to construct file groups using *global name space composition*. Users specify how independent, distributed name spaces are combined to form a global name space. Composition of hierarchical name spaces is modeled as intuitive tree operations. A merge operation over a set of trees provides key semantics for efficient composition of directories that define group membership.

Third, we design the *TBON File System* (TBON-FS) that provides scalable group file operations

and global name space composition. TBON-FS leverages a tree-based overlay network to provide logarithmic scaling for multicast communication with remote servers and distributed aggregation of data.

Fourth, we present *proc++*, a synthetic file system for control and inspection of process and thread groups. proc++ improves upon existing process control interfaces by exposing new abstractions that hide context-sensitive information and reduce tool interactions with the operating system.

Together, these parts form a scalable platform for group operations on distributed processes and files. We evaluate the utility and performance of this platform in a suite of new tools and two widely-used software packages. Our evaluations show the idiom is easy to use and our platform provides excellent scalablility for both global name space composition and group file operations.

Chapter 1

Introduction

Demand for computational resources to solve extremely large problems by industry, government, and educational research scientists continues to drive the scale of existing and proposed high-performance computing (HPC) systems to new heights. According to the most recent Top500 list of supercomputers [56], over half of the systems contain more than 8192 processors, with the current leader having an astonishing 548,352 processors. Initiatives from around the world promise to continue HPC system growth [38]. Furthermore, data centers routinely contain tens of thousands of server hosts. These data centers provide the computational and storage backbone for e-commerce, cloud computing, data warehousing and mining, and Internet search.

Tools and middleware are crucial to the effective use of large scale distributed systems, yet they are often overlooked during the planning and deployment of these systems. Middleware such as runtime environments and distributed monitoring systems enable efficient utilization of distributed resources, while tools provide software developers and application users the ability to diagnose and fix performance or correctness problems in the programs that execute on the distributed system. Developers of tools and middleware for distributed systems face the daunting task of enhancing or redesigning

their software to operate at increasingly large scale.

Tools and middleware commonly perform operations on each member of a group of distributed processes or files. Each *group operation* involves a single process communicating with distributed hosts to apply the operation to each member, and collecting status results or data produced by the operations. Often, group operation status or data results are further processed to derive information that summarizes group behavior or to guide further operations on the group. In many tools and middleware, both the distributed operations and data analysis need to be completed in a timespan suitable for providing interactive functionality. For large distributed groups, however, the distributed communication and data processing required represent critical scalability barriers. Despite the common tasks involved in performing group operations on distributed processes and files, little attention has been paid to developing a common, scalable solution. As a result, each tool is forced to develop its own scalable solution, which leads to replication of effort and results in techniques that are not general enough for adoption by other tools.

Our central thesis is that by casting distributed resource access as operations on files in a global name space, and using a new *group file operation* idiom we have designed for eliminating iteration and encapsulating data aggregation during operations on file groups, we can develop a common, scalable solution for group operations on distributed processes and files. The resulting solution should enable tool and middleware developers to quickly create new scalable software or easily improve the scalability of existing software.

The group file operation idiom allows us to base the actions of tools and middleware during distributed group operations on a common set of tasks: defining the file group; reading or writing the group's data; and (optionally) processing data read from the group. The idiom also frees developers from considerations about how to efficiently perform distributed file access or parallelize operations across the group. Due to the idiom's flexibility, many tool activities can be mapped to group file oper-

ations. For example, group read operations can be used to gather system performance and configuration data, examine system or application logs, and query host or process status information. In each of these group read scenarios, group data aggregation can be used to further classify or transform raw, per-member data into summarized information for all hosts or processes. Group write operations are useful for writing messages to synthetic file systems providing operating system or process control (e.g., `sysfs` and `procfs` on Linux), as used in distributed system management and parallel debugging tools. Group writes can also be used for centralized management of shared-nothing clusters [114] where software and configuration updates must be applied to the local file systems of each host.

This dissertation describes novel techniques for group operations on distributed files and processes. These techniques have three desirable properties:

- **Scalability** - We develop techniques that are effective on the largest existing and upcoming systems, meaning systems containing at least tens of thousands of hosts and hundreds of thousands of processor cores. Scalable approaches to distributed group operations, and in particular group operations used for supporting interactive tools, must avoid linear-time behavior, such as iterations that are proportional to the size of the group, in favor of constant-time or logarithmic-time behavior. Since data processing time often scales linearly with respect to the size of the data, scalable group operations also must incorporate methods for reducing or limiting the amount of data gathered, preferably in a manner that also reduces centralized analysis.

- **Usability** - We start with familiar abstractions: file descriptors, processes as files, and hierarchical file paths. We extend these abstractions to provide primitives for efficient and scalable group operations on distributed files and processes. We define operations using our extended abstractions that have clear semantics, scalable implementations, and can be naturally combined in intuitive ways to support a wide variety of uses such as distributed process control, performance monitoring of applications and hosts, and distributed system administration.

- **Portability** - The landscape of large distributed systems is quite diverse with respect to both the hardware and system software employed. To enable widespread use, we base our techniques on commonly available functionality (e.g., POSIX I/O and user-level overlay networks), making extensions only when necessary. Further, we avoid system-level modifications in favor of user-level approaches.

1.1 Motivation: From Process Groups to File Groups

Surprisingly, the original problem that motivated our development of the group file operation idiom had no direct tie to file access. Rather, we were investigating approaches for scalable control and inspection of distributed process and thread groups. Our goal was to define a common interface to group control and inspection operations that could be easily integrated into new and existing tools and middleware. This interface had to meet our criteria for usability and portability while also supporting use of scalable underlying mechanisms.

Process control and inspection is the ability to manipulate or examine process (or thread) state. Process state is a general term denoting the hardware and software resources used by a program during execution. Group control and inspection is used by several classes of tools and middleware. For example, application runtime environments need to control all the processes in a distributed application, resource and performance monitors need to perform group process inspection, and online tools such as distributed debuggers require both control and inspection of distributed process groups. (A more detailed discussion on the classes of tools and middleware and their requirements for distributed process group control and inspection is presented in Section 2.1.2.)

Many operating systems, including Plan 9 [93], various Unix systems [40,58], and Linux, provide a synthetic (or pseudo) file system that enables process control and inspection. The term “synthetic” refers to the fact that directories and files found in the file system do not exist as data on a storage device, but instead are generated dynamically by the operating system. These file systems, collectively

known as *proc file systems*, represent each process running on a host as a directory containing various files that can be used for specific control or inspection tasks on that process. Some proc file systems also provide directories representing the kernel threads (also known as lightweight processes) associated with each process.

Because proc file systems are widely used, tool and middleware developers are familiar with performing process control and inspection via a file system interface. Using file system interfaces has additional benefits that meet our goals for simplicity, flexibility, and portability. File system interfaces are simple to use and support for file access is ubiquitous in the languages commonly used to implement tools and middleware. A file system interface is agnostic to the format of data, which provides the flexibility to operate on files containing binary data, text, or combinations of both. File system interfaces based on POSIX I/O are also portable, a key requirement given the heterogeneity of the current distributed system landscape. Due to these benefits, we focused our attention to providing control and inspection of distributed process and thread groups via a file system interface, which maps to the more general problem of scalable operations on distributed file groups.

Providing scalable group operations on large sets of distributed files is difficult due to two primary deficiencies in current file system interfaces and implementations. First, file systems have no notion of defining file groups and performing operations on those groups. Second, existing distributed and parallel file systems use client-server communication architectures that are not scalable for a single client operating on a large group of distributed files. We discuss each of these problems in more detail.

The lack of group abstractions and operations in existing file systems leads to iterative behavior in tools that need to apply some sequence of operations to a large file group, as the tool must perform the operations on each member. Such iteration serializes the time to complete operations for each member, resulting in group operations having completion time that scales linearly with the group size. When files are distributed, the additional cost of remote communication further slows the time to complete

the group operations.

Existing distributed and parallel file systems are designed with a focus on scalability in terms of aggregate data bandwidth and storage, and in number of concurrent file system clients. The common assumption is each client will interact with one (or a few) file servers, and thus point-to-point communication channels between the client and server(s) are used. However, we target the widespread use case of a single client tool that needs to perform group operations on a large set of files located across thousands of distributed servers. The choice of per-server communication channels in this case is non-scalable, as it leads to iteration over servers and may exhaust OS resources (e.g., running out of open file descriptors due to too many open socket connections).

1.2 Contributions

This dissertation presents several contributions that address the problems of performing scalable operations on large distributed groups of files and processes. In this section, we provide an overview of each contribution; detailed discussions are available in subsequent chapters.

Group File Operations We introduce *group file operations*, an intuitive new idiom for eliminating iterative behavior when a single process must apply the same set of file operations to a group of related files. The keys to the idiom are explicit identification of file groups using directories as the grouping mechanism, the ability to name a file group as the target for POSIX I/O operations such as `read` and `write`, and explicit semantics for aggregation of group data and status results. Group file operations provide an interface that eliminates forced iteration, thus enabling scalable implementations on distributed files. Given underlying mechanisms for distributed data aggregation, group data and status results can be processed in a distributed fashion that eliminates or reduces the need for centralized analysis or large data storage.

Name Space Composition Group file operations are a solution to the obvious scalability limitations of iterative operations on distributed files. Forming file groups efficiently is an equally important but

less obvious problem, as it is a precursor to the use of group file operations. To avoid iteration and provide scalable definition of file groups, we use a two step process that relies on file system name space composition. First, tools provide a specification that is used at each independent server to generate a custom view of the server's local name space; the custom views are structured to naturally support efficient construction of a global view. Second, the custom server views are merged into a global name space using a composition that automatically groups related files (i.e., files providing similar data or functionality).

Name space specifications are written using a new language called *FINAL*, the **FI**le **N**ame space **A**ggregation **L**anguage, that models hierarchical file system name space composition as operations on trees. Inspired by the private name space manipulation operations of Plan 9 [94], FINAL provides flexible composition semantics based on common tree operations such as copying, pruning, and grafting. To support efficient composition of many trees without explicit iteration, FINAL introduces a merge operation over set of trees, and supports customizable resolution for the name conflicts that can occur during a merge. FINAL's merge operation provides the key semantics required for composing file group directories containing files from independent name spaces. Further, the merge operation maps nicely to scalable distributed composition within a tree-based overlay network (TBON); thus, we can *use trees to compose trees*.

TBON File System We design and evaluate *the TBON File System* (TBON-FS*)*, a group file system that leverages the logarithmic communication and distributed data aggregation capabilities of tree-based overlay networks to support scalable group file operations on distributed files and scalable global name space composition. Similar to NFS [105], TBON-FS does not provide any data storage, and simply acts as a proxy at each file server to access the locally visible file systems. TBON-FS leverages MRNet [101], a general-purpose TBON API and infrastructure, for scalable communication of file system operation requests to thousands of independent servers, scalable aggregation of group data and

status results from group file operations, and scalable composition of its global name space. TBON-FS also supports synthetic file systems within servers as a means for providing file-based interfaces to arbitrary tool functionality. Because TBON-FS is implemented entirely at user-level, it is easy to deploy on a wide variety of distributed systems.

proc++ We design and evaluate *proc++*, a new synthetic file system tailored for use in scalable control and inspection of distributed process and thread groups. The design of proc++ is based on the proc file system found in Solaris, with extensions to provide abstractions that naturally encapsulate interaction intensive operations, such as breakpoint management and gathering stack traces, as a means to reducing the number of interactions between the tool and the process control layer. The primary challenges in designing proc++ were to identify the tool behaviors that were hindered by interactions with existing process control interfaces, and to develop appropriate abstractions that enable group file operations to take advantage of newly provided tool-level capabilities. The difficulty of the latter challenge should not be underestimated, as the placement of the abstractions within the layers of the distributed tool hierarchy is key to enabling the parallelism necessary to scale, and choosing the proper amount of tool functionality captured by the abstractions requires careful thought to avoid limiting their utility.

When combined as shown in Figure 1.1, our contributions form the basis for a scalable platform for group operations on distributed files. We evaluate this platform in terms of utility and performance in three case studies. The first case study focuses on the use of group file operations to develop several new scalable tools for distributed system administration and monitoring. The second and third case studies relate our experiences in integrating group file operations into two existing, widely-used software packages: the Ganglia distributed monitoring system [67], and the TotalView parallel debugger [100]. Our studies show our techniques to be highly scalable, requiring just a few hundred milliseconds to compose a global name space from forty thousand independent hosts or to perform group file operations on over 200,000 distributed files or processes.

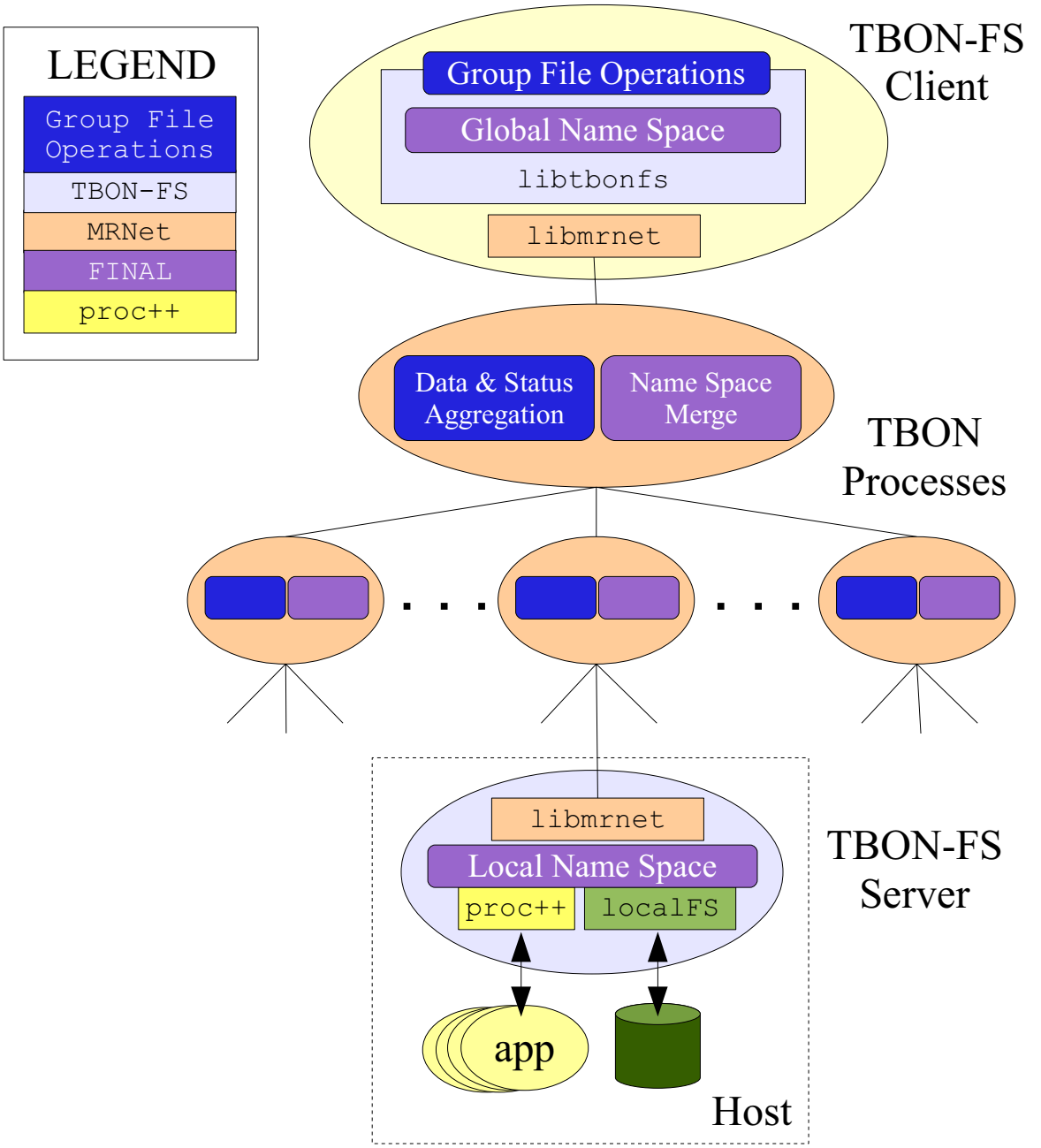

Figure 1.1 Platform for Scalable Group Operations on Distributed Files and Processes
A TBON-FS client uses group file operations on files within a global name space composed by merging the local name spaces of independent, distributed TBON-FS servers. TBON-FS employs a tree-based overlay network (MRNet) to provide scalable communication and distributed data processing for group file operations and global name space composition. The client has complete control over the aggregations used for combining group status and data results from group file operations, and provides a FINAL name space specification that is used to organize the local name spaces at each TBON-FS server. Synthetic file systems such as `proc++` can be loaded within TBON-FS servers to provide file-based access to custom tool capabilities

1.3 Organization

This dissertation contains ten chapters. Chapter 2 presents a survey of tools and middleware the use group operations on distributed files, and then discusses related work in two areas: group operations on distributed processes and files, and composition of file name spaces. In Chapter 3, we describe the group file operation idiom, its core abstractions and semantics, and interesting idiom extensions that increase its utility. Chapter 4 presents techniques for flexible and scalable composition of name spaces from independent file servers. The design and implementation of TBON-FS is discussed in Chapter 5, focusing on its support for scalable group file operations and custom global name space composition. In Chapter 6, we present proc++, a new synthetic file system that enables scalable group control and inspection of distributed process and thread groups. Chapter 7 relates our experience in using group file operations for the development of new scalable tools for distributed system administration and monitoring. Chapters 8 and 9 discuss our integration of group file operations into two real-world tools, and evaluate the resulting scalability benefits. Chapter 8 focuses on our work with the Ganglia distributed monitoring system, and Chapter 9 discusses our efforts using the TotalView parallel debugger. We conclude in Chapter 10 with a summary of our contributions and a discussion of future research directions.

Chapter 2

Related Work

The research in this thesis focuses on providing techniques for group operations on distributed files and processes at extreme scale. To motivate our work, we begin in Section 2.1 with a survey of existing tool and middleware requirements for group operations on distributed files and processes. The survey confirms the widespread use of distributed group operations, yet sparse attention to scalability.

Next, we discuss prior research that relates to our contributions in two areas: group operations on distributed files and processes are covered in Section 2.2, and composition of file name spaces is discussed in Section 2.3. We focus on previous efforts that provide useful abstractions and semantics for group operations and name space composition, and approaches that address scalability concerns for large distributed systems.

Finally, in Section 2.4 we compare group file operations to prior work in distributed and parallel file access. We show that group file operations have a distinct client-server relationship that presents unique scalability concerns not addressed by previous systems.

2.1 Tool and Middleware Requirements for Distributed Group Operations

We surveyed existing tools and middleware to determine their requirements for group operations

on distributed files and processes, as well as methods employed to make such operations scalable. For both distributed files and distributed processes, we enumerate common uses for group operations and the primary factors for each use that should be addressed by scalable solutions.

2.1.1 Requirements for Group Operations on Distributed Files

Group operations on distributed files are used commonly in tools and middleware for monitoring processes and hosts, system administration, and processing system and application logs.

Process and Host Monitoring. Tools and middleware often access proc file systems to collect resource usage and status information on a per-process basis. The Linux proc file system also contains files that provide host-level resource usage and status.

Parallel application performance monitors [28,80] and distributed resource management systems [52,89,95,110,117] read groups of these proc files to obtain processor and memory utilization information for the parallel application processes. Performance monitors may sample resource utilization at sub-second intervals to provide a detailed performance history, while resource management systems are usually interested in average utilization for an entire job and thus sample less often, typically on intervals of several minutes.

Distributed monitoring systems such as Ganglia [67], NWPerf [71], and Supermon [112] place monitor processes on each host of a cluster. The monitor processes extract data corresponding to predefined host performance metrics such as processor, memory, disk, and network utilization from proc files. Metric data from all hosts is collected to a master host, where it may be aggregated into cluster performance summaries. To provide metric performance histories, Ganglia and NWperf store the per-host and cluster summary data for each metric in databases at the master host. The rate of file reads for collecting monitoring data is usually restricted to limit overhead and perturbation of applications running on the hosts. Common intervals vary from several seconds up to a few minutes. Typically, the size

of monitoring data read at each host is relatively small, on the order of a few kilobytes or less.

Ganglia, NWPerf, and Supermon have addressed scalability in various ways. Both Ganglia and NWPerf use IP multicast as a publish-subscribe mechanism for distributing new metric data from each monitor process. Ganglia's monitor processes read metric data at pre-determined intervals, while NWperf initiates metric reads for monitor groups by sending a UDP packet to each monitor process. Ganglia's monitors are members of a multicast receiver group, which results in data for all hosts being available at any host. Ganglia uses a central collector process per cluster that periodically contacts a representative monitor to collect the data for all cluster hosts. As noted by Ganglia's designers [67], its use of IP multicast is not scalable for large clusters containing thousands of hosts, since the data sent from every source is delivered at each multicast receiver. This leads to a large network and processing overhead at each receiver to handle the total volume of data for the group. In contrast, NWPerf uses a single central collector process for each cluster that subscribes to the mulitcast group. For larger clusters, NWPerf suggests running several collector processes and dividing monitors among multiple multicast groups with one collector process per group. However, this approach requires each collector to forward data to a central database host that becomes the bottleneck. As an alternative to multiple multicast groups, NWPerf proposed, but has not implemented, using a tree-based collection to increase scalability. Supermon uses a tree of monitor processes to gather data read from files on a group of hosts. Data is collected only upon request by a client that contacts the root of the monitor tree. None of these prior systems address scalable aggregation of metric data for large clusters, which results in high data processing and storage times at the central collection host for Ganglia and NWPerf or the client for Supermon.

System Administration. Group operations for system administration are commonly found in tools for managing large groups of similarly configured hosts [18,22]. For smaller distributed systems, the use of shared file systems is sufficient for centralized management of software and configuration files.

When the distributed system contains thousands of hosts, shared file system servers can become a bottleneck during file access storms, which are short periods where many hosts access the same files. These storms are common during system boot when configuration files are accessed, and during the launch of programs across many hosts when the executable and shared libraries are accessed. Administrators for large distributed systems may choose to have each host use a local disk for storage of files that are likely to be accessed during a storm. This choice results in the need to keep the local files synchronized across hosts when software packages are installed or updated, or the system configuration is changed. Similarly, lack of a shared file system requires group file distribution for staging user applications, libraries, and data files in preparation for parallel job execution. Group file collection may be employed to gather the data or log files generated by a parallel job to a central location.

Previous tools designed for synchronizing files across groups of hosts include MPISync [35], Cfengine [22], and the C3 tools [18]. MPISync uses MPI to broadcast files from a master host to sets of divergent slave hosts. Current MPI implementations either use broadcast capabilities provided by specialized networks in HPC systems, or construct broadcast spanning trees using efficient point-to-point communication. Both approaches provide the necessary scalability for file broadcast on large systems. Cfengine is normally configured to have slave hosts copy updated files from a master via a shared file system. On large systems, special care must be taken to schedule slave updates in a way that avoids all slaves accessing the master concurrently. A typical approach is to form sub-groups of slaves and stagger copies for each sub-group. This approach may require several minutes to update all slaves. Such latencies may be acceptable for rare updates to installed software or configuration files, but they are not tolerable for on-demand file staging. The C3 tools use multiple concurrent `scp` or `rsync` processes to transfer files or updates, and the master host quickly becomes a bottleneck for even modest size clusters with a hundred hosts.

Log Processing. Many tools gather and process data from system and application log files from dis-

tributed hosts [47,75,121]. System logs are monitored to identify hardware and software configuration problems, misbehaving applications that are denied access to resources, and operating system failures. Application logs are typically monitored to view completed activities or discover errors for long running computations or services.

By correlating related events across distributed hosts, administrators can identify a wide range of problems, from systematic configuration issues to security-related events such as distributed intrusion or denial-of-service attacks. Correlated log events can also help administrators predict imminent failures, which allows for proactive mitigation. The primary obstacles to distributed log event correlation are collecting logs from many hosts to a central location, and performing the correlation analysis over a large log data corpus. Swatch [47] and logtail [121] address the collection of one or more logs from each host to a central host. Both systems simply forward log data to the central host, and neither system is designed for use on distibuted systems containing more than a few tens of hosts. Swatch and MultiTail [75] also support custom searches and filtering based upon regular expression matching to reduce the data that needs analysis. Data reduction based upon equivalence is a key technique for scalable correlation of distributed log events, and we leverage this technique in several demonstrations of tools for distributed system administration. MultiTail is an interactive tool that displays each log in a separate console window, and thus is not suitable for monitoring large distributed systems.

2.1.2 Requirements for Group Operations on Distributed Processes and Threads

Group operations on distributed processes and threads are common in parallel runtime environments, tools for parallel application performance montioring and steering, and distributed debuggers. We divide our discussion among two classes of group operations: control of process and thread groups, and accessing data in the memory or registers of a group of processes.

Process & Thread Control. There are three common types of group control actions: (1) bringing a

group under control by launching or attaching to processes, (2) issuing commands to stop, continue, or signal, and (3) handling events generated by process or thread groups.

Group process launch is often used to start application or tool processes on HPC systems, but can also be used for system administration tasks such as running commands on a group of hosts. During group process launch, the program arguments and environment settings must be communicated to all participating hosts. The program executable and dependent libraries also may need to be distributed if not located at each host or on a shared file system. Systems supporting group launch may also handle standard I/O streams for all processes; standard input is directed to one or all processes, and standard output and error are annotated by source process and aggregated into group streams.

Attaching to a process group is typical in tools for debugging or monitoring parallel applications. Before attaching, information including the host and process identifiers for all target processes must be collected by querying the parallel runtime environment. Once the processes have been identified, the tool must distribute this information to tool daemons on each target host. In some situtations, the tool may use group process launch to first start the tool daemons before distributing the information. Each daemon attaches to colocated processes using the standard OS attach mechanisms.

Group process launch or attach is an important yet relatively infrequent operation for parallel application tools. A single tool use session usually includes only one group launch or attach. Still, the latency of these operations is highly visible to interactive users, as they must wait for completion before using other tool functionality. The most frequent use of group process launch is found within parallel shells [18,29,36,53,70,90,119], where users enter commands to be executed across a set of hosts. The interval between issued commands can be as small as a few seconds. Scalable mechanisms should focus on reducing the latency to execute or attach to large numbers of distributed processes, and providing aggregation of application output and error streams.

After launching or attaching to processes, tools and middleware require the ability to perform job

control such as stopping, continuing, and signaling sets of processes or threads. Parallel application performance tools such as Paradyn [68], DPCL [88], and MATE [72] use group control operations to stop process groups before performing dynamic instrumentation and then continue the groups afterwards. Similarly, distributed debuggers including TotalView [100] and DDT [2] support group stop and continue operations. Group stop is commonly used before examining state or inserting breakpoints or watchpoints. Group continue is used to restart execution, and can be used internally by the debugger to support group step operations. Given their interactive use within debuggers, the latency to issue commands is the main scalability concern for group control operations.

While under their control, tools and middleware receive and process events generated by processes and threads. Although events are generated asynchronously, there are situations where large sets of processes or threads will generate events in a short time period. For example, the processes of a parallel application typically finish execution together, and each will generate an exit event. During parallel application debugging, many threads may hit the same breakpoint in short succession. In such situations, it is useful (and sometimes essential) to identify similar events across processes or threads and handle them as a group.

Among current approaches for scalable process and thread group control, group communication via a tree or ring is most common. BProc [48], yod [17], CRE [111], and ALPS [57] use tree-structured communication to distribute group process launch requests and supply standard input, while MPD [23] uses ring-based communication for this purpose. CRE, ALPS, and MPD use trees to collect standard output and error from all processes for display at the originating terminal, but provide no output reduction, although MPD provides annotation of output lines with source process.

Accessing Registers and Memory. Distributed debuggers [2,5,16,49,100,122] and parallel application performance tools [68,72,74,82,88,98,102,107,108] often view or modify the contents of processor registers or memory for a collection of application processes or threads.

Distributed debuggers use group register reads to provide state or variable information for stopped threads, while performance tools sample registers to perform statistical profiling or collect performance counter data. Group register write operations can be used to modify the behavior of a set of stopped threads, or by performance measurement tools to configure performance counter settings [82]. In group register reads, the set of registers to be read is distributed to all group members, and subsequently read values are collected. For group register write operations, the set of registers and associated values to write are communicated to all members.

Distributed debuggers read and write process memory to support group breakpoints and watchpoints, stackwalks, instruction disassembly, and inspecting or updating the values of variables and function arguments. Parallel application performance tools can use group memory reads to sample performance data across processes. Tools using dynamic instrumentation [68,72,74,82,88,108] read memory across processes to identify instrumentation points and use group memory writes to insert instrumentation code. In group memory access operations, a request indicating the target address and amount to be read/written is distributed to all group processes. For group write operations, the data to write is also distributed.

Scalable approaches for performing group access to memory or registers should address both request distribution latency and aggregation of inspection results. The Interactive Parallel Debugger (IPD) [16], part of the Performance Monitoring Environment for the Intel Paragon [98], used a broadcast spanning tree that took advantage of the mesh communication network to efficiently distribute monitoring and control requests and collect responses. Paradyn and DPCL have been updated [102,107] to use the TBON communication facilities provided by MRNet [101], to enable scalable communication between tool front-end and back-end processes. As a result, these performance tools are able to scalably send group requests. For aggregation of group memory reads, Paradyn computes checksums over memory contents and uses a custom binning aggregation to create equivalence classes

for processes with the same checksum, and then only transfers the actual data once for each class. The STAT [5] debugging tool also uses MRNet for scalable aggregation of the stack traces of parallel processes into a combined call tree. DDT [2] uses a custom tree communication architecture for scalable request distribution and aggregation of equivalent data and stackwalks.

2.1.3 Summary of Requirements for Scalable Group Operations

From this survey, we can see that group operations on distributed processes and files are widely used in tools and middleware. Unfortunately, there has been little prior work that directly addresses this use case in terms of providing a general, scalable solution that is suitable for use within a wide variety of tools and middleware. Our goal in developing the group file operation idiom and the TBON-FS group file system was to rectify this deficiency by providing such a solution.

Among prior uses, we have identified two common scalability goals. First, group operations used for process control or file writes typically involve small data payloads, and thus the emphasis for scalability in these operations is minimizing the latency of distributing data to support interactive tools. Second, group operations used to collect data from files or processes can benefit from distributed aggregation as a means for reducing the memory and computational load associated with data analysis.

Many tools and middleware still use non-scalable methods such as multiple point-to-point connections from a single manager process to distributed daemon processes [90,95,100,110,117] or parallel invocation of a remote shell [18,90,119]. Previous approaches that do consider scalability lack generality in the operations supported, and are not structured so that other software with similar needs can reuse scalable mechanisms. The two most common approaches to achieving scalability include arranging tool processes in ring or tree-based overlay networks. Typically, a master tool process that controls the actions of slave tool processes resides at the start of the ring or root of the tree.

In ring overlays, command distribution requires `O(N)` steps to deliver data to `N` slaves. However, many commands can be pipelined to hide some of the latency of command distribution. Rings also

require that every process in the ring participate in distributing commands, even when the target group is a subset of the ring. Rings are not suited to data collection for large groups, because data from early in the ring is copied many times and the amount of data that must be copied increases with the number of ring processes visited. Because of the limitations of rings for data collection, systems like MPD that use a ring for command distribution may employ a tree-based overlay network for data collection.

Tree-based overlay networks (TBONs) offer a solution to avoiding linear behaviors found in rings. Data broadcast and gather between the master and slaves benefits from the logarithmic properties of tree-based communication; for a balanced tree with fanout `F` and `N` leaves, broadcast and gather require $O(\log_F(N))$ data transmissions. TBONs can also be used to reduce centralized analysis load at the master by distributing the analysis processing among the overlay processes, assuming the analysis is amenable to distributed hierarchical computation.

Several systems have used TBONs to take advantage of the communication and distributed data processing benefits of trees. Most systems use special-purpose TBONs that are not intended for use by other tools [2,16,17,23,48,57,67,111,112]. In contrast, Lilith [39] and Ygdrasil [9] were designed as infrastructures for building parallel tools, but each makes some assumptions regarding tree topology or communication structure that limited widespread adoption. The Multicast Reduction Network (MRNet) [101] is a general-purpose TBON API and infrastructure that gives tools complete control over the tree topology, the placement of overlay processes within a distributed system, the content of data messages sent between the master and slaves, and the aggregations employed when gathering data. Due to its flexibility, MRNet has been successfully used as the scalable substrate in a wide variety of tools [5,60,62,76,82,101] and parallel applications [6]. We also use MRNet as the scalability vehicle for group file operations and TBON-FS.

2.2 Group Operations on Files and Processes

Our research focuses on the use case of a single program performing group operations upon a large

set of distributed files or processes. We review previous efforts that define useful abstractions and semantics for such group operations.

Steere's dynamic file sets [113] share the same motivation as group file operations for eliminating serialization imposed by the file system interface when operating on groups of distributed files. To create a dynamic set, a user provides a specification that names the group members. Sets provide an interator iterface that returns file descriptors one at a time. The benefit from dynamic sets versus standard file operations comes from using a threaded distributed file system client to prefetch whole files in parallel. The iterator interface returns file descriptors in an order based on when fetched data arrives. This allows operations on fully-fetched files while the system gathers other files. Scalability for large groups is hindered by both iteration and fetching all data to the client.

Google's MapReduce system [34] provides fault-tolerant parallel processing of large data sets containing gigabytes or terabytes of data. Each data set is partitioned into many fixed-size chunks that are stored across hundreds or thousands of Google File System (GFS) [44] servers. The programming idiom of MapReduce involves just two user-defined operations: a map operation that is applied to input data to produce key-value pairs, and a reduce operation that aggregates values from pairs with similar keys. Aggregated data is stored as new files on GFS. Considering chunks as files permits a MapReduce operation to be viewed as a group file operation where both map and reduce are combined into a single aggregation. The MapReduce system uses a master-worker architecture. The master is responsible for starting map and reduce tasks on the GFS server hosts and distributing information describing the input and output files to these remote tasks. This architecture results in startup latencies for each MapReduce computation that are on the order of one minute for a cluster containing 1,800 hosts [34]. Such startup overhead makes MapReduce a poor option for use in interactive tools that require fast aggregation of data from large file groups.

Semantic file systems [3,42] allow users to retrieve and specify context that describes the type of

data contained in files and relationships between files. Some semantic file systems [42] allow users to query file data and meta-data to dynamically create directories containing groups of files matching some criteria. Unfortunately, these techniques for defining file groups based upon attributes and data contents are not currently supported in distributed file systems. We believe that group file operations can be used to support such semantic techniques on distributed, independent file systems.

2.3 File Name Space Composition

Group file operations are designed for scalable operations on files located across thousands of independent hosts. The first step in using group file operations is to define file groups, which requires a method of naming distributed files. File name space composition is a fundamental approach for adding files from distributed name spaces to a local name space. We review prior techniques for composing hierarchical file name spaces and summarize their deficiencies when composing global name spaces that comprise thousands of independent name spaces.

For over forty years and starting with Multics [33], operating systems have supported construction of a rooted, hierarchical file name space from one or more underlying file services. The original UNIX `mount` system call is an early example of composition that inserts a file system name space at a specific leaf path in the system's "root" name space [99]. Later versions of `mount` removed the leaf path restriction and instead replace the name space subtree rooted at the mount path with the file system name space. File system mounts are the coarsest form of composition, as entire name space trees are inserted or removed. Name space unions, also known as union mounts, are an extension of traditional mounts that results in files from two name spaces being visible in the combined name space. Such unions are supported by many systems [91,94,115,123], and can be viewed as overlaying one name space on another. Name space unions can have shallow or deep composition semantics. In shallow name space unions, the children of the root of each name space are unified. In deep unions, the entire name space trees are merged (i.e., every directory that exists at the same path in both name spaces will

be present in the combined name space, and will contain the corresponding directory entries from both name spaces). Because traditional and union mounts both require pair-wise composition of name spaces, they are prohibitively expensive for generating a global name space containing thousands of independent name spaces.

Private name spaces are custom views of a default system name space. Common uses for private name spaces include improving user convenience by placing heavily used files near the root of the name space or excluding unused files, and implementing security through isolation of name spaces or omission of sensitive files. Several previous distributed file systems, including Cedar [43], Jade [97], Prospero [78], and Chirp [116], let users create private name spaces for distributed files.

The Cedar File System uses attachments to provide local names for remote files. Remote files can be named using a system-wide path name that includes the remote file server. Attachments are implemented similar to symbolic links by storing the full path to the remote file in a local directory entry.

The Jade File System uses two types of names spaces, logical and physical, to allow users to customize their private name space. A logical name space contains skeleton directories that serve as a virtual structure for organizing user-directed mounts of physical name spaces exported by file systems. Users may also mount portions of another user's logical name space. Jade supports an extended version of shallow unions by allowing users to mount multiple name spaces at the same path.

The Virtual System Model underlying the Prospero File System provides user-centric views of a global name space. Prospero treats directories as a collection of links. A link contains a reference to a path in the name space of a file service. Union links are provided that can be added to a directory to merge the contents of another directory. Filters can be associated with a link to provide a new view of a target directory. Since a directory is a collection of links, filters return such a collection. Filters may create new virtual directories within their views. Prospero supports a few built-in filters that create virtual directories based on the attributes of files in a directory, and also allows users to write filters as

shared library functions using the C language.

Chirp is a user-level distributed file system for use in Grid or wide-area environments. Chirp provides three distinct name spaces: a global name space with a directory per server, a private name space created by specifying a list of mounts that associate private name space paths with paths in the global name space, and shared name spaces that provide a virtual directory structure that has links to files in the global name space as leaves in the name space.

Our FINAL language has its roots in the flac language [125] designed to provide mobile applications with an unchanging name space. Flac is a specification language that is used to describe a private name space provided to applications that interface with the flac runtime environment. The flac language uses a name space tree abstraction and tree combination operations for composing name spaces from local and remote file services. Specifications describe both the organization of the name space and the methods of file access to use when the application is executing in different contexts. Possible context changes include: (1) migration of the process to a new host, (2) a change in the network connectivity of the host (e.g., moving from a wireless to wired network), and (3) disconnection of the host from the network. When a context change occurs, the flac runtime transparently adjusts the access methods employed.

In summary, prior approaches are ill-suited to the construction of global name spaces that comprise tens of thousands of name spaces from independent hosts. Many prior approaches were designed for composing a few name spaces, and thus use inefficient composition techniques such as pair-wise composition or fine-grained directory entry manipulation. These techniques lead to iteration when composing large sets of name spaces. Prior systems that construct a global name space often avoid fine-grained composition by using partitioning to place each independent name space in a separate portion of the the global name space. Partitioned name spaces lead to iteration when defining file groups spanning multiple hosts. Finally, prior approaches adopt inflexible semantics for specific com-

positions, even when many valid choices may exist. For example, name space unions may have shallow or deep semantics, and may treat duplicate names using renaming or overlay semantics that hide duplicates. We designed FINAL, our language for describing custom name space composition, to address the limited flexibility and scalability of prior approaches for composing a global name space.

2.4 Distributed and Parallel File Access

Group file operations target the use case of a single client interacting with a large set of distributed and independent file servers. As the survey of tools and middleware in Section 2.1 confirms, this use case maps to a wide variety of tool behaviors. However, this use case is distinctly different from the common uses of distributed file systems, parallel file systems, peer-to-peer file sharing, and Internet file storage and caching. We discuss the key differences between group file operations and these prior systems, focusing on access patterns, data storage, and scalability considerations.

Distributed file systems [43,50,78,97,105] provide independent clients access to shared files. Many of these systems directly manage the disks containing data, while NFS [105] provides only a proxy service for accessing file systems local to the server. Each client typically interacts with one or a few servers containing the desired file volumes. For large distributed systems, each server may handle requests from tens or hundreds of clients. Designers of distributed file systems focus on scalability in terms of number of concurrent clients and aggregate data storage. Servers are added as necessary to provide additional service and storage capacity, and files may be replicated across many servers to distribute load from commonly accessed files [50].

Parallel file systems [24,30,51,64,106,109] are designed to provide many cooperating clients with shared access to large data sets that span many servers. Clients are typically parallel application processes that perform concurrent computation using a small portion of an input data set. Clients may also generate new data sets as a result of their computation. Each client interacts with one or a few servers to read or write the relevant data set portions. Similar to distributed file systems, parallel file systems

are designed to scale in terms of clients and storage, as well as aggregate I/O bandwidth. The latter scalability consideration limits the number of concurrent clients for a given server.

Peer-to-peer (p2p) file sharing systems [25,26], Internet file storage systems built using p2p overlays [32,59,103], and Internet file caches [1,54] are primarily used for wide-area distribution of frequently accessed, yet rarely updated files to thousands, or even millions, of clients. From a client's perspective, a majority of these systems provide read-only file access. Systems supporting writes typically allow only whole-file replacement [59]. Similar to a parallel file system, a client may interact with one or a few file servers that contain portions of a given file. Depending on the popularity of files, the number of simultaneous clients may range from tens to thousands. Popular files are often replicated across many servers to reduce the chance that a server will be overwhelmed by client requests. Scalability in p2p systems is tied to the number of active peers willing to serve. Internet caches tend to be deployed in a more structured fashion that attempts to place files on servers within geographic regions as a means to limit the latency of document retrieval [1].

Compared to previous systems, group file operations have an inverted client-server ratio where a single client accesses many servers, rather than many clients accessing one or a few servers. This inversion is apparent when considering that the hosts containing files of interest to tools and middleware are often the same hosts where traditional distributed and parallel file system clients reside, such as the computational nodes in an HPC system. Thus, the primary scalability concerns for group file operations are access to a large number of servers, and aggregation of data gathered from many servers. Similar to NFS, group file operations are also best viewed as a proxy for accessing memory-based and disk-based file systems on distributed servers, rather than a file system that directly manages disks.

Chapter 3

Group File Operations

Group file operations are a new programming idiom for applying the same operations to each member of a group of files using an intuitive interface based on POSIX I/O. The idiom eliminates explicit iteration during group operations, which enables the use of scalable mechanisms to implement operations on large distributed file groups. In this chapter, we first introduce the key abstractions and semantics of group file operations, and then demonstrate their use in simple examples. Next, we extend the idiom to enhance its utility; these extensions were motivated by practical experiences during the tool and middleware case studies discussed in Chapters 7, 8, and 9.

3.1 Abstractions and Semantics for Group File Operations

The primary abstraction in the group file operation idiom is the *group file*, which we define as a set of open files that are operated upon as a single entity. Section 3.1.1 covers the establishment of group files using directories as the grouping mechanism and a new `gopen` operation that creates a handle for use in subsequent group file operations. Section 3.1.2 describes the operational and error semantics of using group files with existing POSIX I/O operations, and introduces new operations for controlling the aggregation of group status and data results that are generated during group file operations. We

provide a few examples of group file operation use in Section 3.1.3 to demonstrate the idiom's simplicity and flexibility. Section 3.1.4 addresses the issues that arise from sharing group files between group-aware and group-unaware programs.

3.1.1 Establishing Group Files

Directories are a natural and existing file system mechanism for grouping files. Placing files in the same directory often implies a logical association, and thus there is a good chance that files in the same directory will be operated on as a group. If a user wants to group existing files not already located in the same directory, a new directory whose name identifies the group can be created. To add or remove members, files are added to or removed from the group directory. Symbolic links can be used to add existing files without moving or copying.

The key operation we have defined to enable group file operations is `gopen`, a new directory open operation that on success returns a *group file descriptor* (gfd) to serve as a handle to the group file. Table 3.1 documents the `gopen` interface, which is modeled after POSIX `open`. Using `gopen` can be considered equivalent to calling `open` on each file entry in the named group directory, using the specified access flags and creation mode. A group file operation is performed by passing a gfd to a POSIX I/O operation, such as `read` and `write`, that has a file descriptor operand. We discuss the semantics of group file operations in the following subsection.

A gfd is essentially a view of the set of files that existed in the group directory at the time of the `gopen` and were successfully opened. While `gopen` is executing, the named directory's contents cannot be changed. When `gopen` completes, directory entries may be added or removed to aid the creation of additional groups with similar membership. For example, after operating on a particular group file, a program may identify a subset of the group upon which it wants to operate as a new group. Rather than requiring a new directory to be created containing the subset's files, the program can simply remove files and call `gopen` again. Since the contents of a group directory can change after a group

`int gopen(const char* dirpath, int flags, mode_t mode)` Opens all files in the group directory located at `dirpath` using specified access `flags` and creation `mode`. Returns a group file descriptor, or `-1` on an error. Note that sub-directories in the group directory are ignored.
`int gsize(int gfd)` Returns the number of files in the group specified by `gfd`, or `-1` on an error.
`int gfiles(int gfd, char** files)` Copies directory entry names corresponding to files in the group specified by `gfd` into the user-allocated array of character buffers `files`. The `files` array should be allocated to contain `gsize(gfd)` character buffers, each able to hold a maximum length directory entry name. Returns zero on success, `-1` on an error.

Table 3.1 New operations for managing group files: Interface and Description

file is established, the file system is not an appropriate source of accurate information about the group's membership. Thus, new operations are required to obtain information about the group. Table 3.1 contains the interfaces of two new operations: `gsize` is used to retrieve the group's size, while `gfiles` provides a list of member files.

Special consideration is given to errors that occur during `gopen`. Two modes, *default* and *best-effort*, differentiate how failures in opening individual files affect `gopen`'s completion status. In default mode, `gopen` will succeed only when all the files in the named directory can be opened. In best-effort mode, which is specified by including a new `O_BESTEFFORT` value in the flags, `gopen` will succeed when any of the files are successfully opened. In both modes, a failed completion will return `-1` and set `errno` to the error code `E_GROUP`.

3.1.2 Semantics of Operations Using Group Files

Our goal in defining the semantics of group file operations is to provide intuitive behavior that can also take advantage of scalable methods when operating on distributed files. Group file descriptors allow group file operations to use existing, well-understood POSIX I/O operations. However, existing operations are designed for operating on individual files, and therefore our challenge is to define

semantics for these operations when used with a group file. We follow four guiding principles in defining the semantics for group file operations:

1. Leave existing POSIX operation interfaces unchanged to maintain familiarity and portability, and introduce new group-specific operations only when necessary.

2. Choose default group semantics that are intuitive and handle the common case well.

3. Allow users to specify custom behavior when the default group semantics do not meet their needs.

4. Aggregate results whenever possible to improve scalability and performance, yet provide methods for users to obtain individual results.

Conceptually, group file operations can be considered equivalent to applying the file operation individually to each group member. The complexity in defining group file operation semantics concerns the treatment of operands and return codes. We categorize operands whose values are used only as inputs to an operation as input operands, and those whose values are modified as output operands.

Input operands are the simplest to map to group behavior, as intuition suggests that the same values should be used when operating on each group member. For example, a `write` operation has only input operands: the descriptor, a byte count, and a data buffer. A `write` on a gfd will copy the requested bytes from the provided data buffer to each group member at its current offset.

In contrast, the values of output operands and return codes can differ across group members. Henceforth, we refer to output operands as *data results* and return codes as *status results*. Given our goals of keeping existing interfaces and providing individual results on request, the collections of data and status results produced by each group file operation present two problems. First, existing file operations have interfaces designed to return individual results. Second, the collections have sizes that grow linearly in the size of the group. For very large groups, the collections may have huge memory footprints, and the processing required to analyze results can become a bottleneck that limits the throughput of group file operations.

Our solution to these problems is to aggregate each collection into a group result; the aggregates

are referred to as *group status result*s and *group data results*. Aggregation is the process of constructing a whole from several individual parts. Data aggregation allows for the construction of a single representation of data from multiple sources. The resulting representation may provide complete or reduced information. Representations providing complete information allow for identifying the result from every group member, while those providing reduced information do not. Examples of complete representations include arrays of individual values and value-equivalence classes that record the group members for each unique value. Reduced representations typically summarize, categorize, or filter individual results; examples include value summaries (e.g., sum or average), bins containing member counts for unique values or value ranges, and top-N or bottom-N value sets.

By choosing aggregations with appropriate data representations, group results can fit existing interfaces. Group status results must take the form of a single summary value that can be returned by the group file operation. For group data results, we observe that existing interfaces use pointers for operands that may be modified. One straightforward method to deliver group data results is to require users to pass a pointer to a buffer that can hold an array of individual results.

Furthermore, many forms of data aggregation are suitable for distributed computation, including all aggregations that use the example data representations mentioned above. By computing aggregate results in a distributed fashion, such as within a TBON, the computation required by the tool to analyze data from the group can be greatly reduced. Aggregations that use reduced data representations also decrease the memory footprint for group results.

A group status aggregation is passed a list of member status results, and computes a single summary value that can be returned by the group file operation. A group data aggregation is passed a list of data buffers as pairs of the form (`buffer`, `length`), and must produce a single output data buffer and length. To support aggregation of data that spans multiple group read operations, data aggregations may allocate state that is passed to subsequent executions of the aggregation for the same group file.

Name	Type	Aggregation Description
`status_min`	Status	Returns the minimum status value, or `-1` if any individual errors occured.
`status_max`	Status	Returns the maximum status value, or `-1` if any individual errors occured.
`status_sum`	Status	Returns the sum of all status values, or `-1` if any individual errors occured.
`status_equal`	Status	When all status values are equal, returns the equivalent value. Otherwise, returns zero for varied values, or `-1` if any individual errors occured.
`data_concatenate`	Data	Treats data buffer as an array of buffers. Places data from each group member at an offset in the buffer that corresponds to that member's index in the group (as returned by `gindex`). For per-member data of size `D` and `G` group members, the total buffer size is $G \times D$, and the offset for a member with index $k \in [0, G-1]$ is $k \times D$.

Table 3.2 Common Aggregations for Group Status and Data Results

Aggregation	File Operations
`status_min`	`close, fchmod, fchown, fstat, ftruncate, lio_listio, aio_read, aio_write, aio_suspend` These operations normally return zero upon success and `-1` for an error. The `status_min` aggregation produces the same behavior for group status results.
`status_sum`	`pread, pwrite, read, readv, write, writev` These operations normally return the total count of bytes read or written upon success and `-1` for an error. Using `status_sum` produces a total count across all group members on success, or `-1` if any individual errors occur.
`status_equal`	`lseek` When all member seeks land at the same file offset, `status_equal` will return the common offset. Otherwise, it will return zero, or `-1` if any individual errors occur.

Table 3.3 Default Status Aggregations for Group File Operations

`int gindex(int gfd, const char* file)` Returns the index within group results of the named `file` for the group specified by `gfd`, or `-1` on an error. The index range for a group of size `G` is `[0,G-1]`.
`int gstatus(int gfd, long int* status_array)` Fills user-allocated `status_array` with the individual member status results of the last group file operation on the specified `gfd`. Status values corresponding to successful completion are positive, while errors are indicated by negative values (e.g., `-EACCES`). Returns the number of individual errors.
`int gloadaggr(const char* library, const char* function)` Loads the named aggregation `function` located in the shared object file `library`. Returns a unique identifier for the aggregation that can be used with `gbindaggr`, or `-1` on an error.
`int gbindaggr(int gfd, FileOp fop, AggrType typ, int ag, const char* params_fmt, ...)` Binds aggregation `ag` to a file operation `fop` for the group `gfd`. If `gfd` equals `-1`, the binding is a default for future groups. `FileOp` is an enumeration type indicating the specific file operation, such as `OpRead` or `OpWrite`. `AggrType` is an enumeration type indicating `ag` corresponds to a status or data aggregation. `params_fmt` is a format string that describes the subsequent varargs parameters; the parameters are passed to the aggregation function each time it is run. Returns zero on success, `-1` on an error.

Table 3.4 New operations for status and data aggregation: Interface and Description

For convenience, we pre-define a small set of common aggregations for processing group status and data results. These aggregations are shown in Table 3.2. The initial set includes four status aggregations and one data aggregation. In Table 3.3, we have chosen default group status aggregations for each type of group file operation that are reasonable for common use. When a user wishes to view more detailed status results from members, a new `gstatus` operation (see Table 3.4) can be used to retrieve all individual results from the last group file operation issued on a gfd.

We expect that in common use, `gstatus` will be used only when the group status value indicates unexpected behavior or an error. For example, an anomalous status for a group `read` using the `status_sum` aggregation would be less than the expected value `byte-count` × `gsize(gfd)`. The user could then query individual results to see which members did not read the requested amount. In the event that one or more individual operations return errors, the group status value will indicate a

group error state, regardless of the summary aggregation currently in use. This error status indicates to the user that `gstatus` should be used to identify faulty members.

For group file operations that produce group data results, we believe the logical choice for default aggregation is `data_concatenate`, which combines individual results into an array. Individual results are then accessed in the group data result using a member file's index as returned by the new `gindex` operation described in Table 3.4. Although array concatenation is by no means the most scalable of aggregations, it is intuitive and functional for arbitrary data (e.g., binary data structures and text). Users that know the format of output data a priori are encouraged to use custom aggregation for improved performance and scalability.

We have defined two new operations in support of specifying custom aggregations, `gloadaggr` and `gbindaggr`, with interfaces as described in Table 3.4. The former is used to load new aggregation functions from a shared library for use with group file operations, while the latter binds aggregations to a specific group file operation. Aggregations can be bound to group file operations for a particular group file or as a default for future groups. `gbindaggr` supports optional aggregation parameters that are passed to the aggregation each time it runs. Users can alter the behavior of an aggregation for an existing group file by calling `gbindaggr` with new parameter values that are used by the aggregation to determine the desired behavior.

Figure 3.5 lists the steps required for each group file operation. The implementation of these steps can take advantage of parallel techniques, as we show in Chapter 5. First, the gfd is used to retrieve group file information (line 3). The file operation is called for each group member using the supplied input operands, and individual status values are stored (lines 7-11). The status aggregation function is used to compute the group status result (lines 13-14). When necessary, group data results are computed using the data aggregation function (lines 16-20). Input operands and the array of individual status results are passed to the data aggregation so it can properly handle partial data results and errors. The

```
 1  int fileop(gfd, in_arg, out_arg)
 2  {
 3      group_file* gf = get_group(gfd);
 4      int status_arr[gf->size];
 5
 6      // Call fileop for each member
 7      for( i=0; i < gf->size; i++ ) {
 8          int fd = gf->members[i];
 9          o_arg = out_arg + i;
10          status_arr[i] = fileop(fd, in_arg, o_arg);
11      }
12      // Compute group status result
13      status_aggr_fn = gf->saggr(fileop);
14      int grp_status = status_aggr_fn(status_arr);
15
16      data_aggr_fn = gf->daggr(fileop);
17      if( data_aggr_fn != NULL ) {
18          // Compute group data results
19          data_aggr_fn(status_arr, in_arg, out_arg);
20      }
21      return grp_status;
22  }
```

Figure 3.5 Group file operation algorithm
Algorithm for applying a file operation to a group file and computing group status and data results.

data aggregation function stores the group data result in the output operand. Finally, the group status result is returned (line 21), and group data results are available in the output operands.

3.1.3 Example Uses of Group File Operations

We provide a small tutorial on the use of group file operations to explore its features. This tutorial covers group file definition, group reads and writes, use of custom status and data aggregations, and group error handling. The context for the tutorial is building simple tools for managing a cluster of Linux hosts. For exposition purposes, we assume the existence of a global file name space that provides access to the local name space of each host; the global name space is partitioned and places each host's name space in a separate directory tree. All of tutorial code examples use the C language, and have been simplified to elide details such as error handling.

The first step in using group file operations is to define the group. Figure 3.6 shows code for constructing a group directory (lines 5-16) and establishment of a group file descriptor using `gopen` (lines

```
 1  // an array of cluster host names (populated elsewhere)
 2  extern char* hosts[num_hosts];
 3
 4  // create group directory
 5  char* hostfile = "proc/loadavg";
 6  char* grpdir = "/tmp/my_grp";
 7  int rc = mkdir(grpdir);
 8
 9  // define group membership using symbolic links
10  char member_path[PATH_MAX];
11  char member_link[PATH_MAX];
12  for( int i=0; i < num_hosts; i++ ) {
13      sprintf( member_path, "/hosts/%s/%s", hosts[i], hostfile);
14      sprintf( member_link, "%s/%d", grpdir, i );
15      symlink( member_path, member_link );
16  }
17  // open group and retrieve size
18  int gfd = gopen( grpdir, O_RDONLY | O_BESTEFFORT, 0 );
19  if( gfd == -1 ) report_error();
20  int gsz = gsize(gfd);
```

Figure 3.6 Example: Group definition
Common steps for defining a group of existing files using symbolic links and obtaining a group file descriptor.

18-20). For brevity, the rest of our tutorial examples assume a group directory has been created in this manner, after making the necessary substitution for the target `hostfile`.

Our first tool example demonstrates the use of group read operations and custom data aggregation to monitor load at cluster hosts. The `hostfile` for this example is `/proc/loadavg`, which contains a single text line showing the one-minute, five-minute, and fifteen-minute average loads. Figure 3.7 provides code for the data aggregation function and the monitor tool. The interface to aggregation functions is specified by the file system supporting group file operations. In the tutorial aggregation examples, we use a simple interface that receives the data read from all group members as inputs, and stores the group data result in the data buffer passed to `read`. We define the aggregation function interface employed by TBON-FS in Chapter 5.

Our second tool example, shown in Figure 3.8, also uses group read operations and custom data aggregation to search for lines in system logs that contain a specific phrase. The search phrase is specified using the aggregation parameters provided to `gbindaggr`. The `hostfile` for this example is

```
 1  // load custom tool data aggregation
 2  int aggr_id = gloadaggr( "tool.so", "calc_load_avgs" );
 3
 4  // monitor loop
 5  size_t count = LOADAVG_MAX_FILE_SIZE;
 6  bool failed = false;
 7  double 1min_avg, 5min_avg, 15min_avg;
 8  do {
 9      // open group and bind data aggregation to read ops
10      int gfd = gopen( grpdir, O_RDONLY, 0 );
11      gbindaggr( gfd, OpRead, AggrData, aggr_id, NULL );
12      // read whole files using custom aggregation
13      char buf[ 3 * sizeof(double) ];
14      ssize_t read_sum = read( gfd, buf, count );
15      if( read_sum < 0 ) failed = true;
16      else {
17          // extract and record aggregated load averages
18          get_load_averages(buf, &1min_avg, &5min_avg, &15min_avg);
19          record_load_averages(1min_avg, 5min_avg, 15min_avg);
20      }
21      close(gfd);
22      sleep(300); // five minutes
23  } while( ! failed );
```

(a) Monitor Tool

```
 1  struct file_data {
 2      void* data;           // data (any format)
 3      unsigned file_index;  // file index within group
 4  };
 5  void calc_load_avgs( char* params_fmt, va_list params,
 6                       int num_inputs, struct file_data inputs[],
 7                       char* output_buf )
 8  {
 9      // calculate average of inputs
10      double avg1 = 0, avg5 = 0, avg15 = 0;
11      for( int i=0; i < num_inputs; i++ ) {
12          double one, five, fiftn;
13          scan_loadavg_data(inputs[i].data, &one, &five, &fiftn);
14          avg1 += one;
15          avg5 += five;
16          avg15 += fiftn;
17      }
18      avg1 /= num_inputs;
19      avg5 /= num_inputs;
20      avg15 /= num_inputs;
21      store_averages(output_buf, avg1, avg5, avg15);
22  }
```

(b) Custom Data Aggregation Function

Figure 3.7 Example: Load Monitor

A tool for monitoring load at cluster hosts using group read operations with custom data aggregation. Code for the tool is shown in **(a)**, and a sample aggregation function is shown in **(b)**.

```
1  // load custom tool data aggregation
2  int aggr_id = gloadaggr( "tool.so", "match_lines" );
3  char* keyphrase = "authentication failure";
4
5  // open group and bind data and status aggregations to read ops
6  int gfd = gopen( grpdir, O_RDONLY, 0 );
7  gbindaggr( gfd, OpRead, AggrData, aggr_id, "%s", keyphrase);
8  gbindaggr( gfd, OpRead, AggrStatus, status_max, NULL );
9
10 // search loop
11 size_t count = 4096; // a page at a time
12 char buf[ count * gsize(gfd) ]; // conservative allocation
13 bool done = false;
14 do {
15     // read chunks using custom aggregation
16     ssize_t read_max = read( gfd, buf, count );
17     if( read_max < 0 ) { // group error
18         report_error();
19         done = true;
20     } else if( read_max > 0 ) { // not all logs have reached end-of-file
21         print_matches( buf );
22     } else done = true;
23 } while( ! done );
24 close(gfd);
```

(a) Log Search Tool

```
1  struct file_data {
2      void* data;           // data (any format)
3      unsigned file_index;  // file index within group
4  }
5  void match_lines( char* params_fmt, va_list params,
6                    int num_inputs, struct file_data inputs[],
7                    char* output_buf, void** aggregation_state)
8  {
9      // get phrase to match
10     if( strcmp(params_fmt, "%s") != 0 ) return -1;
11     char* phrase = va_arg(params, char*);
12     int max_lines = 100;
13     char* lines[max_lines];
14     for( int i=0; i < num_inputs; i++ ) {
15         // search file data for lines that match
16         char* partial_line = get_partial_line(i, *aggregation_state);
17         int count = search_log_data(phrase, inputs[i].data,
18                                     lines, max_lines, &partial_line);
19         // record partial line in aggregation state
20         if( NULL != partial_line )
21             store_partial_line(i, partial_line, *aggregation_state);
22         // append lines to output annotated with source index
23         store_lines(output_buf, inputs[i].file_index, lines, count);
24     }
25 }
26
```

(b) Custom Data Aggregation Function

Figure 3.8 Example: Log Search

A tool for finding lines in system logs that contain a key phrase. Code for the tool is shown in **(a)**, and a sample aggregation function is shown in **(b)**.

```
 1  // open group
 2  int gfd = gopen( grpdir, O_WRONLY | O_TRUNC, 0 );
 3  int gsz = gsize(gfd);
 4
 5  // map input file in memory and retrieve its size
 6  size_t file_len = 0;
 7  char* file_buf = map_file("/etc/passwd", &file_len);
 8
 9  // copy file in 4MB chunks
10  size_t count = 4096 * 1024;
11  size_t write_sum = count * gsz;
12  size_t copied = 0;
13  do {
14      // write 4MB chunk using default status aggregation (sum)
15      ssize_t written = write( gfd, file_buf + copied, count );
16      if( written < write_sum ) {
17          long int status_arr[gsz];
18          int num_errors = gstatus(gfd, status_arr);
19          if( written == -1 )
20              report_errors(num_errors, status_arr);
21          else
22              report_partial_writes(count, status_arr);
23          break;
24      } else copied += count;
25  } while( copied < file_len );
26  close(gfd);
```

Figure 3.9 Example: File Replication
Code for replicating a file across cluster hosts.

`/var/log/authlog`, which records actions of a Kerberos authentication server. The tool searches for lines indicating authentication failures. This example also demonstrates the use of an alternative status aggregation (`status_max`) for group reads that is convenient for determining when all member files have reached end-of-file. It also shows the use of aggregation state to handle partial lines that are read across multiple operations.

Our final example in Figure 3.9 shows the use of group write operations in a tool for replicating a configuration file across the cluster. The `hostfile` for this example is `/etc/passwd`. This example also demonstrates the use of `gstatus` to aid in identifying partial or failed writes.

3.1.4 Sharing Considerations for Group Files

In our discussion so far, we have assumed a program using group file operations was *group-aware*

since it had to obtain a gfd. However, in Unix-based environments there are well-established methods for sharing an open file descriptor. We review how a gfd can be shared, indicate the potential problems of *group-unaware* use of a gfd, and discuss approaches for avoiding these problems.

There are three methods for sharing open file descriptors in Unix-based environments. The first method is for a process to call `fork` to create a new process that is a copy of itself, where the copied operating system state includes the descriptors. When the original process is group-aware, we assume the created process is still group-aware and will have no problem when performing future group file operations. The second method is for a process to call `exec` to replace its current address space with the image of an executable file. In this case, the open descriptors are inherited by the new program, but there is no guarantee that the program is group-aware. The final method involves sending a list of open descriptors as ancillary data to a `sendmsg` call on a UNIX domain socket. Again, there is no guarantee that the receiving process is aware of the presence of gfds in the list.

When a group-unaware process uses a gfd, there are three potential outcomes: (1) successful completion with no change in behavior from the program's persepective, (2) successful completion with non-standard behavior, and (3) silent process corruption or failed completion. The first outcome can be expected for operations that do not have output data and where no status aggregation is performed. The second outcome can be anticipated for operations with no output data and aggregated status results (e.g., `write` and `lseek`), since the returned value will not match that expected by a program written for single file operations. This mismatch may lead group-unaware programs to abort immediately due to unexpected behavior, or to fail later on when performing actions based upon the unexpected result value. The second outcome is also possible for operations with aggregated output data, when the aggregated data result does not exceed the size of the output data buffer (e.g., a `read` using a summary data aggregation). The third outcome can be expected for operations with an aggregated data result that is larger than the allocated output buffer (e.g., concatenated results from `read` and `fstat`), as silent

process corruption or an operational failure will occur when copying aggregated results to the buffer.

To avoid the potential problems associated with the second and third outcomes, a process that shares a gfd with another process that is not guaranteed to be group-aware should use precautionary measures. These measures may include changing the aggregations currently associated with the file operations on the gfd before sharing it. Appropriate aggregations would provide status and data results consistent with their single file operation counterparts.

3.2 Extensions to the Idiom

Based on our experience with using group file operations in the tool and middleware case studies presented in Chapters 7, 8, and 9, we identified extensions that provide further convenience and expand the idiom's utility. In this section, we describe extensions for improved error handling, group duplication and subsetting, and group program execution. We briefly review each of these extensions and their intended uses.

3.2.1 Group Error Handling

As the size of a distributed system increases, the expected time between failures of system components decreases. Given our target systems contain tens or hundreds of thousands of hosts, it is important to provide robust and efficient error handling semantics.

We expect users of group file operations will use familiar error handling strategies that work well for normal file operations. For example, application-level retry is often sufficient to overcome transient file system or network failures during operations on distributed files. When faced with extended network outages or permanent host component failures, most programs will try to complete the operation a few times before eventually giving up.

Group file operations add to the complexity of error handling by introducing situations in which only a subset of member operations fail. As in normal file operations, we expect programs will attempt

to recover from group file operation failures by identifying the failed subset and having that subset retry the operation. Because failures from different hosts may be unrelated, there is no guarantee that the entire failed subset will recover from transient problems at the same time, nor that all failed members will ever recover in the presence of permanent failures. When failures persist, the program may decide to continue operating on only the non-failed members. Continued operation is possible even when the program is using group writes, as servers are completely independent and have no consistency requirements. For example, in file replication scenarios, the client program has access to the pristine copy and can distribute its contents to the failed servers later on after they have recovered.

Determining group members that have experienced errors is a prerequisite to both retry and continued operation. As discussed previously, the new `gstatus` operation provides this information in the form of an array of group member status values. Unfortunately, scanning the status array for errors is a sequential operation that in the worst case requires inspecting every index. A more efficient and convenient option is to provide a new `gerrors` operation that identifies failed members.

We consider two options for the functionality provided by `gerrors`. The first option is for `gerrors` to supplement `gstatus` by providing a list of the member indices that contain errors. The drawback of this option is it requires the prior use of `gstatus` to fetch the entire group status array, which can be quite large, even when the program is only interested in errors. The second option is for `gerrors` to serve as an alternate to `gstatus` by providing a list of failed member indices and their associated error values. When the ratio of failed members to group size is small (i.e., a few percent), this second option is preferable. However, one common error situation for group file operations involves the failure of all member operations, such as the situation that results when trying a group `write` on files that have only read-access. In such situations, the second option would require twice the space of the `gstatus` array. We currently favor the first option for its simplicity and memory savings for full group error scenarios. The current interface for `gerrors` follows:

```
int gerrors( int gfd, int* index_array )
```

The `index_array` operand is a user-allocated array whose length should be equal to the number of errors, as returned by `gstatus`. This array is populated with the group indices for failed members.

We plan to investigate a third approach to identifying errors that is based upon an alternative version of `gstatus` that returns an opaque handle to a histogram object. The histogram associates each unique status or error value with a list of members that returned that value. We envision providing several functions for querying the histogram to obtain information such as the set of unique values, the member counts for each unique value, and the membership lists for each value. A histogram-based approach to aggregation of status and error values should provide greater data scalability than the current integer arrays used to hold values and member indices. A key to achieving our desired level of scalability for next generation systems is to devise a compact representation for membership lists, as groups will likely contain millions of files.

Given the ability to easily and efficiently identify failed members, the next requirement for retry or continued operation is to generate group files representing the failed or non-failed subsets. The following subsection discusses our proposed extensions for group subsetting.

3.2.2 Group Duplication and Subsetting

Duplicating a group file by using `dup` on its descriptor is likely more efficient than using `gopen` on the original group directory. Following the POSIX semantics for `dup`, the member files represented by the new gfd are the same as those from the original gfd, and thus share file offsets and status flags. The question for group files is whether to share the current status and data aggregations as well.

We believe duplication could be convenient for interleaving different forms of data aggregation while reading group files. For instance, assume member files contain structured data at known offsets, and a tool wishes to use separate aggregations for each type of structure. As shown in Figure 3.10, there are two file organizations to consider: (a) shows a file where all the structures of a given type are

located within a single range of the file's data, while (b) shows interleaved structures of different types. We assume files are being read sequentially and completely. With the ranged organization of (a), a program only needs to use `gbindaggr` three times, before reading the data in each range. However, for the interleaved organization of (b), a program would be forced to use `gbindaggr` much more often to change the aggregation binding each time the type of data changes, which could have high overhead. A more efficient alternative would be to create duplicate gfds, where each gfd is initially bound to one of the necessary aggregations. Then, the program can simply switch the gfd passed to each `read`.

As introduced in Section 3.2.1, a useful extension to the notion of group file duplication is *group subsetting*, which we define as the creation of new group files that represent subsets of the members of an existing group file. The primary motivation for supporting group subsetting is to handle cases where member subsets diverge from the expected behavior, such as when some members return errors or status values indicating only a portion of the requested data was read or written. In these cases, the program may want to operate on the subset to return it to the expected state.

Using the semantics we previously defined in Section 3.1.1 for managing group files, creating a group file subset requires two actions. First, a directory containing the subset's files must be created by either modifying the original group directory entries or populating a new group directory. Second, `gopen` is called using the subset group directory. Rather than forcing programs to employ this costly sequence of actions, we added a new `gsubset` operation that allows for duplication of a subset specified using member indices from the original group. This operation returns a new gfd that represents the target subset and provides the same semantics as descriptors obtained using `dup` on a group file. The proposed interface is shown below:

```
int gsubset( int gfd, unsigned num_members, int members[] )
```

As with whole-group duplication, programs are free to use different status and data aggregation with the new gfd.

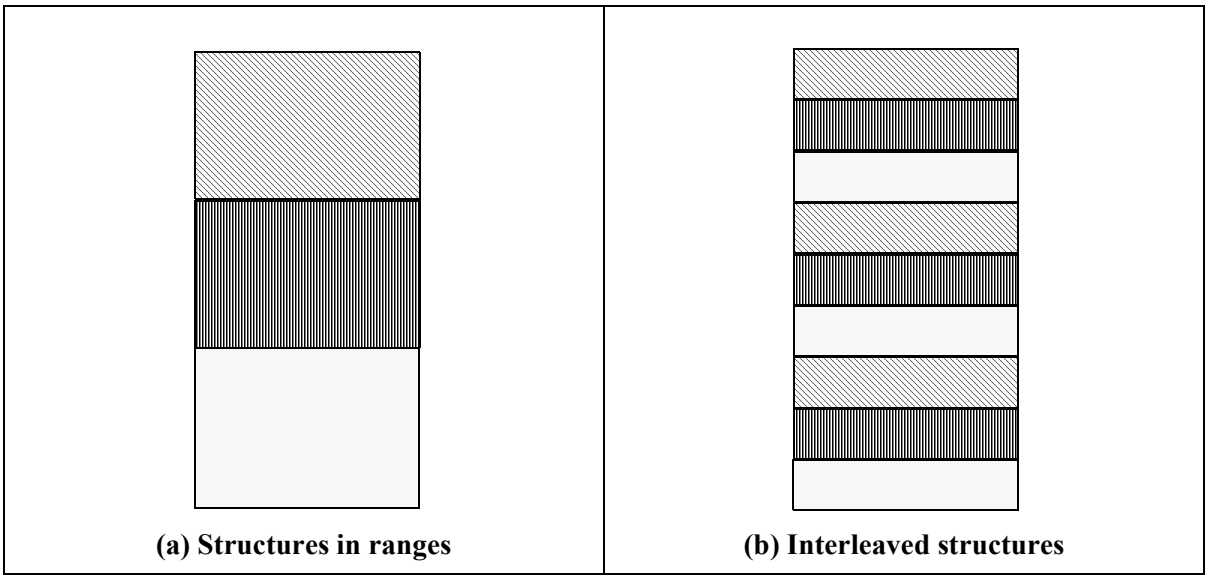

Figure 3.10 Two file organizations containing three types of structured data
(a) All structures of a given type reside within a single range.
(b) Structures of different types are interleaved.

3.2.3 Group Program Execution

A common task for distributed system administration is running the same program on a set of hosts and collecting the programs' output. Many tools [18,29,36,53,90,119] have been created that support this task, but few are designed for practical use on systems containing thousands of hosts. The primary deficiency of previous tools is how collected program output is displayed to users. All prior tools provide attribution of output lines to specific hosts. To avoid user confusion caused by interleaved output from many hosts, most tools provide an option to display all the output from a particular host in one grouping. Such ordering by hosts at least makes the output comprehensible for large numbers of hosts, but does not address the limited abilities of users for digesting thousands of lines of output. When programs on many hosts produce equivalent output, a few tools [29,53,90] help users by displaying only unique sets of program output prepended by the hosts that produced each set.

To provide more flexible and scalable handling of output from distributed programs, we investigated the possiblity of using the custom data aggregation capabilities of group file operations. The primary obstacle was developing a method of capturing the output of distributed processes as a gfd. Our

approach combines the functionality of two common POSIX operations, `execve` and `popen`, within a new `gexec` operation that provides group launch of distributed programs. The interface of `gexec` is modeled after `execve`, and shown below:

```
int gexec( const char* grpdir, char* argv[], char* envp[] )
```

Similar to `gopen`, the first operand to `gexec` is the path of a group directory. This directory is expected to contain links to executable programs that reside on distributed hosts. `gexec` launches the programs in the directory on their respective hosts using `execve`, passing the supplied argument and environment vectors. It returns a gfd that can be written to provide `stdin` to the distributed processes and can be read to obtain data they send to `stdout` and `stderr`. This I/O capture is inspired by `popen`, which returns a file stream for writing `stdin` or reading `stdout`, but not both. Using the gfd, tools can bind custom `read` aggregations that annotate, filter, or transform the programs' output.

3.3 Summary

Group file operations are an intuitive idiom for applying operations to groups of distributed files. The idiom addresses the two primary scalability barriers in such group operations: forced iteration and data that grows with the size of the group. Iteration is eliminated by defining a group file abstraction that can be used with familiar POSIX I/O operations. The idiom provides semantics for explicit aggregation of group status and data results to avoid or reduce the data storage and analysis costs incurred by tools and middleware using group file operations. Together, these features allow for scalable implementations of group file operations, as we show in Chapter 5.

One deficiency of the idiom is that it does not address scalable definition of file groups. As shown previously in Figure 3.6, populating group directories requires iteration, which we strive to avoid. The following chapter presents our ideas for composing custom global name spaces that automatically group files into directories for use with group file operations.

By extending familiar POSIX I/O operations to produce the group file operation idiom, we believe

the learning curve for adoption within new or existing software is significantly reduced. Still, we have identified cases where strict adherence to the existing interfaces and our attempts to mirror the utilitarian style of POSIX in the new operation interfaces results in less than ideal interactions. For example, when using custom data aggregation during a group read, it is often hard to estimate the size of the buffer that is necessary to hold the aggregated result. Unfortunately, the semantics for POSIX `read` dictate that the destination buffer is pre-allocated. It is often necessary to allocate for the largest expected size as a prevention against buffer overruns, which can result in situations where only a small fraction of the buffer space is actually used by the aggregated result. If we relax these semantics to allow the group read to return a buffer rather than copy into an existing buffer, excessive allocations can be avoided. A study of the inefficiencies in the interfaces of group file operations could help to identify and resolve such problems.

Chapter 4

Flexible and Scalable Composition of File Name Spaces

The group file operation idiom uses directories as the file grouping abstraction, but does not address scalable construction of group directories. Using conventional file system operations, populating a group directory with files requires iteration that limits scalability for large groups. In this chapter we discuss a scalable approach to forming file groups that relies on composing the independent file name spaces of thousands of servers into a global name space. We first discuss our goals for providing tools and middleware the ability to compose custom global name spaces. Next we introduce FINAL, our language for flexible and scalable composition of file name spaces. We then demonstrate FINAL's expressive power by using it in several interestingly diverse name space compositions.

4.1 File Name Space Composition Goals

Establishing a group file requires populating a directory with links to member files before using `gopen`. Linking to distributed files requires that they are accesible using local names. We assume this requirement is met using a distributed file system that provides a global name space. However, using file system operations such as `stat` and `symlink`, populating the group directory requires iteration that can take hundreds of seconds for groups containing tens of thousands of distributed files. To avoid

such centralized, iterative group definition, we began investigating approaches that were amenable to scalable implementations using TBONs.

The first approach we considered was parallel path matching using regular expressions. In this approach, a file group's membership is defined by providing a set of regular expressions representing paths, and the TBON is used to distribute these expressions to each distributed host. Hosts independently evaluate the expressions to identify matching files in their local name spaces, and all of the host matches are aggregated using the TBON to form the list of files to be included in the group directory. Strategies based on path matching are sufficient to group files with identical or similar paths on distributed hosts, which is a common case for the file groups targeted in group file operations. Unfortunately, the use of names as the sole matching criteria limits the ability of tools to select relevant files.

One limit of using name-based matching is that file groups are often formed based on non-name file attributes such as size, owner, or access times. For example, a tool for processing distributed system logs may select only those files whose size exceeds some threshold. A security auditing tool may inspect groups of files whose modification time is within some time period, such as password database files that were recently updated.

Another common selection criteria uses host-specific context. For example, a tool for replicating library or executable files across distributed hosts with heterogeneous processor architectures may create architecture-specific file groups for each library or executable. Unless the hosts place files in architecture-specific locations, paths cannot be used to create these file groups. Thus, a mechanism is required for querying the local context at each host to determine the architecture, such as retrieving the value of the `$MACHTYPE` environment variable or executing `uname` on Unix systems. Host context is also important for paths that include process identifiers (pids), as would be used by tools such as parallel debuggers that target a specific set of distributed processes. Since pids are not unique across distributed hosts, expressions for paths that include specific pids can result in unintended matches on

multiple hosts. In such use cases, alternative methods that allow tools to select files owned by a particular user or corresponding to processes executing a specific application program would be beneficial.

Our goal is to support the diverse strategies used by tools to select relevant files for grouping. To this end, we designed an approach to group definition based on file name space composition that provides the necessary flexibility. Our approach accomodates both path-based matching as well as more complex selections based on attributes or host context. The result is a flexible specification language for composing custom file name spaces and a scalable system for composing a global name space by merging thousands of independent name spaces using a TBON. The language provides programs with composition semantics that enable efficient and natural formation of file groups, as a means for generating a name space tailored for use with group file operations.

4.2 A Language for File Name Space Composition

To meet our global name space composition goals, we developed a specification language for describing compositions with three key qualities:

- *Scalability* - many name spaces can be combined using efficient distributed name space construction, avoiding centralized pair-wise operations.
- *Simplicity* - name space composition is easily described using a simple tree abstraction for name spaces and a set of tree composition operators with clear semantics.
- *Flexibility* - many interesting compositions can be specified by combining declarative tree operations with prescriptive programming constructs.

The language provides a semantic foundation that guides our approach for scalable global name space composition, and can be adopted by systems requiring flexible composition. Our language is named FINAL, the **Fi**le **N**ame space **A**ggregation **L**anguage.

FINAL draws heavy inspiration from Zandy's *flac* language [125] that was designed for describing file name spaces for mobile applications. Flac's treats name space composition as operations on trees,

which results in declarative specifications that are simple to understand. However, there are limitations to flac's scalability when applied to global name space composition and to its flexibility in supporting general-purpose name space composition. FINAL addresses these deficiencies by extending flac's semantics to support efficient composition of many trees, and by introducing prescriptive language constructs that provide additional flexibility in describing a wider variety of compositions.

In Section 4.2.1, we review the key design features of flac and discuss its limitations. Section 4.2.2 introduces the two main abstractions used within FINAL, name spaces and file services. Section 4.2.3 describes FINAL's name space composition operations that are based on these abstractions. Section 4.2.4 discusses the benefits for flexibility in specifying interesting compositions that can be achieved by extending FINAL with prescriptive capabilities. Section 4.3 provides several examples that demonstrate FINAL's flexibility and its application to scalable global name space composition.

4.2.1 Flac: A Good Start

Flac, whose name is short for *file access*, is a language for describing the file name space used by a mobile application. Flac treats file name spaces as a traditional tree of names, where internal vertices are directories and files are leaves. A path is a sequence of names that represents a traversal of the tree starting at its root and ending at the vertex corresponding to the named file or directory. Figure 4.1 presents an example name space.

Applications write specifications using flac to compose a custom file name space consisting of files from many file services. A file service in flac is simply a name space tree. A special type of file service, the primitive file service, represents the name space of a local or remote file system and provides operations upon its files and directories. Flac pre-defines a few primitive services such as the `local` service that accesses the local file system and the `smb` service that remotely accesses a Windows network file server.

Composition of file services is expressed using operators called *combinators*. Combinators operate

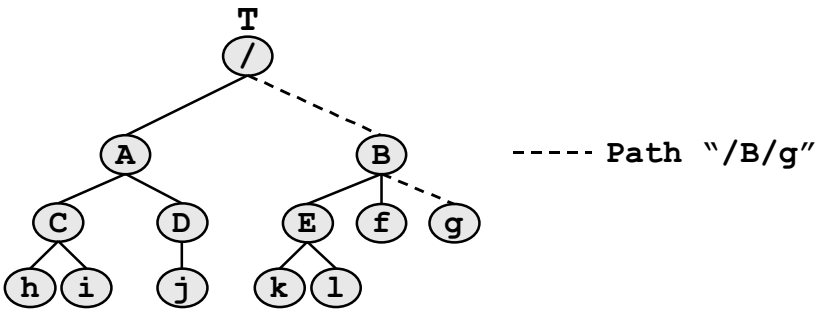

Figure 4.1 Name Spaces and Paths.
A *name space* is a rooted tree of names. A *path* is a tree traversal starting at the root node. In all name space figures, we use uppercase names for directories and lowercase names for files. For the tree **T**, the path `/B/g` yields file `g`.

on paths and services to produce new services. As shown in Figure 4.2, combinators are provided to get or remove the sub-tree of a service at a given path, to prepend a path to a service, and to overlay a service on top of another service to yield a tree containing names from both. The `overlay` combinator gives precedence to the top (first) service in the case of name conflicts. In every flac specification, the keyword `root` immediately precedes the service definition that describes the fully-composed custom name space. Figure 4.3 shows two specifications that describe file system mounts and union mounts.

Because mobile applications roam across hosts or networks, the file service used to access specific files may need to change when the application moves. For example, consider an application executing on a laptop computer that accesses a file server within a private home network. When the laptop is connected to the home network, files can be accessed in the local file name space via a file server mount. If the laptop is moved to a new location, such as the laptop user's favorite coffee shop, then the home network is no longer accessible and the laptop will likely connect to a new network. Given its new network context, the method for accessing files on the home network file server must change to a form of remote access, such as flac's `rpc` service to a public gateway host in the home network.

Flac seamlessly supports access method transitions based on changes in the application's host or

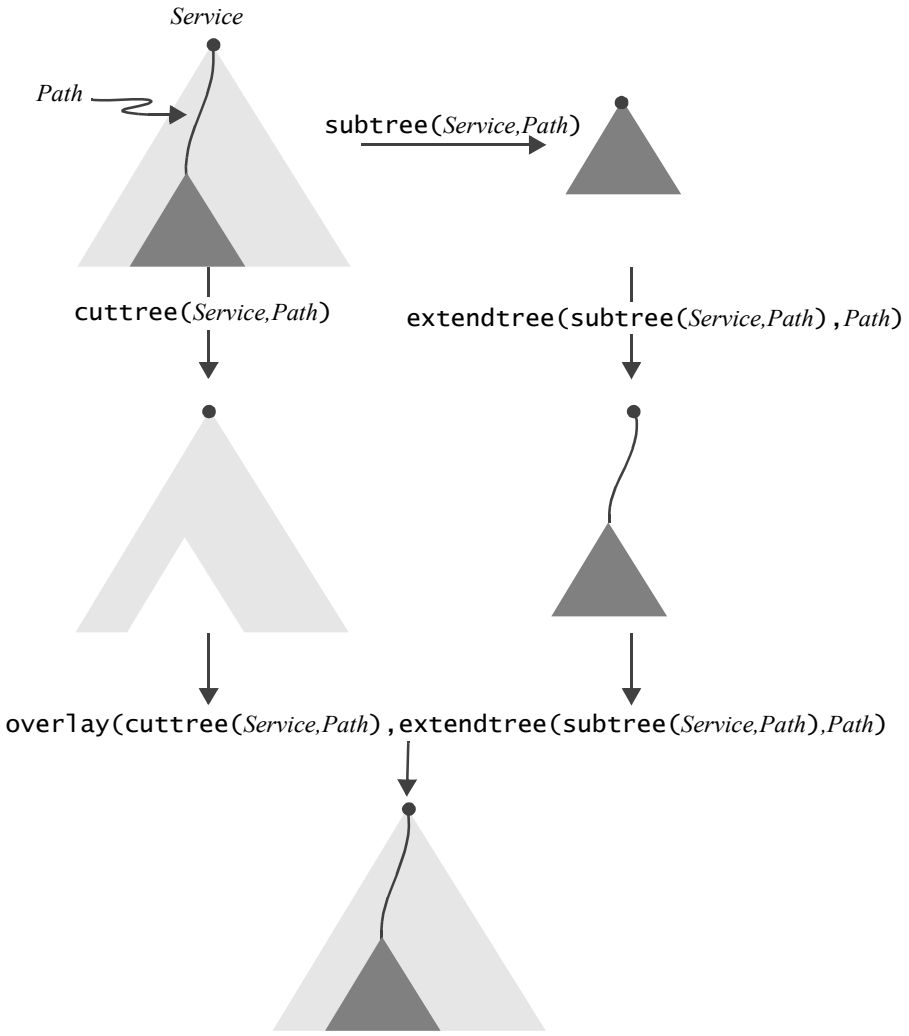

Figure 4.2 Flac Name Space Combinators.
This figure has been reproduced from Victor Zandy's dissertation [125] with his permission.

```
L = local()
N = new_file_svc(args)
root overlay( extendtree(N,path),
              cuttree(L,path) )
```

(a) Traditional Mount

```
L = local()
N = new_file_svc(args)
root overlay( extendtree(N,path),
              L )
```

(b) Union Mount

Figure 4.3 Flac Example Specifications for Mount Semantics.
(a) A traditional mount can be described as removing the sub-tree at the mount point in the local name space and overlaying the new file service name space with the mount point prepended.
(b) A union mount can be described as a traditional mount where the existing sub-tree at the mount point is not removed before overlay the new file service name space.

network contexts. This support relies on a special `select` combinator that applications can use to choose the appropriate file service given the current network location. Network locations can be given as DNS host names or IP addresses, and may contain wildcards. The flac runtime detects changes in location, and chooses the service used to access files based on the selection in the specification for the current location. Figure 4.4 shows a simple specification that uses `select` to support the laptop movement example described above. The `select` combinator also provides two other forms of selection context. One form allows for selection based on matching the value of an environment variable, and another selects according to the specific file operation being used (e.g., `read` or `write`).

In terms of flexibility for general-purpose name space composition, flac is limited in three ways. First, flac has no notion of name space inspection to help in selecting relevant files or directories. For example, there is no way to select or exclude entries in a specific directory that match a name pattern. Second, file attributes such as the modification time or file size can not be queried to aid in selection. Third, `select` is the only method for choosing among alternatives based on context. Only one of the supported contexts, environment variables, is general enough for use in compositions outside of the mobility domain. Because the semantics of `select` limit its use to evaluating simple equality conditions, the possible equality values must be known in advance, and compound conditions that compare multiple variables are not supported directly but can be emulated with nesting.

When composing global name spaces from thousands of independent name spaces, flac has semantic limitations that prevent efficient specifications. If each independent name space is represented as a unique file service, then thousands of services must be defined. The `overlay` combinator is the only way to combine these services in a single name space. Since `overlay` is a pair-wise composition, using it to compose global name spaces requires thousands of serialized operations. Further, the name conflict resolution behavior of `overlay` can not be customized. Thus, there is no way to overlay name spaces in a manner that retains more than one instance of a file at a common path. This

```
LOCAL = subtree( local(), /mount/myfilesvr )
REMOTE = rpc( myhost.isp.net, /mount/myfilesvr )
root select( location, 192.168.100.0/8:LOCAL, else:REMOTE )
```

Figure 4.4 Flac Example Specification using Location-based Service Selection.
When the application is running on a host connected to the private network with subnet 192.168.100.0/8, files can be accessed on the network file server mounted at `/mount/myfilesvr` in the local name space. Otherwise, when connected to any other network, the remote file protocol service (`rpc`) is used to access a similar mount point on a gateway host for the private network.

limits its use in composing group directories containing files with the same path on many hosts.

In response to the flexibility and scalability limitations of flac, we collaborated with Zandy to design FINAL. FINAL adopts flac's name space composition based on tree operations, and introduces a new merging operation that supports customizable composition of many trees. This new operation provides the semantic foundation for scalable global name space composition. FINAL replaces flac's limited name space selection based on domain-specific context with prescriptive language capabilities that provide the flexibility to specify diverse compositions requiring inspection of name spaces, file attributes, and environmental context.

4.2.2 FINAL Abstractions

As in flac, FINAL abstracts file name spaces as trees and name space composition as tree alterations and combinations. We consider these trees to be *name space views* to distinguish them from the physical name spaces of file services. All views used as inputs to composition operations are immutable, and operations produce new immutable views as results. The immutability of views provides clear semantics for composition, as there are no side-effects to the input views, and views can be freely used as inputs to many operations.

FINAL also adopts a file service abstraction that is similar to the primitive file services of flac. A file service provides access to local or remote file systems and a set of common file system operations (e.g., `open`, `read`, and `write` for files, and `list` for directories). Since file services are used as the

Tree Composition Operator	Description
subtree(Tree, Path) → Tree	Returns a copy of sub-tree at the specified path in the input tree.
prune(Tree, Path) → Tree	Returns a copy of the tree with sub-tree at path removed.
extend(Tree, Path) → Tree	Returns a copy of the tree with the specified path prepended.
graft($Tree_1$, $Tree_2$, Path) → Tree	Returns a copy of $Tree_1$ with $Tree_2$ inserted at the specified path.
merge({$Tree_k$}, conflict_fn) → Tree	Returns a new tree that contains all unique paths and entries returned from applying the conflict function to all shared paths.

Table 4.5 Final Composition Operations

building blocks for composition, services conventionally export a view that resembles their physical name space. If a file service removes file system entries after providing a view that contained them, operations on those entries in the composite name space will fail.

4.2.3 FINAL Composition Operations

FINAL treats name space composition as operations on immutable trees. The composition algebra consists of the five tree operations presented in Table 4.5. The first three operations, `subtree`, `prune`, and `extend`, are modeled after the flac combinators and support common manipulations of a single tree. These operations have two operands, a tree `T` and a path `P`. `subtree(T,P)` yields a view of the sub-tree whose root is found by traversing `P` in `T`. If the target sub-tree is a single file, `subtree` returns a tree consisting of a root directory whose only child is the file. `prune(T,P)` complements `subtree`, and results in a view of `T` with the sub-tree rooted at `P` removed. `extend(T,P)` yields a copy of the input tree with `P` prepended to its root. Figure 4.6 applies these three path operations to the example name space of Figure 4.1. The fourth operation, `graft`, inserts one tree into another as shown in Figure 4.6. `graft` does not have a corresponding combinator in flac, which relied on its `overlay` combinator to graft trees. Because `overlay` must traverse both input name spaces to identify shared

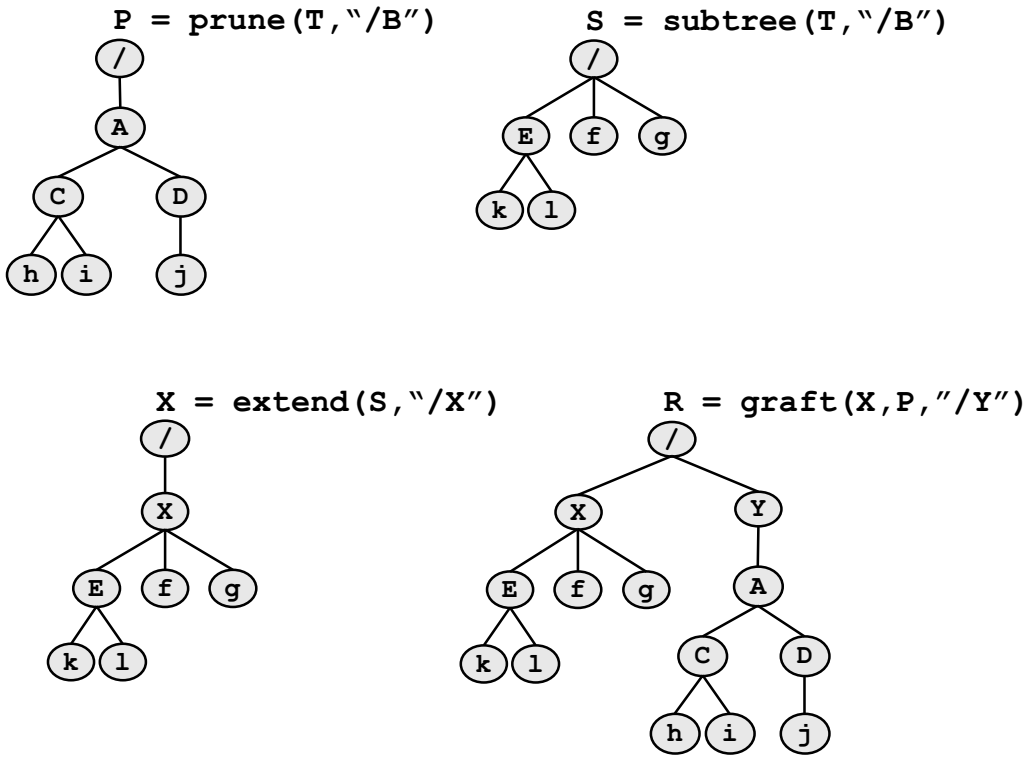

Figure 4.6 Path Composition Operations.
The top two trees show results from applying the **`prune`** and **`subtree`** operations to the name space of Figure 4.1 with the path "**`/B`**". The bottom-left tree shows the result of extending the **S** sub-tree with the path "**`/X`**". The bottom-right tree shows a **`graft`** of tree **`P`** into tree **`X`** at path "**`/Y`**".

and unique paths, it is more expensive to evaluate than `graft`, which specifies the insertion path.

Together, the first four composition operators can describe arbitrarily complex tree compositions. Unfortunately, such flexibility comes with a cost when one wants to perform deep composition of two or more trees, such as is needed for name space overlays. A deep overlay can be described by traversing the trees in level order, identifying shared and unique paths. Sub-trees at unique paths can be grafted into the result tree, while vertices common to both trees result in a vertex copied from the top-layer service. A specification describing these overlay semantics may be extremely verbose for large trees, and the efficiency of composing the trees would be limited to serial evaluation of the composition operations. Similarly, since `graft` operates on two trees, composition of a many trees would

require pair-wise iteration. One of our goals for name space composition is to avoid complex specifications and serialized pair-wise composition. Instead, we favor operations that provide the required semantics while still providing efficient and scalable composition of many trees.

FINAL's last composition operation, `merge`, captures the behavior of a large class of deep compositions while supporting customizable composition semantics. To allow for efficient composition of many trees, `merge` operates on a collection of trees, where the trees are combined at all levels from their roots to leaves. The tree produced by `merge` contains all unique paths that occur in one input tree but not the others, as well as the results of applying a customizable conflict resolution function to paths that are shared among the input trees. Unique and shared paths are determined using a level-order traversal across all input trees. At each level starting with the children of the roots, the conflict resolution function is called for each path that is shared among two or more trees. The function is passed the shared path and a list of conflicting vertices, and produces an output set of *(vertex, path)* pairs. Each *vertex* in the output set is added to the result tree at its paired *path*.

Letting users define custom conflict resolution functions provides flexibility to perform fine-grained manipulation, similar to how directory filters are used within the Virtual System Model [78], while still describing merge compositions at a high-level. Though `merge` captures the behavior of many types of deep composition, it cannot emulate shallow merges, such as the unions provided by Plan 9 in which only first level names below the root are overlayed [94]. However, shallow merges can be specified by merging path-extended sub-trees. Due to the common use of overlay semantics [91,94,123], we provide a pre-defined conflict resolution function called `overlay`. This function outputs the first vertex in the conflicting input list.

Pseudocode for a conflict resolution function that uses a simple file renaming strategy is shown in Figure 4.7; this function returns a single directory for shared directory paths, or a set of renamed files for shared file paths. An example merge operation using this conflict function is shown in Figure 4.8.

```
rename_merge( shared_path, conflicts )
{
    results = [ : ]; // initialize empty hash table
    if( all_directories(conflicts) ) {
        // return single, common directory
        results[shared_path] = conflicts[0];
        return results;
    }
    else if( all_files(conflicts) ) {
        // return all files, renamed to not conflict
        for( int i=0; i < length(conflicts); i++ ) {
            // new path is shared path plus version
            p = shared_path + "." + i;
            results[ p ] = conflicts[i];
        }
        return results;
    }
    // bad input structure
    return nil;
}
```

Figure 4.7 `merge` Conflict Resolution Function.
Renames all files having the same path by appending a version number. Conflicting directories are merged to a single directory.

FINAL's composition operations can be used to specify a wide range of compositions in a simple, declarative manner. For example, we show how to specify four common forms of file system mounts. In these examples, `O` is the original name space, `P` is the path for the mount point, `N` is the new tree to be mounted, and `T` is the resulting tree.

- A traditional mount that replaces the sub-tree of the current name space at the mount point with a new file name space is easily specified using `prune` and `graft`:

  ```
  T = graft( prune(O,P), N, P )
  ```

- A bind mount is a variation of the traditional mount that uses a sub-tree of the existing name space rather than a new file system as the new name space to be mounted. Thus, bind mounts make a portion of the current name space available at more than one path simultaneously. A bind mount where P_1 is the original path to the sub-tree and P_2 is the new path can be specified as:

  $T = graft(O, subtree(O,P_1)), P_2)$

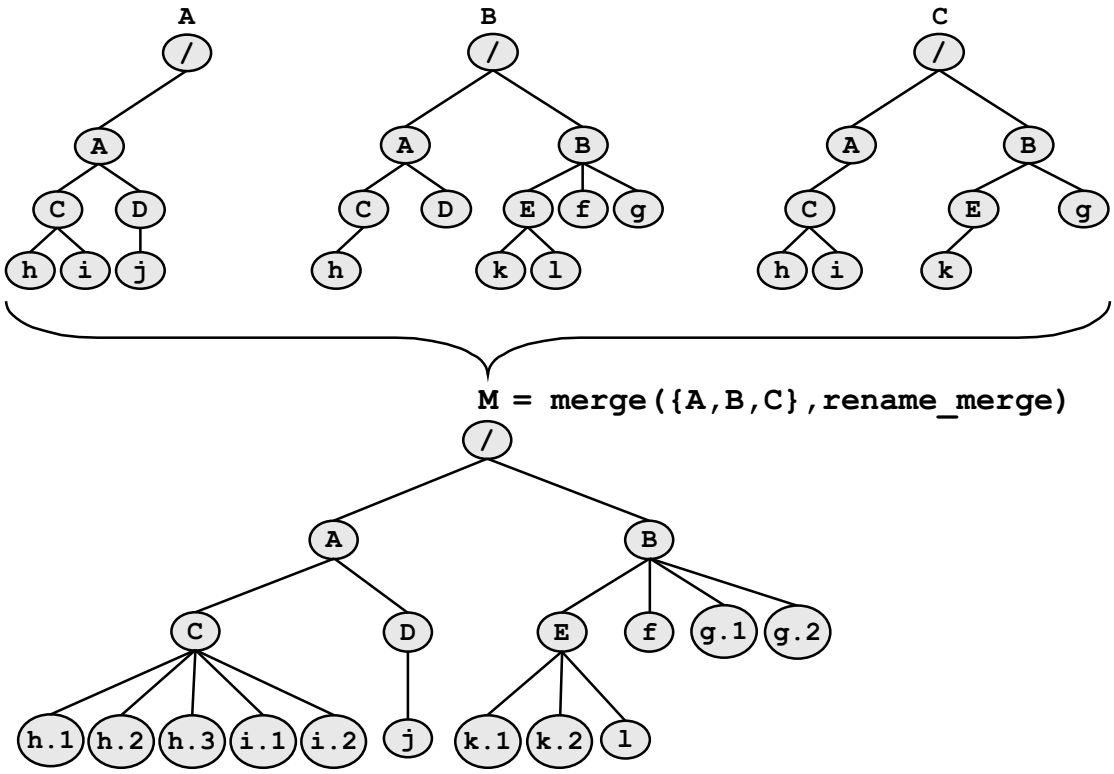

Figure 4.8 `merge` Tree Composition Operation.
The upper three trees, **`A`**, **`B`**, and **`C`**, are merged using the conflict function of Figure 4.7, resulting in the lower tree **`M`**. As specified by the conflict function, common directories are merged into a single directory in **`M`**, and common files are renamed by appending a version number.

- A union-over mount that lays the new name space over the original name space at the mount point, instead of replacing its contents, is specified as:

```
T = graft(prune(O,P), merge({N,subtree(O,P)}, overlay), P)
```

- A union-under mount that lays the new name space under the original name space at the mount point, instead of replacing its contents, is specified as:

```
T = graft(prune(O,P), merge({subtree(O,P),N}, overlay), P)
```

These simple mount examples show the ease in describing common name space compositions using FINAL's composition operations. We now proceed to discuss the flexibility afforded for specifying more diverse compositions that results from combining the tree algebra with prescriptive programming

capabilities. Section 4.3 provides additional example specifications that fully demonstrate the flexibility of name space composition using FINAL.

4.2.4 Prescriptive Language Capabilities

While the declarative, functional nature of FINAL's composition operations is appealing due to its simplicity, there exist interesting composition strategies that are hard to describe using only these operations. To meet our goals for providing flexibility in writing practical specifications, prescriptive specification constructs are necessary to support name space, file attribute, and host context queries. For example, it would be useful to allow iteration over directory entries to find files that match some name pattern or have sizes that exceed some threshold.

Rather than design a completely new language to add prescriptive functionality, we provide the FINAL abstractions and composition operations within an existing programming language. Cinquecento [126] is a dynamically-typed functional language that supports C syntax and operators. Cinquecento programs are sequences of expressions that are dynamically evaluated in order. The language provides many useful functional capabilities such as lambda expressions as well as built-in types for high-level data structures including lists, vectors, and dictionary hashes. Cinquecento's C-based syntax should feel familiar to systems programmers, and its functional characteristics allow FINAL's composition operations to be used in a natural manner. Although we consider Cinquecento a natural fit as the basis for FINAL, it is important to note that the flexibility that results from combining FINAL abstractions and composition operations with prescriptive capabilities is not dependent on the choice of Cinquecento. Similar benefits could be realized using other dynamically evaluated languages.

We have added two new data types to Cinquecento, `filesvc` and `nstree`, and functions that operate on these types to support FINAL's file service and tree abstractions. The `filesvc` type represents an instance of a defined file service. Table 4.10 presents the interfaces for `filesvc`, and Table 4.11 presents the interfaces for the file operations supported by file services. The `mkfilesvc`

Construct a tree from a file service.
`mknstree( svc ) → nstree`

Construct an empty tree.
`nulltree() → nstree`

Walk path in tree to target tree. Returns nil for bad path.
`treewalk( nstree, path ) → nstree | nil`

Retrieve names for children of tree.
`treelist( nstree ) → string[]`

Get vertex name, file service path, or file service for tree.
`treename( nstree ) → string`
`treepath( nstree ) → path`
`treesvc( nstree ) → filesvc`

Table 4.9 `nstree` Interfaces

Define new file service with the given name and a set of functions for the file operations in Table 4.11.
`svcdefine( name, {file_op_fns} ) → int`

Construct an instance of the named file service; "..." is a set of service specific options.
`mkfilesvc( name [, ...] ) → filesvc`

Table 4.10 `filesvc` Interfaces

function creates a new instance of the named file service. `mkfilesvc` has optional parameters that can be used to customize the service instance, such as passing host and mount point information for a remote file service like NFS [105] or 9P [65]. For convenience, a pre-defined service named `local` provides access to the local name space. Additional file services can be defined with the `svcdefine` function, which associates the given service name with a set of functions implementing the service file operations shown in Table 4.11.

The `nstree` data type represents a vertex in a name space tree. Each `nstree` corresponds to a name space entry on some file service, and contains a path and `filesvc` reference. To maintain the name space organization, each `nstree` has references to its parent and children `nstree` vertices. Initial views for a service are created using the `mknstree` function, which takes a `filesvc` operand and

Query the status of a file with given path in the provided file service.
`stat( filesvc, path ) → fileattr`

Open a file with given path in the provided file service according to specified access mode.
`open( filesvc, path, mode ) → fd`

Close the file associated with the given descriptor in the provided file service.
`close( filesvc, fd ) → int`

Read the file associated with the given descriptor in the provided file service.
`read( filesvc, fd, offset, buf, count ) → long int`

Write the file associated with the given descriptor in the provided file service.
`write( filesvc, fd, offset, buf, count ) → long int`

Get the entries for the given directory path in the provided file service.
`list( filesvc, path ) → string[]`

Table 4.11 File Service Operation Interfaces

returns the root `nstree` for a tree that has an identical structure to the physical name space of the service. Table 4.9 presents interfaces for creating and using the `nstree` data type.

4.3 Example Compositions using FINAL

We provide example FINAL specifications for composing private and global name spaces. Specifications describe a composed name space using a sequence of statements that are evaluated in order. Statements correspond to either tree composition operations or prescriptive constructs. A special variable named `root` is used to identify the fully-composed name space. The value assigned to this variable should be of type `nstree`. For example, a specification to access the sub-tree of the local name space located at the path `/usr/bin` would be:

```
LOCAL = mknstree( mkfilesvc("local") );
root = subtree( LOCAL, "/usr/bin" );
```

To help distinguish the name space trees from other Cinquecento variables in the example specifications, we use variable names in uppercase for trees and lowercase for other types.

4.3.1 Private Name Spaces

Private file name spaces are custom views of the default file system name space. Common uses for this customization are for user convenience or system security. For convenience, a process may select some subset of the local name space that is necessary for execution. The reduced name space often makes access to target files more efficient, as the files are placed closer to the root of the name space. In the security realm, private name spaces can be used to prevent unauthorized access to files through isolation or omission. The example specifications presented in Figure 4.12 show how FINAL can be used to generate such private name spaces.

A common use of private name spaces is when a user wishes to substitute their own executables for the system installed versions. Figure 4.12(a) shows how to accomplish such a task using FINAL.

Figure 4.12(b) shows the use of procedural constructs to deal with heterogeneous context. It checks for the presence of three common paths for temporary storage within the local name space and chooses the first available, and defaults to the user's home directory if none of the paths were found. This example demonstrates Cinquecento's support for accessing the environment, which is useful for parameterizing specifications based on context.

Figure 4.12(c) uses name space navigation and attribute inspection to generate a name space containing files from the `/var/log` directory that have sizes larger than four kilobytes. Two functions, `treewalk` and `treelist`, are used for name space navigation. As described in Table 4.9, each function has a `nstree` operand that indicates the node from which navigation begins. `treewalk` takes a relative path operand, walks the path to the target `nstree`, and returns the target or the Cinquecento special value `nil`. `treelist` returns a list of string values for the names of children of the current `nstree`. In the example, `treelist` retrieves the directory entries of `/var/log`, and `treewalk` is used within a lambda function applied to each entry. The specification also shows how to query attribute information for the file referenced by a `nstree`. The query is accomplished using the `stat`

```
LOCAL = mknstree( mkfilesvc("local") );
HOME = subtree( LOCAL, getenv("HOME") );
MYBIN = subtree( HOME, "/install/bin" );
BIN = extend( MYBIN, "/usr/bin" );
REST = prune( LOCAL, "/usr/bin" );
root = merge( {REST, BIN}, overlay );
```

(a) Substituting Personal Executables

```
user = getenv("USER");
LOCAL = mknstree( mkfilesvc("local") );
TMP = subtree( LOCAL, "/tmp/" + user );
if( TMP == nil ) {
    TMP = subtree( LOCAL, "/scratch/" + user );
    if( TMP == nil ) {
        home = getenv("HOME");
        TMP = subtree( LOCAL, home );
    }
}
root = graft( nulltree(), TMP, "/mytemp" );
```

(b) Finding Temporary Storage

```
LOCAL = mknstree( mkfilesvc("local") );
LOG = subtree( LOCAL, "/var/log" );
RESULT = nulltree();
check_entry = @lambda(x){
    ent = treewalk( LOG, x );
    attr = stat( treesvc(ent), treepath(ent) );
    if( isfile(attr) && attrsize(attr) >= 4096 )
        RESULT = graft( RESULT, ent, "/" + treename(ent) );
}
entries = treelist( LOG );
foreach( check_entry, entries );
root = RESULT;
```

(c) Selecting Log Files

```
REST = mknstree( mkfilesvc("local") );
excludes = [ "/etc", "/root", "/sbin", "/usr/sbin", "/var/log" ];
excl_fn = @lambda(x) {
    REST = prune( rest, x );
}
foreach( excl_fn, excludes );
root = REST;
```

(d) Hiding Sensitive Files

Figure 4.12 FINAL Specifications for Private Name Spaces.
(a) Substituting personal versions of executables for those provided by the system.
(b) Finding a directory providing temporary storage using local context information.
(c) Selecting log files that match some criteria using name space navigation and attribute queries.
(d) Hiding sensitive sub-trees of the name space using pruning.

method described in Table 4.11, which takes a `filesvc` and path and returns a `fileattr`. A `fileattr` object contains information similar to a `struct stat` as used by the POSIX `stat` operation.

System administrators may desire to exclude sensitive portions of the system name space from the view of regular user processes. The specification shown in Figure 4.12(d) supports such exclusion using a lambda function to `prune` excluded paths.

4.3.2 Global Name Spaces

Global file name spaces combine many distributed name spaces into a single view that allows programs to operate on distributed files as if they were local. Such name spaces are useful in a variety of contexts including distributed file systems, single-system image (SSI) environments, and cloud computing. We present several examples that show how programs can specify global name spaces.

We assume a global name space is composed using a two step process. First, a FINAL specification is evaluated at each independent file server to generate a server-local name space. Second, the global name space is composed by applying the FINAL `merge` operator to the set of server-local name spaces. Chapter 5 discusses our approach for implementing this process in a scalable manner within TBON-FS. For each example, we provide the FINAL specification that is evaluated at each file server, and show the global name space that is generated. Portions of the global name space that are not relevant to the effects of local name space composition are elided.

In all examples, the `file_group_merge` conflict resolution function shown in Figure 4.13 is used in the global name space `merge`. Similar to the `rename_merge` function presented in Figure 4.7, `file_group_merge` produces a single directory for each common directory path. For common file paths, however, the function produces a new directory at the common path. This group directory is populated with the conflicting files, after renaming the files using a version number.

Our first example shows a custom name space composition for a program that wants to use group file operations. The target group often corresponds to a file having the same path on all servers.

```
file_group_merge( shared_path, conflicts )
{
    results = [ : ]; // initialize empty hash table
    if( all_directories(conflicts) ) {
        // return single, common directory
        results[shared_path] = conflicts[0];
        return results;
    }
    else if( all_files(conflicts) ) {
        // return all files in new group directory
        for( int i=0; i < length(conflicts); i++ ) {
            // new file path is group directory path plus version
            p = shared_path + "/" + i;
            results[ p ] = conflicts[i];
        }
        return results;
    }
    // bad input structure
    return nil;
}
```

Figure 4.13 `file_group_merge` Conflict Resolution Function.
Places all files having the same path in a new group directory, and renames files using a version number. Conflicting directories are merged to a single directory.

```
// overlay files within /groups
grps = [];
LOCAL = mknstree( mkfilesvc("local") );
grps.append( subtree(LOCAL,"/proc/cpuinfo") );
grps.append( subtree(LOCAL,"/proc/meminfo") );
grps.append( subtree(LOCAL,"/var/log/syslog") );
root = graft( nulltree(), merge( grps, file_group_merge ), "/groups" );
```

(a) FINAL Specification

```
/groups/
       /cpuinfo/
               /[0-99999] # files from servers 0 to 99999
       /meminfo/
               /[0-99999]
       /syslog/
              /[0-99999]
```

(b) Global Name Space

Figure 4.14 Automatic File Groups: Specification and Name Space.
(a) Server FINAL specification that selects target files.
(b) Global name space with files in group directories.

```
// get hostname from environment
myhost = getenv("HOST");
LOCAL = mknstree( mkfilesvc("local") );
root = graft( nulltree(), LOCAL, "/hosts/" + myhost );
```

(a) FINAL Specification

```
/hosts/
      /host0/...
      /host1/...
      /host2/...
      /...
      /host9999/...
```

(b) Global Name Space

Figure 4.15 Distributed Hosts: Specification and Name Space.
(a) Server FINAL specification that prepends hostname to local name space.
(b) Global name space with hosts in separate sub-trees.

Figure 4.14 shows an example server specification that merges sub-trees corresponding to three file paths, and the resulting global name space. After the merge, group directories have automatically been created due to the conflict resolution employed.

Several systems [10,13,21,50,70,116] provide a global name space that is partitioned on organizational or host boundaries. Figure 4.15 shows a specification that results in a global name space organized into separate sub-trees for each host.

Figure 4.16 shows how to construct a global process space from many independent proc file systems. A renaming scheme is used to avoid conflicts between processes with the same numeric identifier on different hosts, which maintains a name space that can be used by utilities like `ps` and `top`. However, the use of this name space by existing utilities would be extremely inefficient, as they would need to iterate over hundreds of thousands of process directories. In Chapter 7 we discuss `ptop`, a parallel version of `top` that uses group file operations to provide the same interactivity and features when monitoring hundreds of thousands of distributed processes.

With the advent of cloud computing, global name spaces could be used to help manage the large distributed computing resources provided by providers such as Amazon, Google, and Yahoo.

```
// define pid remap function
rank = getenv("SERVER_RANK");
@define remap_pid(pid) {
    newpid = (rank * 100000) + atoi(pid);
    return string(newpid);
}
// retrieve pids and remap
LOCAL = mknstree( mkfilesvc("local") );
PROC = subtree( loc, "/proc" );
entries = treelist(proc);
pids = [];
for( i=0; i < length(entries); i++ ) {
    name = entries[i];
    if( name[0] > '0' && name[0] <= '9' ) {
        newname = remap_pid( name );
        REMAP = extend( subtree(PROC,name), "/proc/" + newname );
        pids.append( REMAP );
    }
}
// merge pids
root = merge( pids, overlay );
```

(a) FINAL Specification

```
/proc/
     /1/...          # server 0, pid 1
     /...
     /100001/...     # server 1, pid 1
     /...
     /9999900001/... # server 99999, pid 1
     /...
     /9999932768/... # server 99999, pid 32768
```

(b) Global Name Space

Figure 4.16 Global Process Space.
(a) Server FINAL specification that remaps process ids.
(b) Global name space that includes all processes.

Figure 4.17 shows how server-local context can be used to organize files by the type of system (e.g., the operating system and machine architecture). Cloud system administrators could use the resulting name space to simplify common tasks such as software installation and update.

4.4 Summary

FINAL is a language for custom file name space composition using an intuitive tree abstraction for name spaces. FINAL provides common tree composition operations with clear semantics, as well as a

```
// get OS and machine architecture from environment
os = getenv("OSTYPE");
arch = getenv("MACHTYPE");
// get executables and libraries
LOCAL = mknstree( mkfilesvc("local") );
BIN = subtree( LOCAL, "/usr/bin" );
BIN = extend( BIN, "/bin" );
if( strstr(arch, "64") == nil )
    LIB = subtree( LOCAL, "/usr/lib" );
else
    LIB = subtree( LOCAL, "/usr/lib64" );
LIB = extend( LIB, "/lib" );
// place files according to local context
MYTREE = merge( [ BIN, LIB ], overlay );
osarch = "/os/" + os + "/" + arch;
OSTREE= graft( nulltree(), MYTREE, osarch );
root = OSTREE;
```

(a) FINAL Specification

```
/os/
   /Linux/
         /ppc64/...
             /bin/...
             /lib/...
         /x86/...
         /x86_64/...
   /Solaris/
           /sparc/...
           /sparc64/...
```

(b) Global Name Space

Figure 4.17 Heterogeneous Cloud: Specification and Name Space.
(a) Server FINAL specification that selects local executables and libraries.
(b) Global name space organized by operating system and machine architecture.

new `merge` operation supporting deep composition of large sets of trees with customizable name conflict resolution. The `merge` operation provides the basis for efficient composition of many name spaces and is amenable to scalable implementations within tree-based overlay networks. By combining FINAL's tree abstraction and operations with prescriptive language capabilities, programs are given the flexibility to describe a wide variety of name space compositions.

Chapter 5

The TBON File System

The group file operation idiom and FINAL provide useful semantics for operating on large distributed file groups and composing global name spaces. By avoiding explicit linear behaviors during group operations, these semantics enable implementations based on scalable techniques. In this chapter we present the TBON File System (TBON-FS), a new group file system that provides scalable group file operations and global name space composition. We discuss the principles used to guide the design of TBON-FS and the resulting user-level architecture in Section 5.1. Section 5.2 presents the scalable approach used for global name space composition within TBON-FS. The mechanisms TBON-FS uses to provide scalable group file operations are discussed in Section 5.3. We evaluate the performance of TBON-FS for global name space composition in Section 5.5; additional evaluations of group file operation performance are included with the case studies of Chapters 7, 8, and 9. An alternative design that supports group file operations at kernel-level in TBON-FS clients is evaluated in Section 5.6; this design was not pursued due to a lack of demonstrable performance improvement.

5.1 Designing a Group File System

We refer to TBON-FS as a *group file system* to emphasize its design as a system supporting group

file operations, and to clearly distinguish its target use case from existing file systems providing distributed file access, as previously discussed in Section 2.4. Our design for a group file system is guided by three principles: scalability, flexibility, and portability. We describe the key features of TBON-FS that are motivated by these principles, and then discuss the resulting architecture.

Scalability. Scalable approaches to distributed group operations must avoid linear-time behaviors, such as iterations that are proportional to the size of the group, in favor of constant-time or logarithmic-time behavior. Since data processing time often scales linearly with respect to the size of the data, scalable group operations also must incorporate methods for reducing or limiting the amount of data gathered, preferably in a manner that also reduces centralized analysis.

As evident from its name, TBON-FS employs a tree-based overlay network as its primary scalability mechanism. TBONs leverage the logarithmic properties of trees to provide scalable multicast communication from a root process to leaf processes and aggregation of data gathered from leaves. In tree-based aggregation, a function at a non-leaf tree vertex will be passed a list of input data, where each input datum is the output produced by the function at one child. To provide scalable performance, tree-based aggregations should be designed so that data does not grow as it flows toward the root. Thus, aggregations that produce a constant or decreasing amount of output data at each subsequent level are desirable. This ensures that processes near or at the root are not overloaded with data or computation.

A TBON architecture easily scales to very large distributed systems by simply increasing tree fan-out or depth. For example, a TBON-based debugging tool [60] was the first to run at full scale on the BlueGene/L system with over 200,000 processors. TBONs also provide natural hierarchical redundancy, which has beneficial properties for recovery and reconfiguration after faults [7].

We describe how TBON-FS uses MRNet [101], a general-purpose TBON API and infrastructure, to support global name space composition and group file operations using scalable techniques in Section 5.2 and Section 5.3, respectively. Extensions made to MRNet to support TBON-FS are

described in Section 5.4.

Flexibility. TBON-FS is designed to adapt to various client use cases. Clients are given control over several aspects of the system. Clients provide the TBON topology that dictates the structure of the tree overlay and the placement of overlay and server processes on hosts in the distributed system. To control the contents and organization of the TBON-FS global name space, clients provide FINAL specifications that are evaluated at servers to produce local name spaces that are merged to form the global name space. As previously described in Section 4.2, specifications are used to compose views of the name spaces exported by FINAL file services. To further extend the scenarios where group file operations can be used, TBON-FS supports synthetic (i.e., memory-resident) file services that export file interfaces for non-file resources on server hosts. For example, the `proc++` synthetic file service presented in Chapter 6 provides file-based control and inspection of process and thread groups. When used with the scalable group file operations provided by TBON-FS, `proc++` enables debugging of large parallel applications as we show in Chapter 9.

Portability. TBON-FS employs a purely user-level architecture, which enhances its portability and eases deployment. In contrast, kernel-level approaches are often platform-specific, limiting the type of distributed systems on which they can be used. Further, it is difficult and sometimes impossible to convice the owners of distributed systems to deploy kernel-level changes. User-level approaches are also easier to develop, test, and debug.

As shown in Figure 5.1, the architecture of TBON-FS consists of a library that is linked into client programs, server processes that run on distributed hosts, and the TBON that connects the client to servers. All TBON and server processes run in the context of the user executing the client program. This enables TBON-FS to rely on existing user authentication and access control mechanisms provided by a distributed system. As a result, TBON-FS can be used only on hosts to which the user can authenticate, and TBON-FS servers have the same permissions as the user when performing file operations.

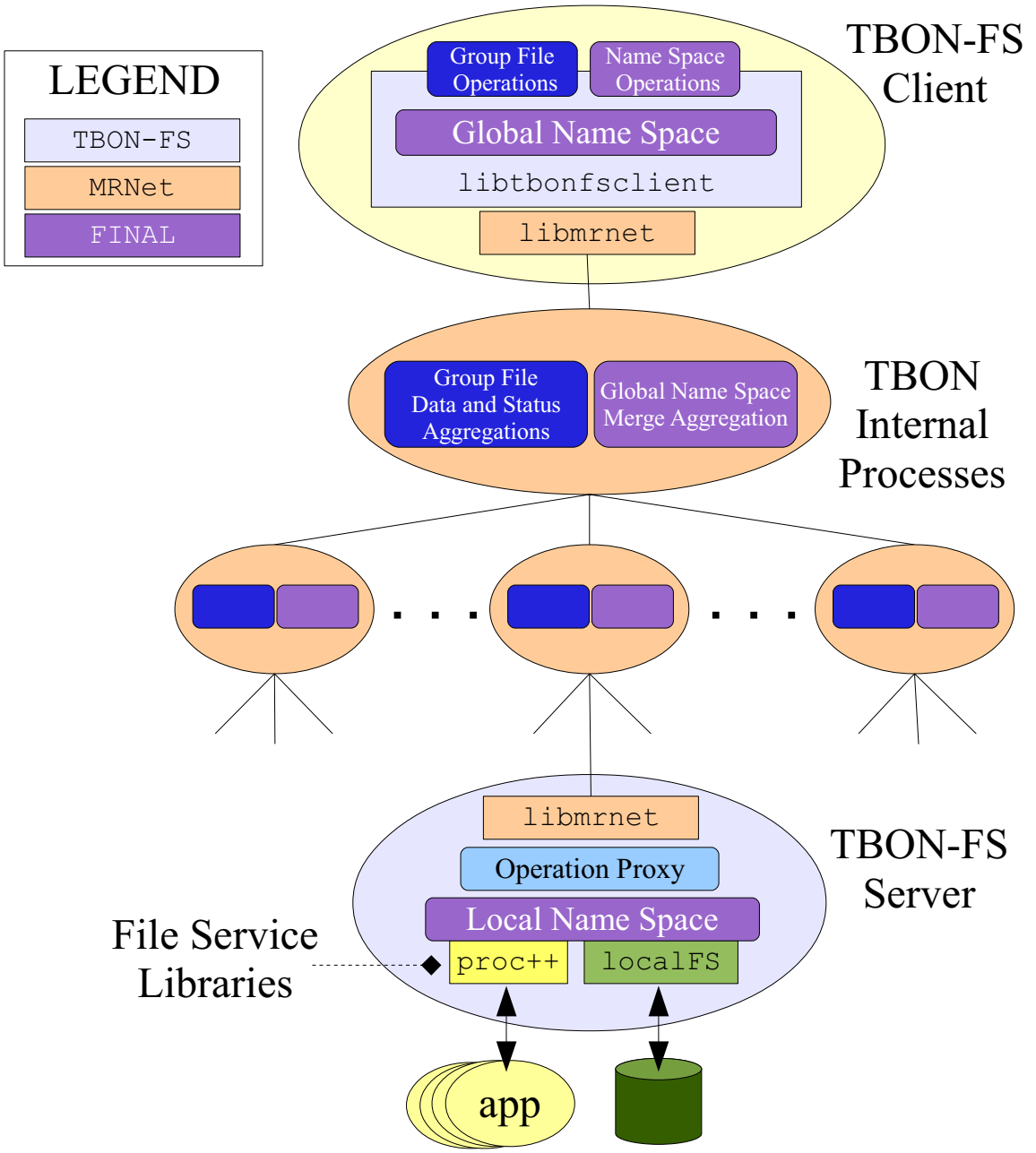

Figure 5.1 TBON-FS: A Scalable Group File System.
The figure provides a more detailed view of the TBON-FS architecture previously introduced in Figure 1.1. Client programs link with the `libtbonfsclient` library. The library provides group file and name space operations on entries in the TBON-FS global name space, which is a merged view of local name spaces at each TBON-FS server. The MRNet infrastructure, which consists of `libmrnet` and the TBON internal processes, is used to connect the client to servers. Each server provides a proxy service that translates requests received from the client into operations on entries in the local name space. Server local name spaces are composed from file services that have been registered with TBON-FS.

The client library exports C language interfaces for common file system functions including mounts, directory listings, status queries, and file I/O operations. The library also provides the new functions defined in Chapter 3 for use during group file operations, such as `gopen` and `gstatus`. Only group-aware versions of the file I/O operations are provided; these functions use the name prefix "`gfo_`" to clearly distinguish them from their standard single file counterparts. Single file operations are supported by simply treating them as operations on a group of size one.

To initialize TBON-FS, a client program first calls `mount_tbonfs`. As shown in Table 5.2, this function is modeled after the Linux `mount` function and has parameters for providing the mount point, the type of file system, mount flags, and the file system options. Included in the options passed to `mount_tbonfs` is the pathname of a file that describes the TBON topology. The client library instantiates an MRNet network using the provided topology, which results in the launch of all TBON internal processes and the TBON-FS servers.

A TBON-FS server is a proxy that provides operations on files and directories within a local name space. The local name space is composed by evaluating a FINAL specification provided by the TBON-FS client. Files and directories within this name space are exported by FINAL file services, which provide access to physical or synthetic file systems that are responsible for managing data storage. Each file service supports the operations previously defined in Table 4.11, and is implemented as a shared library.

Before a file service can be used in a specification, it must first be registered by the client using the `tbonfs_register_service` function defined in Table 5.2. This function has operands that define the name of the new file service, the shared library containing the service operations, and the names of initialization and finalization functions for the service. Upon calling `tbonfs_register_service`, the client distributes the operands to all servers, who attempt to load the shared library and find the addresses of the initalization and finalization functions. The servers then call the initialization function

```
int mount_tbonfs( const char* source, const char* mountpoint,
                  const char* fstype, unsigned long mountflags,
                  const void* options )
```

Initializes TBON-FS according to the supplied parameters. Returns `0` on success. If an error occurs, `-1` is returned and `errno` is set.

The `source` parameter specifies the pathname of a file containing a FINAL specification that is evaluated at each TBON-FS server to construct its local name space. The `mountpoint` parameter indicates the path in the TBON-FS client name space at which the composed global name space will be rooted.

The `fstype` parameter should be set to "`tbonfs`".

The `mountflags` parameter is used to specify a few common file system mount flags that have been assigned new semantics for use with TBON-FS. `MS_RDONLY` indicates that TBON-FS should support only read access. `MS_NOEXEC` indicates that programs cannot be executed on TBON-FS servers using `gexec`. `MS_REMOUNT` is used when TBON-FS has already been initialized, but the client wishes to add new composed global name spaces within the client name space by providing a new `source` and `mountpoint`.

The `options` parameter is used to pass file system options as a C-string containing pairs of the form "`option=value`" separated by semi-colons. On the first call to this function, `options` must contain a pair of the form "`TOPOLOGY_FILE=pathname`" to indicate the file containing the TBON topology.

```
int unmount_tbonfs( const char* mountpoint )
```

Unmounts the portion of the client name space rooted at `mountpoint`. If `mountpoint` is null, all active mountpoints are unmounted and TBON-FS is shut down. Returns `0` on success. If an error occurs, `-1` is returned and `errno` is set.

```
int tbonfs_list_servers( char** hosts )
```

Returns the number of servers. If `hosts` is non-null, it is expected to be an array of `char*` with length equal to the number of servers. The array is filled with the server hostnames.

```
int tbonfs_server_rank( char* hostname )
```

Returns the unique rank assigned to the server indicated by `hostname`, or `-1` if `hostname` is null or invalid. Server ranks are assigned within the range `[0, #_of_servers)`.

```
int tbonfs_register_service( const char* svcname, const char* svclib,
                             const char* svcinit, const char* svcfini )
```

Registers a new FINAL file service with name `svcname`. The shared library named by `svclib` will be loaded at all servers, and the initialization and finalization functions, `svcinit` and `svcfini`, will be located within the library. Returns `0` on success, or `-1` if registration failed.

Table 5.2 TBON-FS Client Library: Mount-related interfaces

to obtain an object that is used to make requests of the file service. The finalization function is used to safely clean up the state of the file service, and is called by a server when the view containing the service is unmounted. The `local` service that provides access to local file systems is automatically registered when each server is started.

All communication between client and servers takes place in the context of an MRNet stream, which is a multicast-reduction channel defined over a set of leaf processes (i.e., the servers). Each stream is defined to use specific data synchronization and transformation filters when data travels from leaves to the root. A synchronization filter controls how stream data received at a parent process in the TBON is treated before being passed to the transformation filter that implements the data aggregation. Custom synchronization and transformation filters can be implemented by MRNet users, which is crucial for supporting the custom data aggregation semantics of group file operations.

Calls to client library functions generate requests that are multicast to servers via the TBON. Servers then process the requests in parallel. Requests for group metadata are satisfied directly by a server, while group file operation requests are handled by making requests of one or more underlying file services. Results generated during request processing are sent back to the client using the TBON. The TBON aggregates results from all involved servers to produce a single result that is delivered to the client library.

The streams used for request distribution and response aggregation are chosen based on the request type. A global control stream is used for broadcasting requests to all servers and collecting their responses, as is necessary for requests involving name space mounts, file service registrations, and creation of group files. A control stream for each group file is established during `gopen` that allows for sending group file operation requests to only those servers that contain member files. Both client and servers record the aggregations bound to operations on a group file, so that the appropriate data aggregation streams are used to process group data and status results. The client library creates these streams

on demand when they are first used for a particular group file.

5.2 Composing a Global Name Space

The TBON-FS global name space is a composed view of the local name spaces at servers. This view is private to the client process, and is composed by applying a FINAL `merge` operation to the set of server local name spaces. Each server's local name space is constructed according to a FINAL specification provided by the client. The pathname of the file containing the specification is given as the `source` operand to `mount_tbonfs`. As previously introduced in Section 4.2, a `merge` composition produces a name space that contains unique paths from all input name spaces as well as the results from applying a conflict resolution function to shared paths that exist in multiple input name spaces.

TBON-FS uses two methods to improve the scalability of global name space composition. First, it uses the TBON to multicast the specification to servers, who initiate construction of their local name spaces in parallel. Second, it evaluates the results of the `merge` composition lazily when specific paths are accessed, such as during calls to `gopen` or `stat` or when listing directories. Lazy merging avoids the need for the client to keep the entire global name space in memory, which is a potential scalability barrier for large name spaces.

Lazy merging is implemented by using an MRNet stream for pathname resolution. After stripping the prefix corresponding to the mount point in the global name space, a target pathname is multicast to servers on this stream. Servers query their local name space to check if the target exists. If the target is found, each server sends a response on the stream that includes attributes for the matching directory entry and the server's rank. The aggregation used by this stream implements the semantics of the `merge` operation to combine matches from all servers. To resolve conflicts, TBON-FS uses the `file_group_merge` algorithm previously defined in Figure 4.13. As shown in Figure 5.3, this algorithm produces a single directory for each directory path that exists in multiple server name spaces. For file paths that exist on multiple servers, the algorithm produces a new synthetic directory at the shared

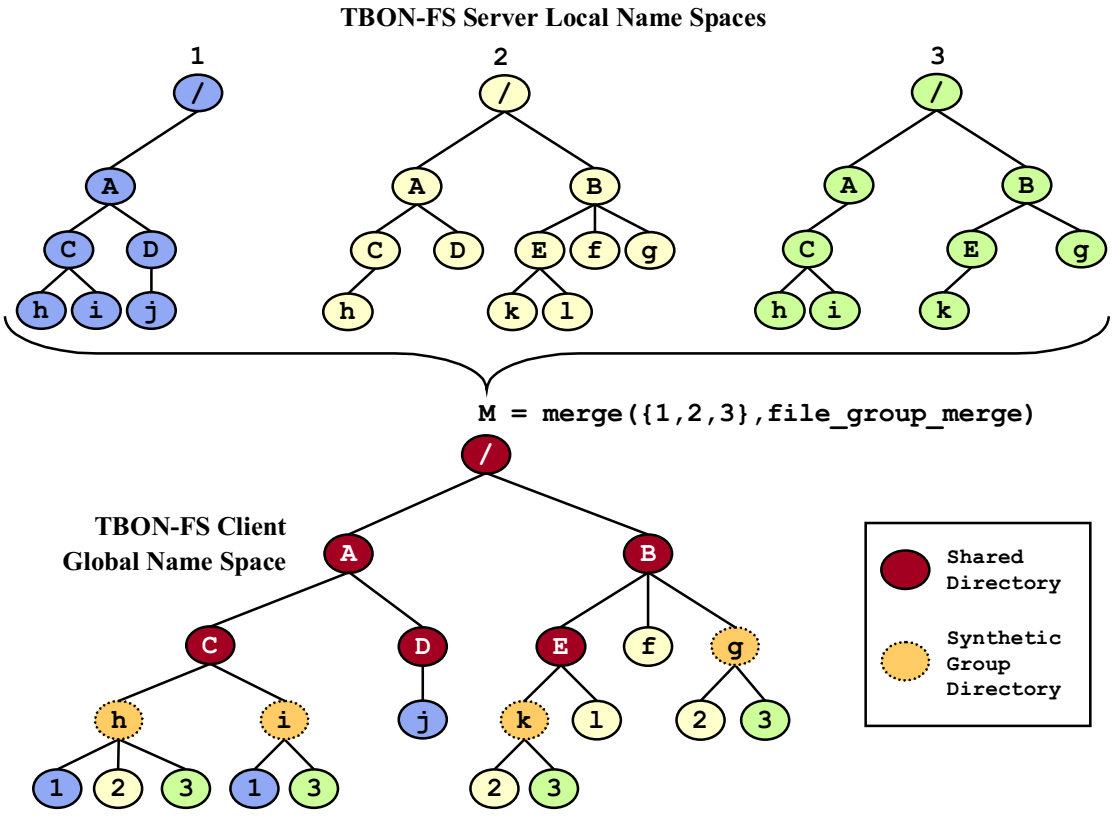

Figure 5.3 Composition of TBON-FS Global Name Space.
The global name space visible to a TBON-FS client is a merged view of the local name spaces. To resolve conflicts due to shared file paths, TBON-FS generates synthetic directories containing the conflicting files, which are renamed using the rank of the server from which they originate.

path, and populates it with the conflicting files after renaming them using a version number. TBON-FS uses the rank of the server from which a file originated for the version number. TBON-FS uses `file_group_merge` to make it easy for clients to generate the group directories used in group file operations. To create a group directory, the client simply needs to design a FINAL specification that results in the member files having the same path in all server local name spaces..

TBON-FS is able to use lazy merging due its choice of a conflict resolution function that only generates synthetic directories for shared file paths. Since files are always leaves in the name space, there is no possibility that synthetic directory paths unknown to servers will be accessed when resolving paths corresponding to a target file's ancestors in the name space. This allows TBON-FS servers to

treat directory listings for a path that maps to a file as a valid request that returns the file.

5.3 Operations on Distributed Files

TBON-FS uses three key techniques to improve the scalability of group operations on distributed files. Briefly, these techniques are:

- Use multicast streams for all communication with distributed hosts. Unicast communication is never used as it leads to iterative, non-scalable behavior.
- Use distributed data aggregation to reduce memory use and computational load at the client.
- Keep servers completely unaware of each other. Server independence ensures that operations can be executed in parallel across servers, and eliminates the potential for costly distributed consensus.

In this section, we discuss how these techniques are applied within the protocol used between the client and servers to process group file operations.

The group file operation idiom allows for parallel execution when member files are distributed across many file servers. With both `gopen` and group file operations, each file server can operate independently on target files in their local name space. To take advantage of the independence of servers, TBON-FS employs a synchronous protocol where the client library multicasts requests to a set of servers and then waits to receive an aggregated response from all targeted servers.

The use of a synchronous protocol simplifies interactions between the client and servers, which results in a design that is easier to maintain and extend. A synchronous design also helps to quickly identify performance bottlenecks and provides strong motivation for eliminating them when found. Somewhat unexpectedly, this synchronous protocol works well even for POSIX asynchronous I/O operations such as `aio_read` and `lio_listio`, due to their use of synchronous actions for starting operations, completion notification, and result collection. An extra benefit of supporting asynchronous I/O is that `lio_listio` is useful for batched delivery of a series of group file operation requests, which reduces network utilization and may improve the latency for completing the requests.

The TBON-FS request-response protocol is similar to classical remote procedure calls [14], but extends the RPC idiom to enable a single request to be processed on many remote hosts. Requests include a tag that identifies the desired action by servers. Auxiliary data values needed to satisfy the request, such as the operands to a group file operation, are also included with the request. Upon receiving a request, the tag is used to identify a handler function that is called by the server. The handler function extracts the auxiliary data from the request, performs the desired action, and sends generated results back to the client.

Results produced at servers are aggregated within the TBON before being delivered to the client. The aggregation used is dependent upon the request type. For example, name space lookup results are aggregated by a stream providing the semantics of the FINAL `merge` operation, while group file operation status and data results are processed using the aggregations bound to the operation by the client.

As is necessary for all tree-based computations, the interface to TBON-FS aggregation functions is structured to support hierarchical execution. In tree-based aggregation, a function at a non-leaf tree vertex will be passed a list of input data, where each input datum is the output produced by the function at one child. TBON-FS uses the `file_data` structure shown at the top of Figure 5.4 to pass input data. This structure contains raw or aggregated file data and a list of group file member indices. We define raw data as unprocessed data that has been read from files at the servers. For raw data, the member list will contain a single integer denoting the origin file index (i.e., the value that is returned by `gindex`). With aggregate data, the list will contain each of the indices that contribute to the aggregated result. The aggregation function accepts an array of `file_data` structures as an input, and stores aggregated results in a single `file_data` output structure. An example TBON-FS aggregation function is shown in Figure 5.4. This is a modified version of the aggregation function used in the custom load average calculation from Figure 3.7. Custom aggregation functions defined by users for use in group file operations are required to provide the same interface.

```
struct file_data {
    unsigned data_len;  // length of data
    void* data;         // data (any format)
    unsigned files_len; // number of files
    int* files;         // group file indices
}

int calc_load_avgs( char* params_fmt, va_list params,
                    unsigned num_inputs, struct file_data inputs[],
                    struct file_data* output,
                    void** aggregation_state )
{
    // calculate average of inputs
    double avg1 = 0, avg5 = 0, avg15 = 0;
    unsigned total_num_files;
    for( unsigned u=0; u < num_inputs; u++ ) {
        int nfiles = inputs[u].files_len;
        void* idata = inputs[u].data;
        if( nfiles == 1 ) {
            // scan raw file data
            double one, five, fiftn;
            scandata(idata, &one, &five, &fiftn);
            avg1 += one;
            avg5 += five;
            avg15 += fiftn;
        } else {
            // data contains aggregated loads
            double* avgs = (double*) idata;
            avg1 += avgs[0] * nfiles;
            avg5 += avgs[1] * nfiles;
            avg15 += avgs[2] * nfiles;
        }
        total_num_files += nfiles;
    }
    avg1 /= total_num_files;
    avg5 /= total_num_files;
    avg15 /= total_num_files;

    // prepare output data
    allocate_output_data( output );
    store_averages(output, avg1, avg5, avg15);
    fill_output_file_list( inputs, output );
}
```

Figure 5.4 TBON Data Aggregation Function
An example data aggregation function that supports hierarchical execution.

Aggregated data is never cached within TBON-FS. This policy was chosen for two reasons. First, the typical data access pattern used for group file reads is sequential, mirroring that of single file reads. Thus it is unlikely that the same data will be read again. Second, in scenarios when data is read more than once, either the client is using a different aggregation to supplement the information gleaned dur-

ing the prior reads, or the client expects that the file data has changed. In both cases, caching of the previous result provides no benefit.

TBON-FS also avoids caching global state for group files at the client. We define global state as complete information about some aspect of the group, such as the list of all member files. We assume the size of global state data grows with the group size. For large groups, keeping global state at the client requires significant memory use, and can lead to high computational load when scanning data to extract information for an individual member. To avoid such inefficiencies, the client library keeps only summary information such as the group size for each group file. When global state is required, such as when `gstatus` or `gfiles` is used, the TBON is used for on-demand, efficient collection of information from the servers.

5.4 Extensions to MRNet

During the development of TBON-FS, we extended the capabilities of MRNet. Although features were added to meet specific TBON-FS requirements, all extensions are general-purpose and have subsequently been used in other MRNet-based tools.

MRNet originally provided data aggregation only within root and internal processes of the TBON. However, executing aggregation functions at the leaves is an important technique for scalable data management. For example, when filtering is used to select only relevant data, executing the filter at the data source keeps unused data from being sent over the network. The ability to query local host context at the data source is also useful, such as when mapping a numeric user id to the associated user's name. For these reasons, we extended MRNet to support data aggregation at leaves.

We further extended MRNet's data aggregation capabilities by adding the concept of aggregation function parameters. Parameters are defined by the root process for a particular data stream, and provide persistent data inputs to the stream's aggregation function. Each time the stream executes its aggregation function at any TBON process, the parameter values are passed as inputs. Thus, parame-

ters can be used to control the behavior of aggregation functions. For example, an aggregation function that performs data binning can use parameters to control the number or width of bins. The root process is free to set new parameter values at any time during the stream's lifetime, which is useful for feedback-based control. Although aggregation parameters are rarely used internal to TBON-FS, they have proved useful for many client data aggregations during group file reads.

The last two extensions to MRNet were motivated by the need to deliver polling events from servers to the client in a timely manner. Because a separate event stream is used for each group file, the TBON-FS client needed a mechanism for being notified when data was available on a particular event stream. Originally, MRNet provided notification when any stream had data available. As such, we added per-stream data notifications to the MRNet API. The second "extension" was implementing the time-out based synchronization filter described in the very first MRNet paper [101]. Although the API guide for all MRNet software releases indicated this filter was supported, it was never implemented until required by TBON-FS.

5.5 Evaluation

Group file operation performance is dependent upon both the target file group and the data aggregations employed, which are both chosen by TBON-FS clients. As such, the case studies of Chapters 7, 8, and 9 serve to evaluate the scalability and performance of group file operations. Here, we evaluate the scalability of global name space composition within TBON-FS, and the performance of name space inspection operations on the composed name space.

The four global name space compositions presented in Chapter 4 were used to evaluate TBON-FS. We refer to these name spaces as "`automatic-groups`", "`distributed-hosts`", "`global-proc`", and "`heterogeneous-cloud`", respectively. We briefly review the contents of each name space; the name spaces can be seen in Figures 4.14, 4.15, 4.16, and 4.17. The "`automatic-groups`" name space contains group directories representing the collection of files from every server that exist

at three common paths. Each composed group directory has one entry per server. The "`distributed-hosts`" name space places each server's name space in a separate sub-tree rooted at `/hosts/<hostname>`, similar to the partitioned global name spaces found in many distributed file systems. The "`global-proc`" name space places process directories from the `proc` file systems of many servers within a single global `/proc`. Process directories are renamed to indicate the server rank and process id. The "`heterogeneous-cloud`" name space organizes a collection of heterogenous machines in a cloud environment according to their operating system and processor architecture. This name space is structured to ease software upgrades for machine-dependent executables and libraries.

The system used for the evaluation was JaguarPF, a Cray XT5 system located at Oak Ridge National Laboratory. JaguarPF contains over 18,688 compute nodes. Each node contains two six-core AMD Opteron 2435 processors and 16GB of DDR2-800 memory, for a total of 224,256 processing cores. Nodes are connected in a three-dimensional torus topology by a high-bandwidth, low-latency SeaStar 2+ network. The balanced TBON topologies used for each experiment size are found in Table 5.5. Topologies are given as the fan-out for each level starting with the root. TBON-FS servers were run on separate hosts from those running the internal tree processes. Twelve servers were run on each host (one per available processor) to virtually increase the number of available hosts. Reported results represent averages of ten measurements.

Our first experiment measured the time to construct the global name space. We measured the total latency observed at the client, and the per-server time to evaluate the FINAL specification and construct the server's local name space. Figure 5.6 shows the results for the four compositions. Figure 5.6(a) shows the average time in microseconds to construct each server's local name space, and Figure 5.6(b) shows the total latency for global name space composition as observed at the client. Across all scales, servers require little time to construct their individual name space, regardless of the input specification. At the client, we see excellent performance, as we are able to compose the global

Number of Servers	TBON Topology (fan-out per tree level)
2592	2 × 36 × 36
5184	4 × 36 × 36
10368	8 × 36 × 36
20736	16 × 36 × 36
31104	24 × 36 × 36
46656	36 × 36 × 36

Table 5.5 TBON Topologies used in Global Name Space Experiments

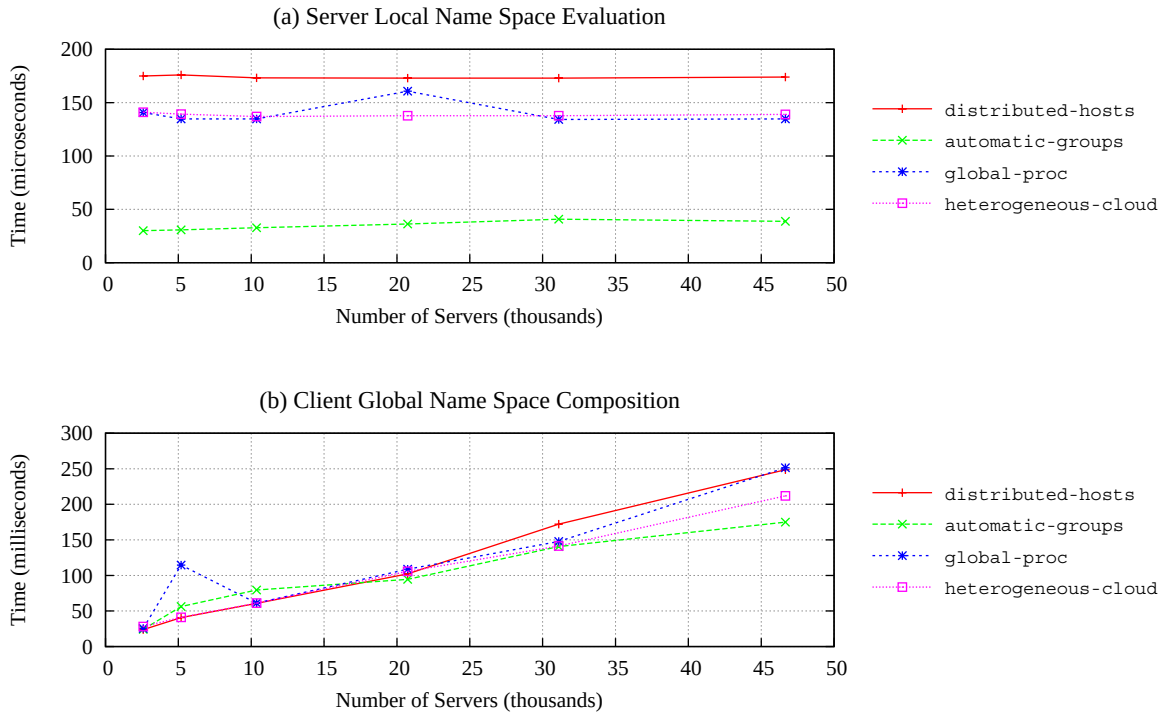

Figure 5.6 TBON-FS Global Name Space Composition Latency
(a) Average time to evaluate the FINAL specification and construct the local name space at individual servers.
(b) Global name space composition time observed by the TBON-FS client.

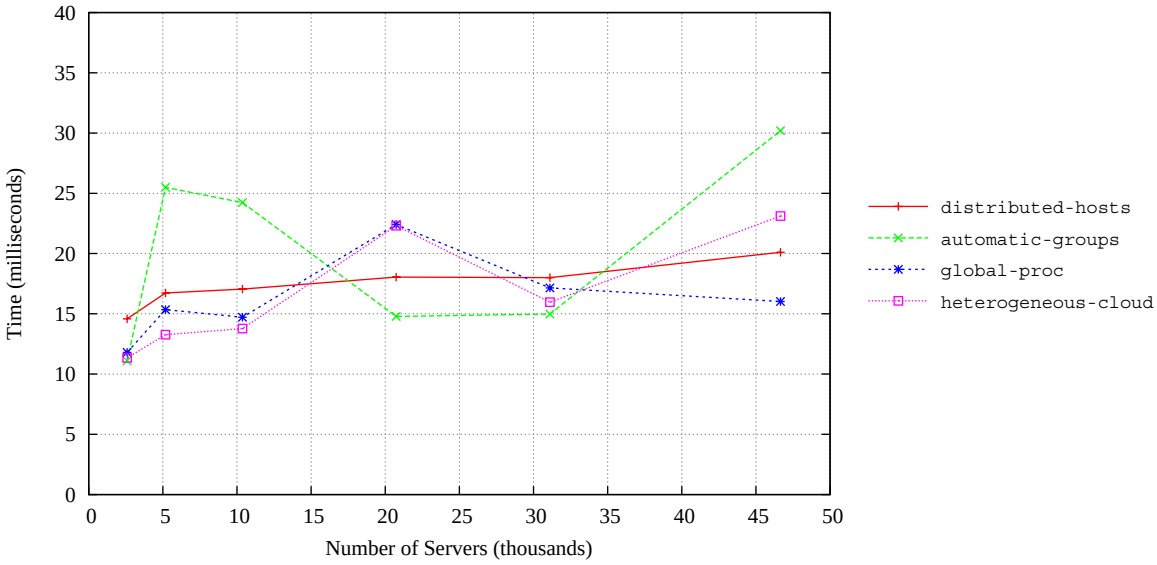

Figure 5.7 TBON-FS Global Name Space File `stat`
Average time in milliseconds to `stat` a single file in each of the four global name spaces.

name space for over 45,000 servers in approximately 250 milliseconds.

The next two experiments were conducted to evaluate the performance of name space inspection operations within composed global name spaces. Inspection operations were used on regular files and directories that reside at a single server host, as well as synthetic group directories that contain files or directories from many servers.

Figure 5.7 shows the time to `stat` a regular file residing on a single server within each of the four global name spaces. Because the TBON-FS client has no knowledge of the particular server on which a target file resides, the pathname for the target is multicast to servers. The server that contains the target then returns the requested status information. As expected, the time required does not vary much as the number of servers is increased, or across the four name spaces. The slight uptrend in the time can be attributed to increased fan-outs in the TBON at larger scales, which affect the time necessary to multicast the pathname. We attribute the occasional latency spikes to random interference from other applications using the JaguarPF network.

Figure 5.8 reports the time to gather the list of entries for directories within the global name

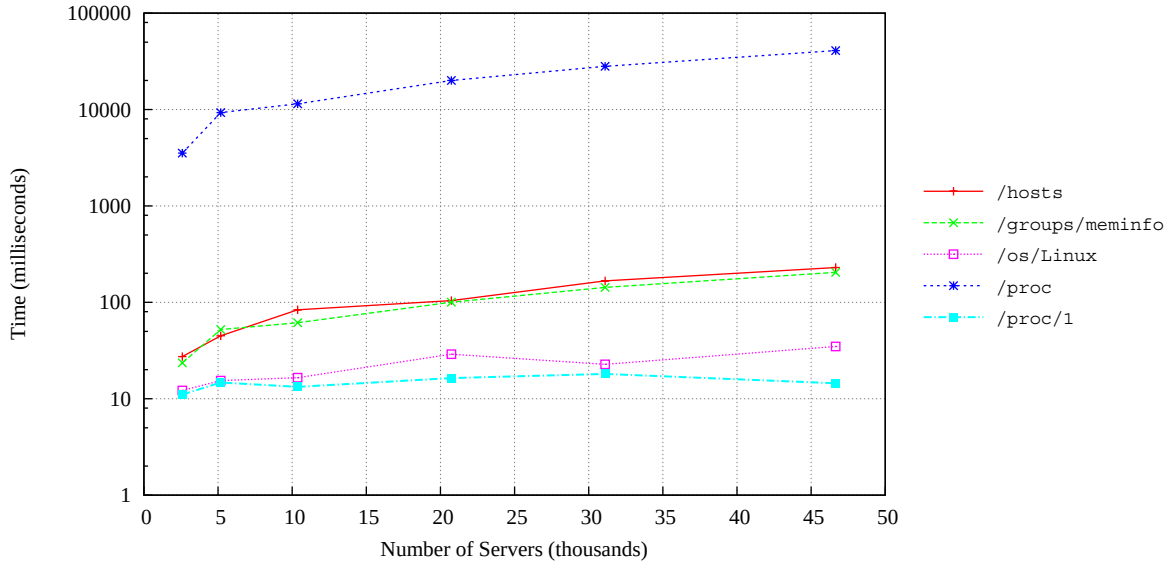

Figure 5.8 TBON-FS Global Name Space Directory Listing
Average time to list the entries of various group directories and a regular directory in the global name spaces. Table 5.9 provides additional information describing the listed directories.

Directory Path	Global Name Space	# Directory Entries	Description of Entries
`/hosts`	`distributed-hosts`	`#servers`	one directory per server
`/groups/meminfo`	`automatic-groups`	`#servers`	one file per server
`/os/Linux`	`heterogeneous-cloud`	`3`	platform directories: `x86`, `x86_64`, `ppc64`
`/proc`	`global-proc`	`#servers × ~100`	one directory for each process on every server
`/proc/1`	`global-proc`	`28`	contents of Linux `/proc/1/` directory

Table 5.9 Characteristics of the Directories Listed in Figure 5.8

spaces. Table 5.9 describes the relevant characteristics of the target directories, including the name space in which they are found and information about the number and type of entries.The first four directories in the table are synthetic group directories that contain entries from many servers. The last directory, "`/proc/1`", is a regular directory from a single server. The results indicate that the time to

list a directory is proportional to the number of entries. This linear behavior is expected, due to the need to copy the name of each directory entry to a buffer in the client. From the figure, it is clear that listing group directories such as "`/proc`" that contain millions of entries is an expensive operation and should be avoided. We also see that the time to list the regular directory is similar to that observed when applying `stat` to a regular file.

5.6 Kernel-level Group File Operations

Early on in the design of TBON-FS, we considered an alternative kernel-level approach for supporting group file operations at clients. We first discuss the potential transparency and performance benefits of kernel-level client support, and then describe our design for extending the Linux operating system to support group file operations. Next, we describe our experience with a prototype implementation of this design that failed to provide the expected performance benefits. This experience motivated our decision to focus on a purely user-level client design. As such, the kernel-level client implementation is not used in our case study evaluations, and it is safe for readers who are not interested in the details to skip this section.

Although our user-level approach to supporting TBON-FS clients eases development and improves portability, there are transparency limitations compared to a kernel-level implementation. One limitation is that existing programs that have not been linked with the library cannot access and use TBON-FS files. Dynamic library interposition [31] could be used to enable this by redirecting file system calls to the corresponding client library functions. Although we have not implemented this technique, it is quite common and we do not foresee any obstacles. Another limitation in using a library is that it is difficult to provide functionality that uses signals to notify the client program, such as is supported by POSIX asynchronous I/O.

A kernel-level TBON-FS client would eliminate these limitations, but requires extending the file system call interfaces of the operating system to support group file operations. However, extensions

that are specific to one file system are generally considered taboo. Thus, the argument for kernel-level support requires additional benefits. We hypothesized two such benefits. First, kernel-level support provides an opportunity for other distributed or parallel file systems to directly support group file operations, which may lead to improved performance and scalability when operating on large groups of distributed files. Second, when operating on groups of local files, kernel-level support may provide improved performance versus a series of user-level file operations that require many costly transitions and data copies between kernel and user space.

Based on these potential transparency and performance benefits, we designed extensions to Linux that provide basic support for group file operations. Our design requires no changes to existing group-unaware file systems, but allows for new group-aware file systems such as TBON-FS to directly support group file operations. The design also ensures that the performance of single file operations is not adversely affected.

Modern operating systems use a file system abstraction layer that maps a general file system interface onto a variety of file system implementations. The Virtual File System (VFS) layer in Linux defines a common file model [15,63] that provides abstractions for files, directories, links, inodes, superblocks, and file operations, and requires file systems to conform to these abstractions. Our design extends the VFS layer, and is likely applicable to other operating systems that use a similar file system abstraction layer. Figure 5.10 shows how file operations are mapped to functions within specific file systems in Linux. When a user application performs a file operation, the corresponding system call provided by the operating system is executed. System calls are redirected to functions within the VFS layer that map the common file operations to calls to specific file system functions. From an object-oriented perspective, the VFS mapping is similar to having methods of a virtual base class resolve to implementations in a derived class object.

The Linux system call layer uses the file descriptor to retrieve a pointer to the `struct file` for

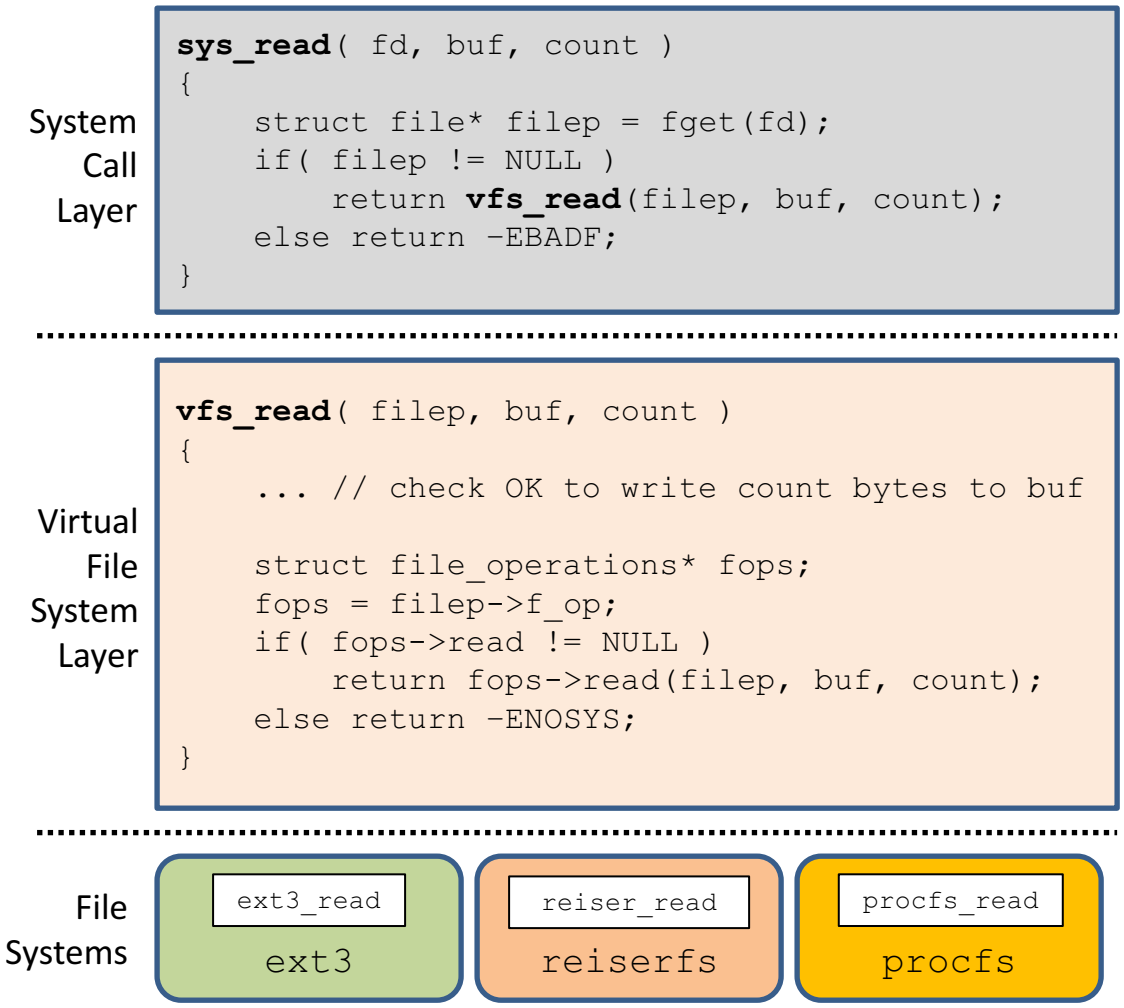

Figure 5.10 File Operation Processing in Linux

A `read` by a user level application results in the execution of the `sys_read` system call. `sys_read` looks up the `struct file` for the given file descriptor and then calls the Virtual File System function `vfs_read`. `vfs_read` checks that the file system for the target file supports the `read` operation, and then calls the appropriate function within the file system.

the target file. This pointer is then passed along with the other system call operands to a VFS layer function for the current operation (e.g., `vfs_read` for a `read` operation). The VFS layer function then examines the contents of a `file_operations` structure that is referenced by the `struct file`. The `file_operations` structure is used to indicate which operations a specific file system implements, and contains function pointers for each file operation supported by the VFS. When initially loaded, a

file system registers an instance of this structure that contains the addresses of the functions that it implements and null pointers for unimplemented functions. All files have their `struct file` initialized to point to the `file_operations` structure for the file system in which they reside. If the VFS layer function finds a valid function pointer for the current operation, it calls that function with the operands supplied to the system call. If the target file system provides no implementation, then either a generic VFS implementation of the operation is used or an error indicating the operation is not supported is returned .

Our design adds two new fields to `struct file` to support group file operations. These fields are present only when the kernel has been configured to support group file operations. The first field is a pointer named `group_data` that references a newly defined `struct group_file`. This new structure records information about a group file including the access flags, group size, and group directory path. This structure also contains fields for three arrays that store the name, last operation status code, and `struct file` pointer for each group member. During `gopen`, the `group_data` field is set to an instance of `struct group_file` that has been initialized with the appropriate group information. The second field is named `gf_op` and points to a newly defined `struct group_file_operations`. The `group_file_operations` structure contains function pointers for both group-aware versions of the standard file operations and the new functions used during group file operations. File systems directly supporting group file operations register an instance of this structure when loaded.

To support group file operations even when the underlying file system does not provide an implementation, our design also extends the VFS layer function for each operation. These extensions simply iterate to call the single file operation and record the status result for each member. Group status and data results are generated using the default status aggregations from Table 3.3. Custom status and data aggregations are not supported, but can easily be implemented by the client at the user-level.

Figure 5.11 shows how single and group file operations are processed in our modified Linux VFS.

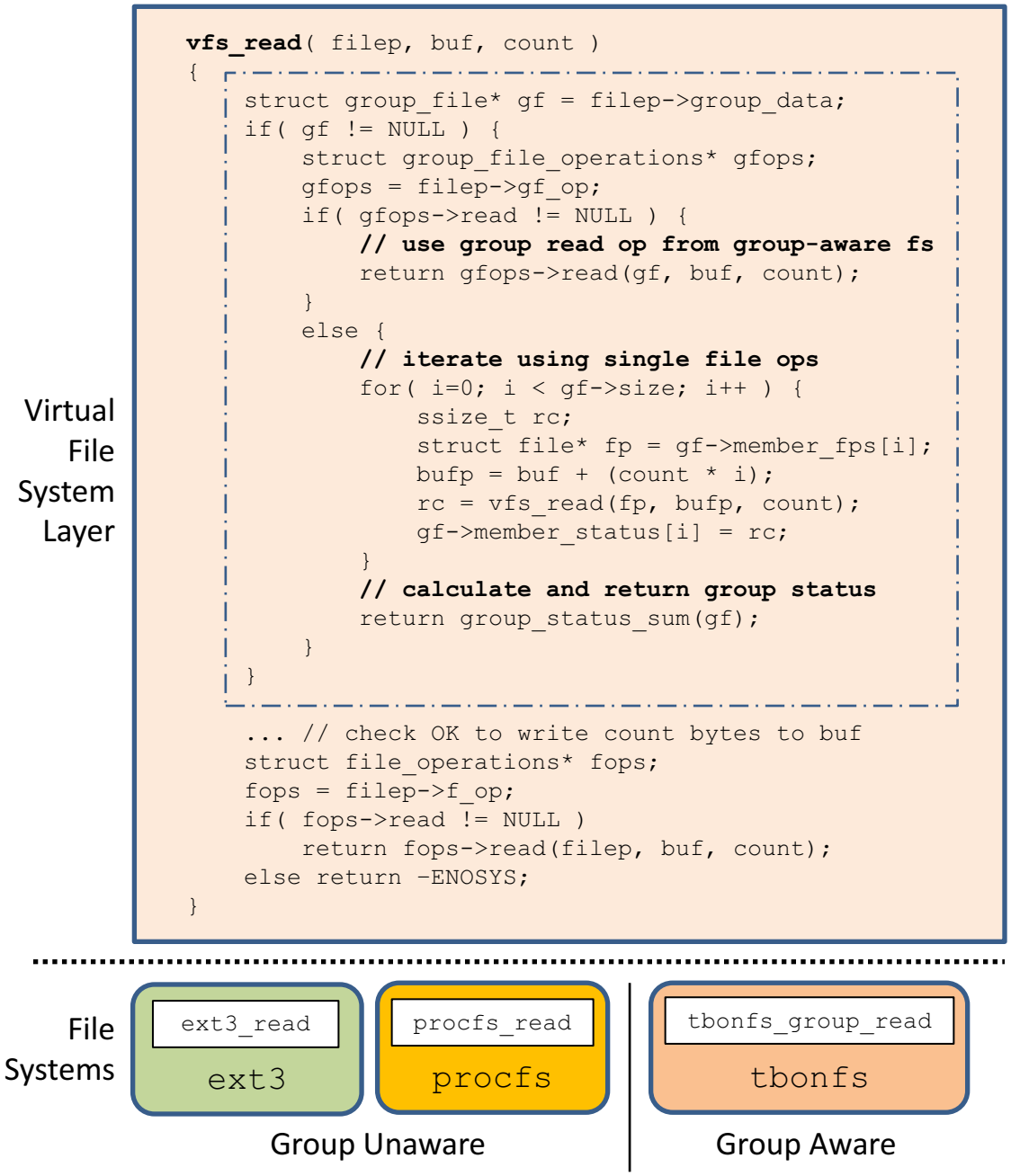

Figure 5.11 Group File Operation Processing in Linux

The Linux VFS layer was extended as shown to support group file operations. Code in the dashed box is newly added for group `read` operations; similar code was added to support other group file operations within their corresponding VFS layer functions. Operations on group files are detected by the presence of a non-null `group_data` field in the `struct file` for the target. When the underlying file system is *group aware*, it provides an implementation of the group file operation that is used by the VFS layer function. Group operations on files in *group unaware* file systems are supported by iterating over group members.

After receiving the `struct file` pointer from the system call, the VFS layer function for the current operation checks to see if the target is a group file by testing if `group_data` is null. If the file is not a group file, the VFS layer function proceeds to execute the existing single file code as originally shown in Figure 5.10. If a group file is identified, the `group_file_operations` structure referenced by the `gf_op` field is consulted. If the current group operation is supported by the underlying file system, as indicated by a valid function pointer, then that function is called and passed the `struct group_file` and system call operands. Otherwise, the VFS layer function uses the general support provided by our extensions that iterate over group members.

To test our second hypothesis that kernel-level group file operations would improve performance versus user-level iteration, we measured the performance of a prototype implementation of this design when using group file operations on local files. Our implementation was based on a stock Linux 2.6.27.5 kernel [61], and provided support for group `read`, `write`, `lseek`, and `close` operations.

Performance benchmarking was done using two versions of a simple application that performed group operations on 1,024 files residing within a directory on an ext2 file system. The first version of the application used a traditional loop structure to operate on group members. Within each loop iteration, a file was opened, a `read` operation was issued, and the file was closed. The second version used group file operations, and consisted of a single `gopen`, a group `read`, and a group `close`. The total walltime to execute all operations on the group was measured using `gettimeofday`. The host system for our experiments had a dual-core AMD Opteron 1214 running at 2.2GHz and 3GB of DDR2 RAM.

Unexpectedly, the performance of the application using group file operations was roughly 10% worse than the looping application. To identify the source of the slowdown, we instrumented our modified kernel to collect the time spent in various activities during `gopen` and group file operations. Our investigation revealed that the overhead of intializing group state during `gopen` was significant, and was the overriding factor in the slowdown. Figure 5.12 presents a performance model and the mea-

Assume the following model parameters:

- N : number of files in a group
- T : latency of trapping into and out of the kernel during a system call
- S_o : latency of an `open` on a single file
- S_r : latency of a `read` on a single file, including data access and copy to user space
- S_c : latency of a `close` on a single file
- G_o : latency of managing state for a single group member during `gopen`
- G_r : latency of managing state for a single group member during a group `read`
- G_c : latency of managing state for a single group member during a group `close`

For user-level iteration, the total latency for using `open`, `read`, and `close` on a group of N files is approximately:

$$USER \cong 3TN + N(S_o + S_r + S_c)$$

For kernel-level iteration, the total latency for using `gopen`, `read`, and `close` on a group of N files is approximately:

$$KERNEL \cong 3T + N(S_o + S_r + S_c) + N(G_o + G_r + G_c)$$

For kernel-level iteration to outperform user-level iteration, we have:

$$KERNEL \leq USER$$
$$3T + N(S_o + S_r + S_c) + N(G_o + G_r + G_c) \leq 3TN + N(S_o + S_r + S_c)$$
$$3T + N(G_o + G_r + G_c) \leq 3TN$$
$$N(G_o + G_r + G_c) \leq 3TN - 3T$$

Since we assume $N >> 1$, $3TN - 3T$ approximates $3TN$, and thus we have:

$$N(G_o + G_r + G_c) \leq 3TN$$
$$(G_o + G_r + G_c) \leq 3T$$

We observed the following values for our implementation:

$$G_o \cong 3000 \text{ ns}, \; G_r \cong 10 \text{ ns}, \; G_c \cong 10 \text{ ns}$$
$$T \cong 90 \text{ ns}$$

Using these values, we can see that 3020 ns is not less than 270 ns. However, our model is not dependent on a constant number of system calls. If we substitute a variable number of system calls C for 3 in our equations, and assume all system calls except `gopen` require approximatley the same time $G_k = 10$ ns for managing group state, then we have:

$$G_o + CG_k \leq CT$$
$$(G_o \div C) + G_k \leq T$$
$$(G_o \div C) \leq T - G_k$$
$$(G_o \div (T - G_k)) \leq C$$

Finally, substituting the observed values again and solving for C:

$$(3000 \div (90 - 10)) \leq C$$
$$37.5 \leq C$$

Thus, at least 38 system calls on the same group are necessary to overcome the `gopen` overhead.

Figure 5.12 User-level vs. Kernel-level Group File Operations: A Performance Model

sured activity values for a sequence of group file operations. This model shows that the seqence of operations must be relatively long, on the order of 40 operations, to overcome the performance degradation due to management of group state within the kernel implementation. Unfortunately, none of the potential use cases we considered for group file operations on local files, such as accessing proc files to support `ps` and `top`, would require more than a few operations per group.

5.7 Summary

TBON-FS is designed as a scalable, flexible, and portable group file system. TBON-FS leverages a tree-based overlay network to provide both scalable definition of file groups using global name space composition and scalable operations on file groups. The current implementation has been shown effective for use on distributed systems containing tens of thousands of hosts, and design decisions were made with systems containing millions of hosts in mind. By adopting a purely user-level design for client and servers, TBON-FS improves its portability and avoids the problems of deploying system-level modifications. TBON-FS gives clients control over the composition of its global name space to ensure that target file groups are created efficiently. Using synthetic file services loaded into TBON-FS servers, clients can further extend their use of scalable group file operations to non-file resources.

The current TBON-FS design also has some limitations related to flexibility in merging name spaces, fault tolerance, and its reliance on a single user mode of execution. We discuss these limitations and our plans for addressing them in future work.

In our study of global name space composition, we have focused on a single type of merge composition that places distributed files that have the same path into a single directory in the global name space. This composition is key to creating the directories that define the membership of group files. Our implementation of global name space composition and path resolution within TBON-FS (described in Section 5.2) reflects this limited focus. To fully support the customizable merge semantics provided by FINAL, this implementation requires a few extensions. First, our merge aggregation

must be modified to use a user-provided function for conflict resolution. This is a simple change that is easily supported by our current TBON infrastructure. Second, since custom conflict resolution functions may create new synthetic directories at arbitrary paths within the global name space, our merge aggregation must keep track of these new directories so that subsequent path resolution requests are properly satisfied. This tracking will likely require caching portions of the composed global name space in the aggregation state at TBON processes. Third, once name space caching is in place, we must study and implement cache invalidation policies that can handle dynamic name spaces whose contents can change over time. A common example of a dynamic name space is the name space provided by a proc file system, which has entries corresponding to processes that are created and removed as processes begin and end execution.

A design for fault tolerance should accompany any scalable system design, since faults occur with increasing frequency as the size of a system is scaled up. In the current design of TBON-FS, we have not explicitly addressed fault tolerance. TBON-FS relies on the fault tolerance capabilities of MRNet to provide automatic TBON reconfiguration after failures of internal tree processes. Still, TBON-FS server process (or host) failures are expected to be common on next generation systems due to their scale. Thus, a thorough examination of the survivability of TBON-FS in the presence of server failures and server restart strategies are needed. MRNet also provides mechanisms for recovering aggregation data cached at processes that fail. Since TBON-FS does not currently cache data within the TBON, we have not used these data recovery mechanisms. Future aggregations that do employ caching will require appropriate use of these mechanisms.

TBON-FS executes all processes within a single user context, which simplifies its design and enables it to use existing distributed system authentication and access control. Still, this single user mode of operation limits the use of TBON-FS in distributed system services such as job launching that require elevated privileges to start processes for multiple users. Although TBON-FS can be run in the

context of a system administrator, no mechanisms exist in TBON-FS for ensuring user isolation. Thus, system services that use TBON-FS would need to provide the necessary security mechanisms. We plan to investigate the changes required to TBON-FS when used in such system services. The single user approach is also incompatible with use cases that span administrative domains with disjoint user identities, such as arises in the use of computational grid systems and commercial clouds. To support multi-domain use, TBON-FS requires a method for clients to specify credentials that should be used for TBON processes running in each domain, and associated mechanisms that use these credentials when launching processes and accessing file systems.

Chapter 6

Control and Inspection of Process and Thread Groups

The previous three chapters describe the group file operation idiom, the use of name space composition for efficiently creating directories used to define group files, and the TBON-FS group file system that provides scalable group file operations and global name space composition. Together, these three pieces provide a general framework that tools and middleware can use for scalable operations on distributed files. In this chapter, we describe a fourth piece that maps operations on processes and threads to file operations. When combined, these four pieces provide a scalable framework for control and inspection of distributed process and thread groups.

For over a quarter-century, operating systems have provided synthetic file systems for control and inspection of processes [40, 58, 93]. Abstracting processes as files allows tools to locate and perform control and inspection operations on processes using standard file system operations, and thus avoids creating a litany of special-purpose `ioctl` or system calls for exchanging control commands or process data between user- and kernel-space. Existing utilities that read or write files can be used for control and inspection, such as using `cat` on Linux hosts to view the process status information available in `/proc/`*`pid`*`/status`. The proc file system, or procfs for short, was introduced with 8th-edition

UNIX [58] and refined by Plan 9 [93]. Current operating systems that provide a procfs use a layout similar to that pioneered in Plan 9. In the Plan 9 procfs, a directory exists for each process running on the host. Files that can be used for control and inspection of a specific process are located within these per-process directories. In some implementations, such as in Solaris and Linux, the lightweight processes used to support user threads are also included as directories containing files for control and inspection. Lightweight process directories exist as sub-directories of their associated process.

Given that existing proc file systems already provide file-based interfaces for control and inspection of processes and threads, it is easy to assume we can simply use group file operations on these existing files to achieve our goals for scalable control and inspection. Unfortunately, the abstractions provided by existing proc file systems can limit or even prevent the use of scalable group file operations. In response to these problems, we have designed a new synthetic file service named proc++. The primary challenges in designing proc++ were to identify the tool behaviors that were hindered by interactions with existing procfs abstractions, and to develop new abstractions that enable efficient and widespread use of group file operations. The difficulty of the latter challenge should not be underestimated, as the placement of the abstractions within the layers of the distributed tool hierarchy is key to enabling the parallelism necessary to scale, and choosing the proper amount of tool functionality captured by the abstractions requires careful thought to avoid limiting their utility.

Existing procfs abstractions correspond to OS and hardware state for each process or thread. Thus, procfs operations are typically primitives for reading or updating this state, such as stopping or continuing the execution of processes and threads, and reading or writing memory and registers. Since procfs operations involve access to low-level state, the operands to these operations often correspond to state values such as specific virtual memory addresses. A requirement for using group file operations is that the same operand values must be used for all group members. Unfortunately, operand values are often based on context specific to a process or thread, and may vary across targets. For

instance, parallel debuggers often map a code source line to a memory address. Due to address space randomization employed by many operating system program loaders [12], the program's code may be mapped at different base locations in the address space. Similarly, operating systems may randomize the base address of a thread's stack or heap, resulting in varied locations for program data and variables. Varied addresses prevent the use of simple group file operation sequences such as performing a group seek to a common memory address before doing a group read or write. To overcome context variations, a tool may identify and operate on sub-groups of members sharing the same operand values. Although custom group file operation aggregation may be useful in identifying sub-groups, the overhead necessary to deal with large amounts of variation can be substantial. In the worst case, each sub-group will contain exactly one member, and the tool is forced to use non-scalable iteration.

When implementing tool-level functionality such as breakpoint management and gathering stack traces, the use of low-level procfs abstractions results in possibly long sequences of interactions with procfs. For example, to insert a breakpoint in a group of processes, a debugger will stop the processes that were running, read and write the process address spaces, and finally continue processes if necessary. We refer to such tool activities as interaction-intensive operations. Even when group file operations are used to apply each procfs operation, the tool activity incurs the latency of several rounds of network communication. Further, some tool activities are implemented as sequences that include feedback, where the results of one operation are used to determine the next operation. A common example of an activity involving feedback is gathering thread stack traces, where the next process address to read is dependent upon previously read values. When the values used for feedback decisions vary due to context, the tool again must choose between handling the variation by keeping track of sub-groups or falling back to iteration.

In this chapter we describe *proc++*, a new synthetic file service providing control and inspection of process and thread groups. proc++ addresses the problems that hinder the efficient scalable execu-

tion of group file operations on existing proc file systems by exporting abstractions designed to eliminate context variations and encapsulate interaction-intensive activities. Section 6.1 discusses the design and implementation of proc++. When combined with TBON-FS and group file operations, proc++ provides a scalable debugging platform. In Section 6.2, we show that group file operations on distributed proc++ groups provide scalable performance suitable for interactive debugging of the largest of applications, currently demonstrated on applications with over 200,000 processes.

6.1 proc++ Design

proc++ provides functionality similar to Solaris procfs, which is widely regarded to provide the most complete set of abstractions among existing implementations. It provides abstractions for processes, lightweight processes, events, address spaces and associated mappings, hardware register sets, open files, signals, exceptions, system calls, and watchpoints. Together, these abstractions give tools the greatest flexibility to achieve the desired control and inspection actions. Like the Solaris procfs, proc++ files use binary data for the structured commands written to control files and for data returned from reading inspection files. The proc++ organization of process directories with thread subdirectories, and the names for various files, are also modeled after Solaris (whose interface was also adopted by AIX and IRIX). There are, however, significant departures from the Solaris design to support our new abstractions (Section 6.1.1), and to aid in composition of a global proc++ name space whose layout is tailored for easy use with group file operations (Section 6.1.2).

Tables 6.1 and 6.2 list the proc++ files found in process and thread directories, respectively. Each table entry shows the path of the file within the process/thread directory, the supported access modes, and the functionality the file provides. Files that support write access are used for control, and those supporting read access are used for inspection. Many of these files will be familiar to users of Solaris procfs. We have intentionally used the same file names when the files can be expected to provide similar functionality to Solaris.

FILE	RD	WR	FUNCTIONALITY
`as/map`	y	n	A map of the regions (e.g., code, data, stack, heap) of the address space and their backing files if applicable.
`as/mem`	y	y	Read/write access to the address space.
`as/image/`*`imagename`*	y	y	Read/write access to the portion of `as/mem` where the named executable or library image has been mapped.
`ctl`	n	y	Issue process control operations such as *stop*, *continue*, or *signal*. Insert or remove breakpoints.
`events`	y	n	A queue of thread events that have stopped the process.
`status`	y	n	General process status information.
`threads/`	-	-	Directory containing thread directories.

Table 6.1 proc++ Process Directory Contents

FILE	RD	WR	FUNCTIONALITY
`ctl`	n	y	Issue thread control operations such as *stop* or *continue*, or enable/disable singlestep mode.
`event`	y	n	The last event that occured on the thread.
`reg/fpr`	y	y	Read/write access to the floating point registers.
`reg/gpr`	y	y	Read/write access to the general purpose registers.
`reg/pc`	y	y	Read/write access to the program counter register.
`reg/sys`	y	y	Read/write access to the system-specific registers.
`stacktrace`	y	n	The current stack trace for a stopped thread.
`status`	y	n	General thread status information.

Table 6.2 proc++ Thread Directory Contents

6.1.1 proc++ Abstractions

proc++ provides several abstractions not previously found in existing procfs implementations. These abstractions are designed to solve the two problems faced when using group file operations on files from existing proc file systems. First, they are context insensitive to reduce variations due to process- and thread-specific context, which enables group file operations to use the same operand values for all group members. Second, they encapsulate sequences of procfs operations that are commonly

used for tool activities. Our design attempts to provide abstractions that are flexible to enable their use as primitives for building higher-level tool functionality. Further, we strive to avoid excessively increasing the cost, in terms of computation and memory use, of performing control and inspection operations on target hosts (viz., hosts containing target processes and threads).

The primary example of both varied context and interaction-intensive tool activities involves accessing memory locations that correspond to code or data. Often, the tools accessing memory rely on auxiliary information sources to generate the target addresses. For example, symbol tables are used to locate specific functions or global data variables, and debug information encoded in executables during compilation can be used to determine the locations of function local variables or members within data structures. These information sources often represent memory addresses as expressions that can be used to compute the target address at runtime by filling in observed values for expression variables such as the base address where a code section has been mapped or the value of a hardware register. Such expressions are thus useful for handling varied context.

proc++ provides an abstraction for the address space of a process that uses a simple *base+offset* expression. The address space is considered as a collection of images. We define an image as a contiguous region of the address space that contains the code and data of a program executable or a shared library. proc++ exposes one file for each executable and library image. The zero offset for each image file corresponds to the base address of its address space mapping. Tools can use group file operations to seek to and read or write code or data located at known offsets within the image file. For example, to read the value of a program global variable across a group of processes, the tool first determines the offset for the variable by examining the program symbol table. The tool then uses a group seek to that offset within the image file for the program executable, followed by a group read to gather and possibly aggregate the values.

Although not yet supported in proc++, the base+offset scheme for memory addresses can be

extended to support additional types of bases such as the values of specific registers or the base address for the process stack or heap. These extensions would be useful to provide context insensitive access to data structure members and stack or heap variables.

proc++ also introduces two new abstractions, breakpoints and stack traces, that encapsulate interaction-intensive tool activities. These abstractions move decision logic typically used by the tool front-end to the proc++ service on each target host. Although this move results in slightly increased memory use by the service, it greatly simplifies the implementation of tool activities and ensures that the tool will not become a bottleneck due to the increased processing necessary to handle sequences of operations and potentially varied context.

In proc++, breakpoints are attributes of a process address space. Tools can explicitly create and remove breakpoints, which results in the appropriate actions to safely update the process address space. Breakpoint locations are specified by providing the name of the image containing the location and a set of associated offsets. A set of offsets is used to handle cases where a source code line maps to multiple instruction addresses. The tool also provides a unique identifier that names the breakpoint. Event records generated by proc++ indicate the active breakpoint by including the provided identifier.

The stack trace abstraction introduced by proc++ uses a new `stacktrace` file present in the directory for each thread. When read, this file provides a stack trace as a list of frames. Each frame is a 3-tuple of addresses representing the frame base address, the frame stack pointer, and the frame return address. By directly providing stack traces, proc++ eliminates the need for tools to use the typical long sequence of address space and register reads. The typical method can be extremely costly when gathering stack traces for groups of distributed threads, due to the added latency of distributed communication and the wide variability in the memory addresses that need to be read, which reduces the ability of tools to employ scalable group file operations for those memory inspection operations. Another benefit of providing stack traces is that it enables tools to use scalable aggregations, such as the call-graph tree

merging used by STAT [5], to process the traces during group read operations.

The last abstraction introduced by proc++ involves access to hardware registers. proc++ provides four separate files representing four common register classes: general-purpose registers (GPR), floating-point registers (FPR), system-specific registers, and the program counter (PC) register. Using these files, tools can obtain exactly the set of register values they need, rather than being forced to read the contents from all registers as is the case for Solaris procfs. Separate files also enable the use of custom aggregations for each class of register, which could be useful for many data analysis tasks. For example, the calculations necessary to identify register value skew across distributed processes are quite different for integer versus floating-point data. Having a separate file for just the PC is an optimization for the common debugging task of finding the addresses of stopped threads, and permits the use of efficient clustering of equal PC values.

6.1.2 proc++ Global Name Space

The proc++ name space is organized to work in conjunction with the global name space composition used by TBON-FS. The goal is to produce a global proc++ name space that is tailored for use with group file operations; this is accomplished by ensuring that tools never need to manually create and populate the group directories used to define group files. To meet this goal, proc++ uses two strategies. First, it places files that are expected to be operated upon as a group at the same path on all hosts, so that TBON-FS will generate synthetic group directories containing these files. Second, when a tool creates a new process or thread group, proc++ automatically generates group directories corresponding to each of the files found in a process/thread directory, and populates those directories with symbolic links the the associated file for each group member. The links have names that encode the host, process, and thread (if applicable) of the target member file. Tools provide a name for each new group that is used to ensure that the group directories exist at the same path across all hosts. Because TBON-FS merges the contents of directories located at shared paths, all the members from every host are placed

in the same directory within the global proc++ name space.

Unlike previous procfs implementations, proc++ exposes host information within its name space. Directories for processes executing on a given host are placed within a host-specific sub-tree, as shown in Figure 6.3. The use of host-specific sub-trees has two benefits when composing a global proc++ name space. First, since processes are isolated by host, name space conflicts due to distributed processes that share the same process ID are avoided. Second, tools using the global name space can easily identify the subset of distributed processes running on a specific host, and can quickly locate files for host-specific processes or threads.

proc++ provides a session abstraction that is used to manage a set of processes that are being controlled. A tool front-end specifies exactly the set of processes it needs to operate upon, such as the processes of a particular parallel application. Processes are added to a session by writing messages to the session's `ctl` file with instructions to either launch a new process or attach to a set of existing processes. Newly added processes are then visible in the proc++ name space. Once finished operating on processes, a tool can detach them from the session. A tool may employ multiple sessions to manage groups of processes corresponding to different programs, such as may be found in multiple-program multiple-data (MPMD) parallel applications.

Sessions also serve as the container for defining process or thread groups, which are given first-class status in proc++. Each session contains predefined groups representing all processes and all threads; processes and threads are automatically added to these predefined groups when they are added to the session. Users can create custom, named groups and manage their membership using structured messages written to a session's `ctl` file. Group membership is specified as a list of 3-tuples; each tuple identifies a host, process, and optional thread. As shown in Figure 6.4, each process or thread group exists as a sub-directory of a session's `groups` directory. Each process or thread group directory contains the set of sub-directories that proc++ automatically generates for each file.

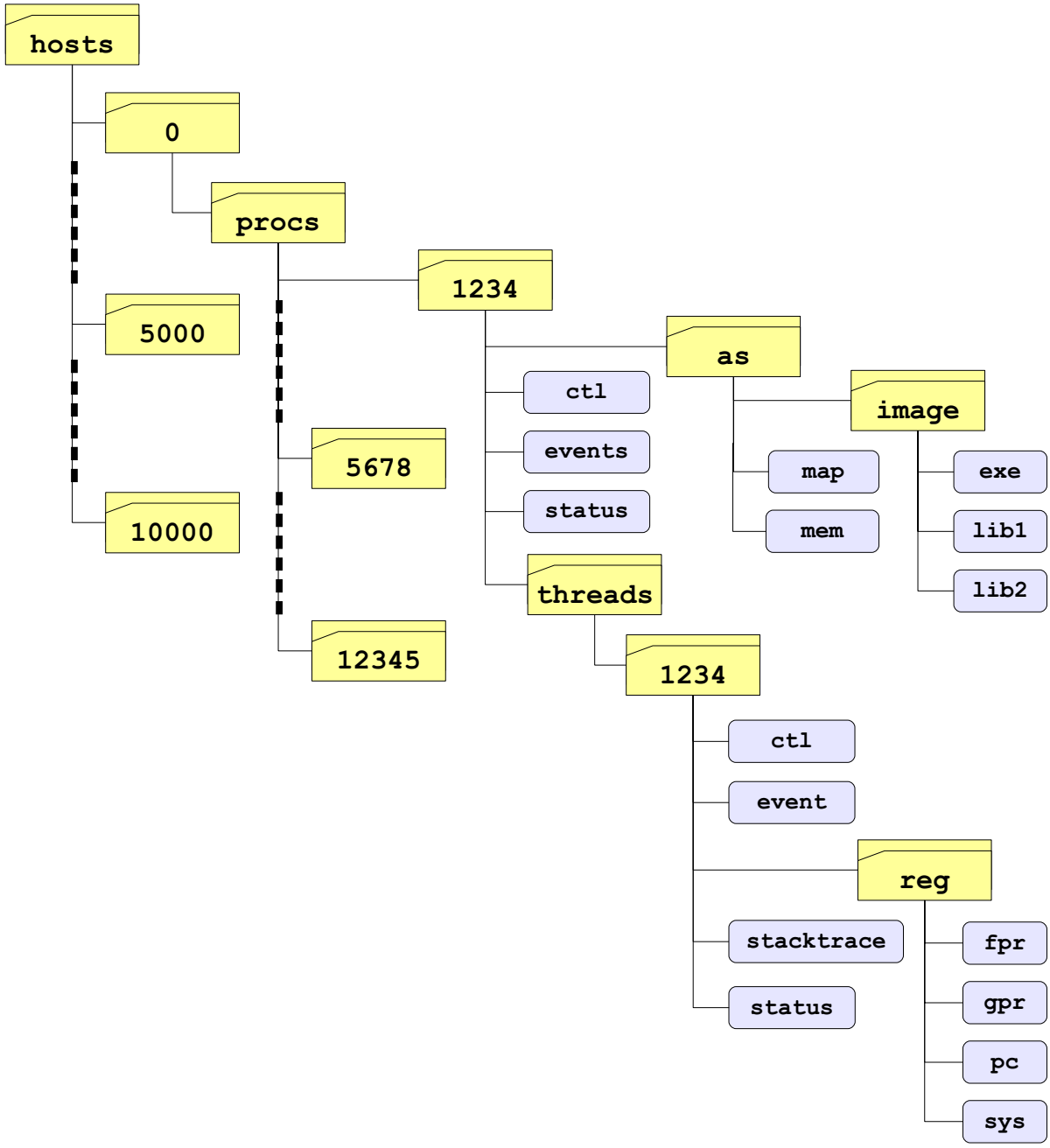

Figure 6.3 proc++ Host Name Space.
To prevent name space conflicts for processes on different hosts having the same process id, proc++ places process directories within host-specific sub-trees

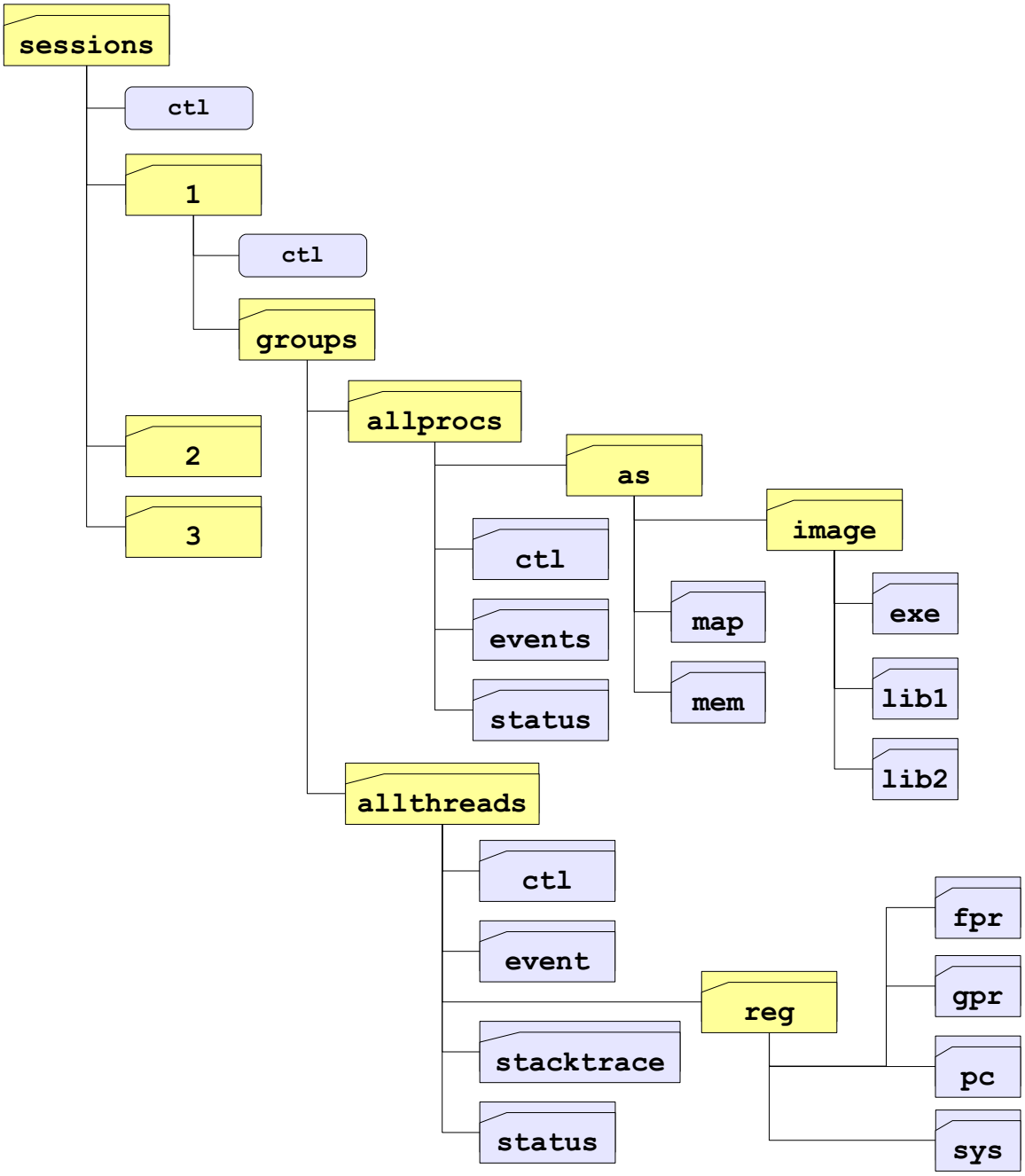

Figure 6.4 proc++ Session Name Space.
Sessions are containers for the pre-defined groups "`allprocs`" and "`allthreads`", as well as user-defined groups. Process and thread group directories contain sub-directories for each control or inspection file. These sub-directories contain symbolic links to the relevant files for each group member.

6.1.3 proc++ Implementation

proc++ is implemented as a FINAL file service that can be plugged into TBON-FS servers. The functionality provided by proc++ files is built upon ProcControlAPI [85], a cross-platform library for control and inspection of processes and threads. Generation of thread stack traces uses StackWalker-API [86], a cross-platform library for walking function call stacks.

ProcControlAPI provides platform-independent interface classes for processes, threads, and events. ProcControlAPI supports common procfs control and inspection operations such as stopping and continuing processes, reading and writing process memory or thread registers, and reporting stop events. Additionally, ProcControlAPI provides advanced capabilities for detecting system events such as process `fork` and `exec`, thread creation and deletion, and shared library load and unload. Event notifications are delivered using a callback system that allows users to register handler functions for any subset of the supported event types.

For many proc++ control and inspection operations, ProcControlAPI already supported the required functionality. However, we needed to extend ProcControlAPI's capabilitites to support break-point insertion/removal and thread stepping. We chose to extend ProcControlAPI to provide cross-platform breakpoint management and stepping, rather than complicate the proc++ internal design by adding platform-specific code. A deficiency of ProcControlAPI is that it does not provide information on the address space mappings for the code and data sections of a program's executable and dependent libraries. proc++ currently gathers this mapping information directly from the operating system.

6.2 Evaluation

During the development of the proc++ file service, we also constructed *tbon-dbg*, a command-line parallel debugger that serves as a test harness for using group file operations and TBON-FS with proc++. Although initially designed for feature testing, we were able to extend its capabilities for use in large-scale performance evaluations. The capabilities of tbon-dbg are described next, followed our

evaluation of the performance of group file operations on distributed proc++ files.

6.2.1 tbon-dbg Parallel Debugger

The command-line interface (CLI) of tbon-dbg is built using the GNU readline library [96], which provides advanced line editing and command history for CLI programs. Similar to a conventional command-line debugger, users enter commands at the tbon-dbg prompt, and any output is printed before the next prompt is displayed. tbon-dbg also supports sending command output to a Unix pipe or file, a feature that has proved useful for validating large output.

tbon-dbg supports multiple concurrent sessions and many process and thread groups per session. All debugging commands are associated with either the focus session, or the focus group. Users may change the focus session or focus group at any time. Table 6.5 presents the commands supported for session management, and Table 6.6 shows the commands available for group managment and focus group operations. Absent from tbon-dbg's command set are single-target debugging operations. To test single-target debugging operations, a user must define a singleton group and use group operations.

In addition to its debugging capabilities, tbon-dbg also gives users the ability to navigate the proc++ global name space. Similar to a conventional shell, users can issue `cd`, `ls`, and `pwd` commands to explore the name space. Although not necessary for parallel debugging tasks, this navigation capability has proven useful for debugging problems in the name space composition of TBON-FS.

Three new capabilities were added to tbon-dbg so that it could be used for large-scale performance evaluation of group file operations on distributed proc++ files. First, we added support for scripted execution, which is necessary when experiments must run in non-interactive environments such as the batch job systems used by large HPC systems. Second, we added the ability to launch a parallel application using the HPC system's parallel runtime. Finally, we instrumented the command processing engine to collect timing data and generate a log file that reports the latency for each command issued.

COMMAND	COMMAND DESCRIPTION
`session [id]`	Change the focus session to `id`, when provided. Otherwise, display the id of the focus session.
`session create id`	Create a new session named `id`.
`session remove id`	Remove the session named `id`.
`attach host:pid[,...]`	Attach list of host processes to the focus session.
`detach host:pid[,...]`	Detach list of host processes from the focus session.
`kill host:pid[,...]`	Kill list of host processes, and remove them from the focus session.
`attachcmd exename`	Attach to all processes on all hosts that are executing `exename` to the focus session.
`detachcmd exename`	Detach all processes in the focus sesssion that are executing `exename`.
`killcmd exename`	Kill all processes in the focus sesssion that are executing `exename`.

Table 6.5 tbon-dbg Session Management Commands

6.2.2 proc++ Evaluation

Our goal for evaluating proc++ is to show that it can be used as the basis for building scalable tools that require control or inspection of distributed process and thread groups. To this end, our experiments used tbon-dbg to perform a set of common debugging tasks on a parallel application as we increased the number of application processes. These tasks include attaching to the application's processes, issuing group control commands such as *stop*, *continue*, and *step*, inserting and removing breakpoints, reading register values, and processing events.

During each tbon-dbg run, we measured the latency of operations at two levels of detail. The first was the latency of individual group file operations, as measured within the TBON-FS client library. The second level was tbon-dbg command latency as reported in its performance log. A single tbon-dbg command typically involves many operations to `gopen` the target group, bind any needed aggregations, perform a sequence of group file operations, and close the group.

Experiments were run on the JaguarPF Cray XT5 system [77] located at Oak Ridge National Laboratory. JaguarPF has 18,688 compute nodes. Each node contains two six-core AMD Opteron 2435

COMMAND	COMMAND DESCRIPTION
`group [id]`	Change the focus group to `id`, when provided. Otherwise, display the ID of the focus group.
`group create id type`	Create a new group within the focus session named `id`. `type` is used to indicate a process or thread group; valid values are “proc” or “thr”.
`group remove id`	Remove the group named `id` from the focus session.
`group add host:pid[,...]`	Add list of host processes to the focus group.
`group add host:pid:tid[,...]`	Add list of host threads to the focus group.
`group del host:pid[,...]`	Remove list of host processes from the focus group.
`group del host:pid:tid[,...]`	Remove list of host threads from the focus group.
`group ctl cont|stop`	Continue or stop the focus group, which should be a process group.
`group ctl step|nostep`	Enable or disable single-step mode. The focus group should be a thread group.
`group events`	Display all events that have occured on the focus group since the last time the command was issued. The focus group should be a process group.
`group map`	Display the address space mappings for the focus group, which should be a process group.
`group readmem addr count`	Read `count` bytes at memory address `addr` in the focus group, which should be a process group.
`group readimg image offset count`	Read `count` bytes at `offset` within `image`. The focus group should be a process group.
`group break image offset`	Insert a breakpoint at `offset` within `image`. The focus group should be a process group.
`group clearbreak image offset`	Clear a breakpoint at `offset` within `image`. The focus group should be a process group.
`group readreg gpr|pc`	Display the general-purpose or program counter register values. The focus group should be a thread group.

Table 6.6 tbon-dbg Group Management and Operation Commands

Number of Application Processes	Number of Servers	TBON Topology (fan-out per tree level)
1,728	144	8 × 18
4,096	342	18 × 19
8,000	667	23 × 29
13,824	1,152	32 × 36
39,304	3,276	7 × 13 × 36
54,782	4,590	9 × 17 × 30
74,088	6,174	7 × 21 × 42
97,336	8,112	8 × 26 × 39
125,000	10,440	10 × 29 × 36
157,464	13,122	18 × 27 × 27
195,112	16,280	20 × 22 × 37
216,000	18,000	20 × 25 × 36

Table 6.7 TBON Topologies used in tbon-dbg Experiments

processors and 16GB of DDR2-800 memory, for a total of 224,256 processing cores. Nodes are connected in a three-dimensional torus topology by a high-bandwidth, low-latency SeaStar 2+ network.

The TBON topologies used for each experimental scale are shown in Table 6.7. One TBON-FS server was placed on each target node. Reported measurments are the minimum latency across multiple runs at each scale. We report minimum rather than average latency due to frequent interference from other applications using the JaguarPF network that results in random spikes in observed latencies.

The target application we used is the implicit radiation solver (IRS), a general diffusion equation solver applied to radiation transport. IRS is one of the applications from the ASC Sequoia Benchmark Codes [8]. Although IRS can use both MPI and OpenMP, IRS was configured to use only MPI. One MPI process was placed on each of the twelve cores per compute node. IRS requires the total number of processes to be equal to k^3 for an integer $k \geq 2$.

Figure 6.8 shows the performance of group file operations for runs with up to 216,000 application processes (18,000 nodes). The top graph (a) reports the time necessary for operations used to manage groups, including opening and closing a group, and binding an aggregaton for use with group read operations. The middle graph (b) shows the time required to perform group write operations with various data sizes. The bottom graph (c) presents group read times for various data sizes and aggregations.

All three group management operations in Figure 6.8(a) exhibit good performance up to the scale tested. The varied performance between the three operations is attributable to the amount of data broadcast and gathered. The nearly flat line for group `close` shows the performance that can be achieved when the data sent is minimal and the reply uses a simple summary aggregation. The time for `gbindaggr` includes creation of the new aggregation stream and a simple request-reply that is similar to that used by group `close`. As is evident from comparison to the `close` line, stream creation adds a linear component to the total latency. Similar to `gbindaggr`, the time for `gopen` includes a new request stream creation, as well as additional linear processing on the TBON-FS client. Despite their linear components, the slopes for both `gopen` and `gbindaggr` are small enough that the operations are projected to provide sub-second latencies for applications containing over one million processes.

The group `write` operations of Figure 6.8(b) exhibit performance similar to that of `gbindaggr`. For the small sizes (less than 100 bytes) of proc++ control messages, group `write` operations show little deviation. Again, the linear behavior is due to creation of a stream for aggregating operation status codes. In practice, group `write` operations will only incur this linear cost for the first `write` on the group, as subsequent operations will use the same status aggregation stream. Subsequent operations can be expected to provide excellent scaling behavior similar to the group `close`.

The choice of data aggregation for group `read` operations has a large impact on performance, as shown in Figure 6.8(c). Group `read` operations that use the default concatenation aggregation perform an order of magnitude worse than those that use scalable equivalence aggregations. The benefits

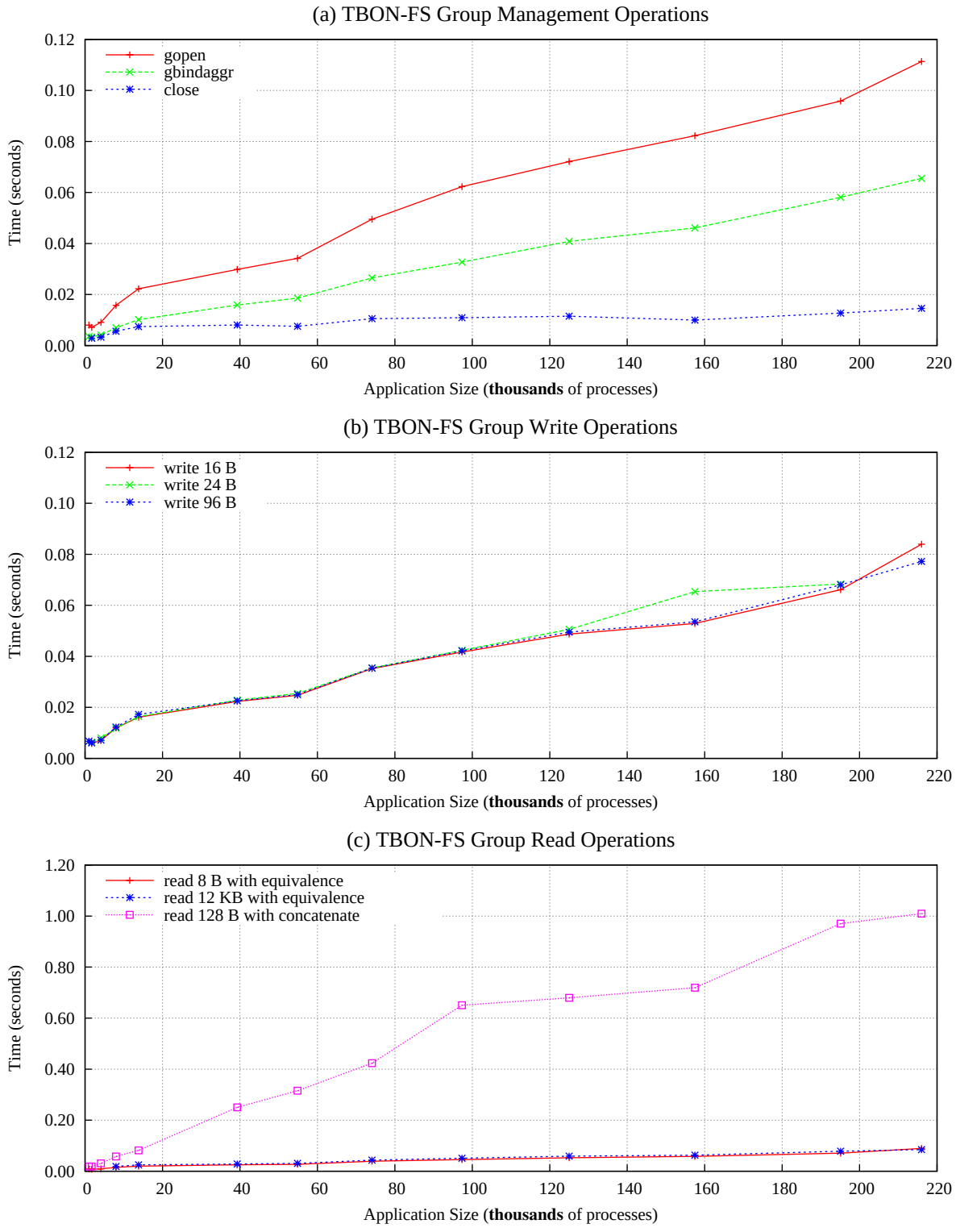

Figure 6.8 Group File Operations on Distributed proc++ Files
Time required for various operations for group management, writing to control files, and reading inspection files.

of equivalence aggregation are substantial even for large `read` counts. For instance, at 200,000 processes, a group `read` of 128 bytes with concatenation takes one second, while a group `read` of 12 kilobytes that uses an equivalence aggregation finishes in one-tenth of a second.

Figure 6.9 presents the performance of tbon-dbg's group control operations and group inspection operations. Each tbon-dbg group operation requires at least three group file operations: a `gopen`, a group `read` or `write`, and a group `close`. When an equivalence aggregation is employed during group inspection, `gbindaggr` is also used.

As shown in Figure 6.9(a), tbon-dbg group control operations have performance that can be estimated with high precision by adding the times for the underlying `gopen`, `write`, and `close` operations. Since the overhead of stream creation during `gopen` and the first group `write` dominate the performance, it is beneficial for oft-used groups to be held open to avoid this initialization overhead.

The tbon-dbg group inspection operations shown in Figure 6.9(b) also include the time spent processing data results on the client so they can be printed in the command output. In our experiments, command output was sent to `/dev/null` to avoid any cost for local file system access. For concatenated data, the processing involves iteration over the result for each group member. When data has been placed into equivalence classes, processing costs are much lower as tbon-dbg simply needs to print the membership of each class, and the number of classes is usually small. In the figure, we see that GPR register set collection using concatenation requires over 1.5 seconds to gather and print approximately 26 megabytes of data (128 bytes per group member, to hold sixteen 64-bit register values) at the largest scale. Event collection using concatenation performs only slightly better due due a smaller read count (96 bytes per group member, to hold up to four 24-byte event records). In contrast, collecting the same event data using an equivalence aggregation reduces the latency by over 50%. The best performance in Figure 6.9(b) comes from using an equivalence aggregation to gather program-counter register values from a thread group. This curve represents the best-case scalability for using a

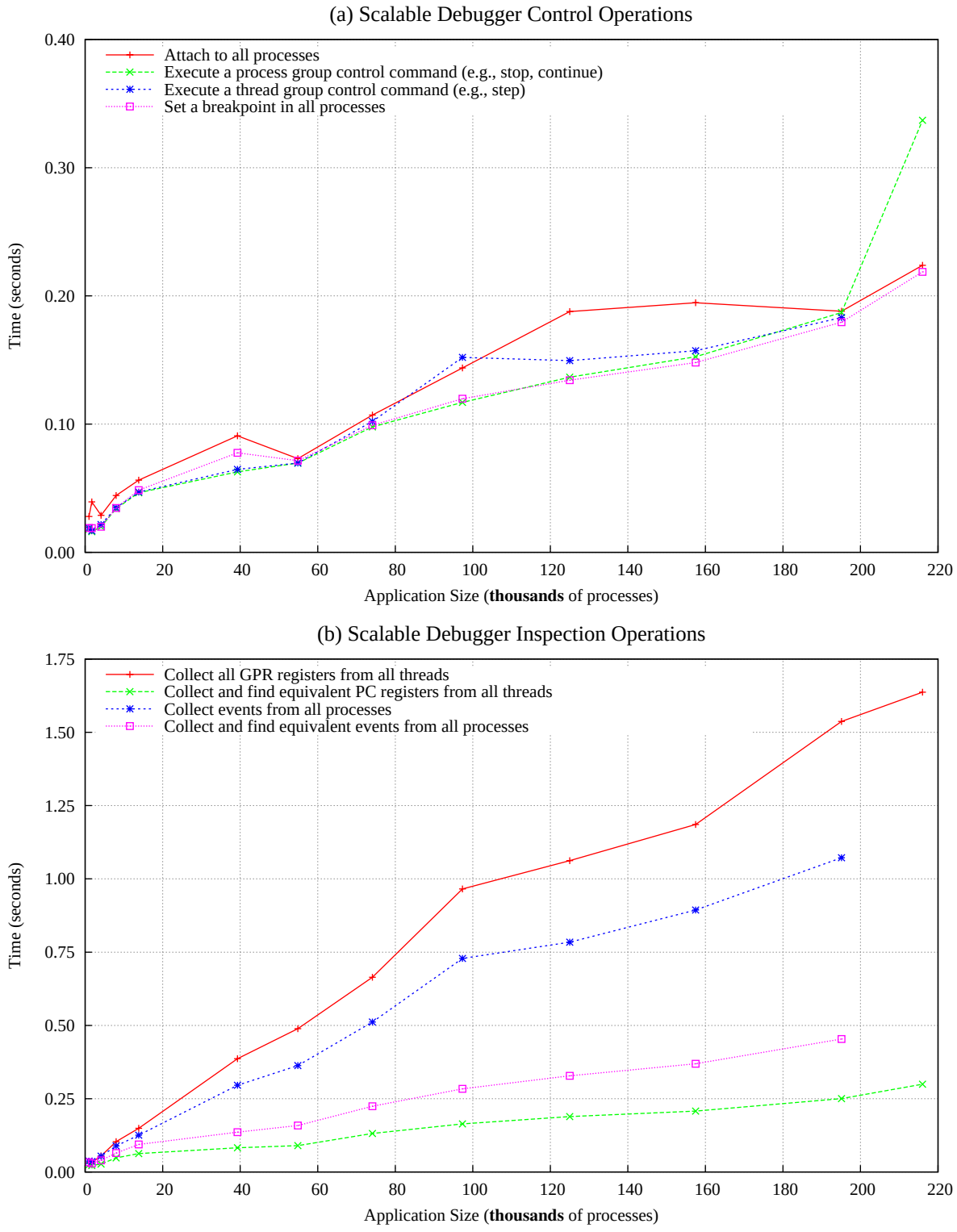

Figure 6.9 tbon-dbg Group Control and Inspection Operations
Time required to complete common debugging tasks using distributed proc++ files

value equivalence aggregation, as all threads were stopped at the same address.

6.3 Summary

The design of proc++ was motivated by observing that many tool activities are hindered by interactions with existing procfs abstractions, and new abstractions were required to enable efficient and widespread use of group file operations. Abstractions were carefully designed to be context-insensitive and to place functionality at the appropriate layer of the distributed tool hierarchy, which enables the parallelism needed to scale. When combined with group file operations and TBON-FS, proc++ provides excellent scalability for a wide-vareity of group control and inspection operations. A simple parallel debugger we have developed using proc++ is able to provide sub-second latencies for common group debugging tasks at over 200,000 processes.

We intend to distribute proc++ as open-source software to serve as the basis for the implementation of other scalable tools. Before the first public release of proc++, however, we need to finish implementing a subset of core functionality that was not required for our evaluation. This functionality includes generating thread stack traces using StackWalkerAPI and providing process and thread status information including resource usage.

To improve the utility of proc++, we plan to extend its interface to provide support for user-level threads. We also plan to extend our use of abstractions to overcome problems related to varied context. Finally, although ProcControlAPI and StackWalkerAPI support multiple platforms, proc++ has been used only on x86 and x86-64 hosts running the Linux OS. In future releases, we will add support for the other platforms supported by the underlying libraries; support for IBM BlueGene platforms is currently the highest-priority target.

Chapter 7

Case Study: Tools for Distributed System Administration

Our first case study is designed with two primary goals. First, we wish to demonstrate the ease of using the group file operation idiom in the construction of new scalable tools. Since ease-of-use is a qualitative and often subjective measure, we attempt to quantify the benefits of using the idiom by reporting the time required to develop the tools and relevant source code metrics. Second, we evaluate the performance of our implementation of group file operations within the TBON-FS group file system on large distributed systems.

The context chosen for this case study is scalable tools for administration and monitoring of distributed systems. In particular, we developed parallel versions of common Unix utilities such as `cp`, `grep`, and `top`. The focus on parallel command-line tools is motivated by our previous work in the Cluster Command & Control (C3) tool suite [18], which itself was inspired by the efforts of the Parallel Tools Consortium [46,87]. These prior works introduced useful techniques for denoting the sets of hosts targeted by a parallel tool and for delimiting output generated on specific hosts. Unfortunately, the implementations used in previous efforts provide no mechanisms for aggregating tool output to reduce the amount of data collected to the user's host. Thus, further data processing to aid in analysis

or presentation to the user is relegated to centralized computation, which represents a critical scalability barrier for large distributed systems.

In this chapter, we describe the parallel tools we have constructed using group file operations and evaluate their performance on two large clusters. When applicable, we discuss the custom data aggregations employed by the tools to avoid centralized data analysis; these aggregations leverage the scalable mechanisms for distributed aggregation within TBON-FS. Our tools fall into three classes representing their intended uses. Section 7.1 discusses two tools for replicating files across distributed hosts. Section 7.2 introduces several tools that provide scalable display of data retrieved from distributed files. Section 7.3 describes two tools that provide scalable monitoring of remote processes. We conclude in Section 7.4 with a summary of the observed qualitative and quantitative benefits of using group file operations and TBON-FS to construct scalable tools for managing distributed systems.

All experimental results were collected on two Linux clusters, Thunder and Atlas, located at Lawrence Livermore National Laboratory. The 1024-node Thunder cluster uses a Quadrics QsNetII Elan4 interconnect, and each node has four 1.4GHz Intel Itanium2 processors and 8GB of memory. The 1152-node Atlas cluster has nodes containing eight 2.4GHz AMD Opteron processors and 16GB of memory, and uses a 4X-DDR Infiniband interconnect. Due to job resource limits, we could use up to 493 and 700 nodes at a time on Thunder and Atlas, respectively. To overcome this limit, we ran all Thunder experiments with four TBON-FS servers per node and Atlas experiments with eight servers per node. In all experiments, the TBON-FS client tool and internal tree processes were run on nodes separate from those hosting TBON-FS servers. The topologies used during experiments on each cluster are shown in Tables 7.1 and 7.2.

7.1 File Replication

There are two basic approaches to managing homogenous configurations of hosts in large distributed systems, file replication or file sharing. With replication, files are kept on disks local to each host

Number of TBON-FS Servers	TBON Topology (fan-out per tree level)
576	24 × 24
784	28 × 28
1,024	32 × 32
1,280	8 × 10 × 16
1,536	6 × 16 × 16

Table 7.1 Topologies used on Thunder

Number of TBON-FS Servers	TBON Topology (fan-out per tree level)
1,024	4 × 8 × 32
2,048	8 × 8 × 32
3,072	8 × 12 × 32
4,096	8 × 16 × 32

Table 7.2 Topologies used on Atlas

and must be updated if the global configuration changes, such as when new applications are installed or a system configuration file is updated. When file sharing is employed, a distributed or parallel file system serves the files to all hosts and changes only need to be applied centrally.

Unfortunately, on very large systems, the file sharing approach can become a bottleneck. In situations where a majority of clients request the same file at the same time, the server may not be able to service all requests in a timely manner. A common example of this situation occurs when an entire distributed system is restarted and every host needs to access the same set of startup files. To avoid these situations, administrators often replicate a small set of oft-accessed files.

To improve the scalability of file replication and global configuration updates, we have developed

parallel versions of both `cp` and `rsync`:

- `pcp` uses group write operations to multicast a source file to all servers.
- `psync` uses the `rsync` block checksum comparison algorithm [32] to identify differences between the servers' copies of a file and the source file on a client, and only multicasts the data that has been changed or added. `psync` uses group read operations with an aggregation that checksums file data blocks. The aggregation identifies sub-groups of servers whose file data is identical. The block checksums for each sub-group are compared to the source file, and updated data is multicast using a group write. In homogenous environments with one unique group, the computational load for `psync` on the client is similar to `rsync` with a single server.

During the development of `psync`, we also created a new tool named `pchecksum` that performs checksumming to identify groups of files with equivalent contents. `pchecksum` is useful for quick identification of divergent files across distributed hosts.

We evaluated the performance of `pcp` and `psync` in terms of replication time for three file sizes on the Atlas cluster. Table 7.3 shows the size of each file. To support our experiments using `psync` that only distribute file updates, the files were chosen by looking for configuration files in `/etc` that had been modified recently on Atlas. We selected files where both the current and previous versions were available. The previous version was used as the copy at all servers, while the client source file was the current, modified version.

For comparison, we measured the time required for every destination host to use the `cp` command to copy the file from an NFS and a Lustre file system. Lustre [109] is a parallel file system designed for providing a large number of clients concurrent access to large data sets whose data is spread across many servers. The `cp` tasks were launched in parallel using the `srun` command of SLURM [110], which is the system used on Atlas for launching serial and parallel programs. To account for the overhead of `srun`, we measured the time to run a trivial program, `hostname`, in parallel at each experi-

		psync		
File	Size (bytes)	New / Modified Data (bytes)	Block Matches Meta-data (bytes)	Total Data Transmitted (% of total size)
/etc/fstab	7,063	1,579	216	25.4%
/etc/passwd	38,742	1,025	1,332	6.1%
/etc/services	559,563	3,531	19,548	4.1%

Table 7.3 Replicated File Statistics

Reports the total size of each file. When replicated with psync, only new data and meta-data for unchanged file blocks are multicast to TBON-FS servers. The size of new data and meta-data is given for each file, and the sum of these two sizes is reported as a percentage of the total file size.

mental scale, and subtracted that time from the NFS and Lustre copy times.

Two versions of pcp were tested, one using synchronous group write and one using asynchronous group write. In all experiments, synchronous pcp was faster, so we report only those results, although the asynchronous version still outperformed the parallel cp from NFS and Lustre. All cp and pcp experiments use /dev/null as the destination file to avoid the effects of local disk writes.

Figure 7.4 shows the replication times for each of the three files. pcp is always the fastest, and shows logarithmic performance as we increase the number of destinations. Lustre comes in second, but shows poor linear scaling. We speculate that Lustre's poor performance stems from the relatively small files being accessed, which are not likely to be partitioned across many servers. Thus, all hosts are forced to copy the file from a single Lustre data server. Interestingly, NFS does not exhibit linear scaling, but is an order of magnitude slower than pcp. We expected the parallel copy from NFS to have similar linear behavior to Lustre, but we think that caching of file blocks at the NFS client hosts is mitigating many of the remote requests to the NFS server. Caching effects would also help to explain why some NFS copies at larger scales occasionally complete slightly faster than at smaller scales.

For the two smaller files, psync is faster than the parallel copy from NFS at all but the largest

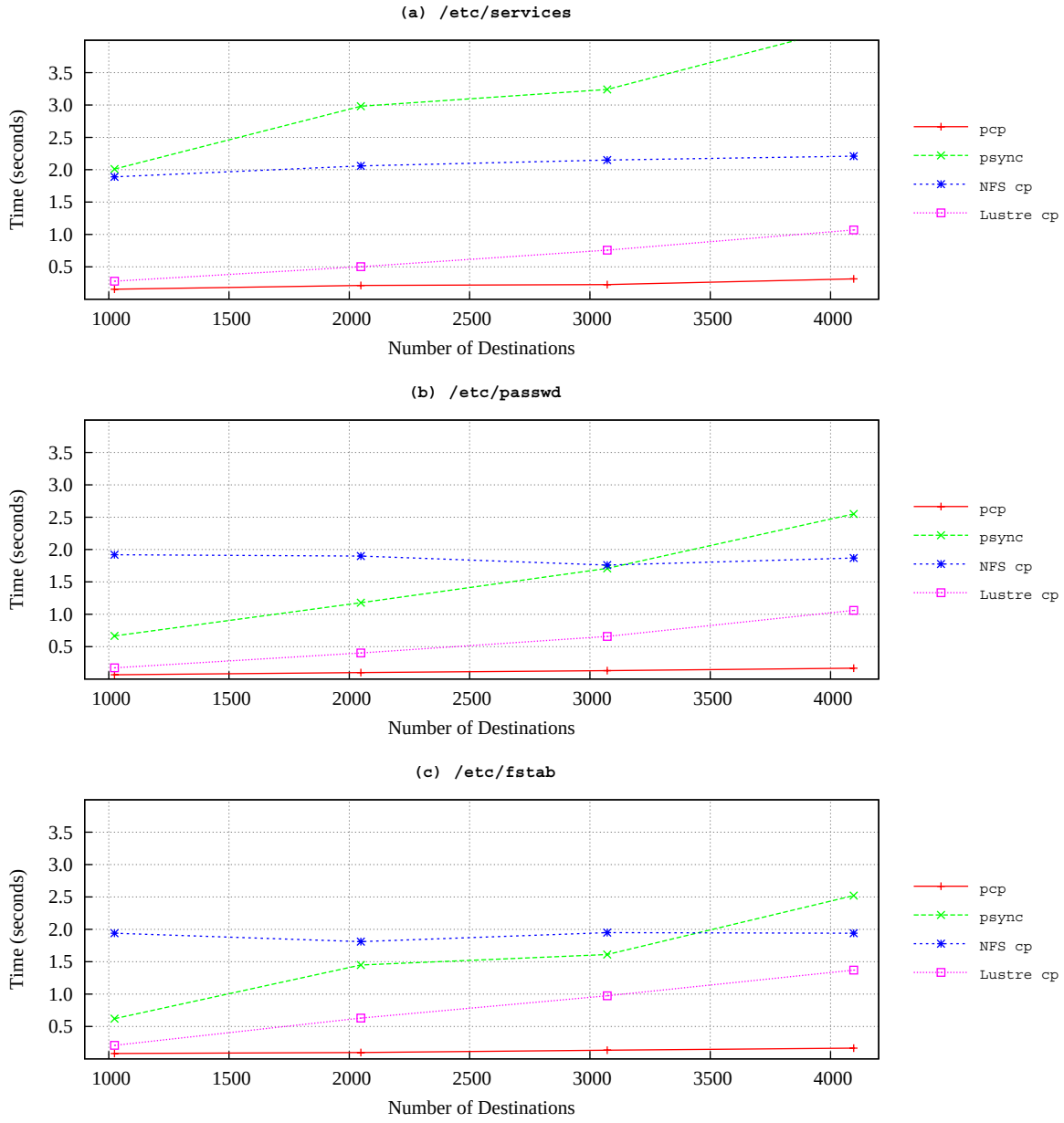

Figure 7.4 File Replication Scalability

Compares the scalability of file replication for three files using `pcp` and `psync`, which multicast file data from the source host to destination hosts, versus parallel invocation of `cp` on destination hosts to pull the files from NFS and Lustre.

(a) Time to replicate `/etc/services`.

(b) Time to replicate `/etc/passwd`.

(c) Time to replicate `/etc/fstab`.

experimental scales, even though it must read and checksum all of the destination files. For the largest file, the overhead of checksumming and comparing more than five hundred 1024-byte blocks becomes too costly. However, `psync` sends much less data over the network by only transmitting new data and information detailing the matched blocks. Table 7.3 reports the bytes sent by `psync` as a percentage of the file sizes. Due to its network savings, `psync` is an attractive option for updating files on systems where the network is highly-utilized, yet computation is comparatively cheap.

7.2 File Inspection

Many command-line utilities are used by system adminstrators for inspecting text-based files corresponding to configuration files and system logs. Common examples include `cat`, `head`, `tail`, and `grep`. In this section, we describe our parallel versions of these utilities that are designed for operating on large groups of distributed files.

Because users are not likely to be happy with digesting thousands of lines of output, all of our parallel utilities use an aggregation based on line equivalence. This aggregation identifies each unique line found in any member file. Unique lines are accompanied by a list of member files that contained the line. Member files are represented as strided ranges of group file indices, where each range has a start index, stride, and count. Each equivalence class generated by the aggregation can be printed on a single line that includes the member ranges and the unique line's text, as shown in the following format:

```
(start,stride,count)[,...] unique-line-text
```

This format is human-readable for cases involving significant equivalence, and is amenable to further text-processing using scripts when necessary.

A strided range representation has benefits over simple ranges because it can compactly represent periodic intervals of varying size. A notable example of this benefit is in cases of odd-even behavior, when all odd indices share the same value and all even indices share a different value. Using simple ranges to represent odd-even behavior devolves into a list of every index, while the same behavior can

be represented using just two strided ranges.

In its base form, the line-equivalence aggregation does not distinguish between lines that may be repeated, such as may occur when an error is indicated many times in a system log. To help distinguish such repetitions, we added the ability to prepend the line numbers from source files. When line numbers are used, both the line's number and text are used to determine uniqueness. Keeping track of the number of lines processed and partial lines from individual member files was the biggest challenge in implementing the aggregation.

7.2.1 Parallel `cat` and `head`

The first utility we created using the line-equivalence aggregation was `pcat`, a parallel version of `cat`. `pcat` takes the name of a target file as an operand, and then creates a group file representing the target file on every server host. Data is processed by a loop that uses group reads of file blocks until all the data has been read. Users can optionally specify the block size to use during reads. The line equivalence classes generated during each group read are printed at the end of each loop iteration. Figures 7.5 and 7.6 show the benefits of using line equivalence with strided membership ranges in `pcat`. `pcat` supports an option for dumping the mappings of group member indices to distributed files to aid in further processing and line attribution.

Using `pcat` as a starting point, it was easy to create `phead`, our parallel version of `head`. We simply extended the line-equivalence aggregation with a parameter that indicates the maximum line number. Once each member file has produced the maximum number of requested lines, the aggregation function simply ignores subsequent data.

7.2.2 Parallel `grep`

One of the most useful Unix utilities is `grep`. System administrators use `grep` for various tasks including searching configuration files, scanning system and application logs for interesting or alarm-

```
(0) nfs 398778 2 - Live 0x0
(0) lockd 74270 1 nfs, Live 0x0
(0) nfs_acl 2647 1 nfs, Live 0x0
(0) auth_rpcgss 44895 1 nfs, Live 0x0
(0) sunrpc 244046 18 nfs,lockd,nfs_acl,auth_rpcgss, Live 0x0
(0) acpi_cpufreq 7955 1 - Live 0x0
(0) freq_table 4881 1 acpi_cpufreq, Live 0x0
(0) mperf 1557 1 acpi_cpufreq, Live 0x0
(0) nvidia 11933029 26 - Live 0x0 (P)
(0) i2c_i801 11231 0 - Live 0x0
(0) i2c_core 31276 2 nvidia,i2c_i801, Live 0x0
(0) iTCO_wdt 13662 0 - Live 0x0
(0) iTCO_vendor_support 3088 1 iTCO_wdt, Live 0x0
(0) e1000e 219500 0 - Live 0x0
(0) i7core_edac 18184 0 - Live 0x0
(0) edac_core 46773 1 i7core_edac, Live 0x0
(0) ext4 364410 1 - Live 0x0
(0) mbcache 8144 1 ext4, Live 0x0
(0) jbd2 88834 1 ext4, Live 0x0
(0) ahci 40455 2 - Live 0x0
(1) nfs 398778 2 - Live 0x0
(1) lockd 74270 1 nfs, Live 0x0
(1) nfs_acl 2647 1 nfs, Live 0x0
(1) auth_rpcgss 44895 1 nfs, Live 0x0
(1) sunrpc 244046 18 nfs,lockd,nfs_acl,auth_rpcgss, Live 0x0
(1) acpi_cpufreq 7955 1 - Live 0x0
(1) freq_table 4881 1 acpi_cpufreq, Live 0x0
(1) mperf 1557 1 acpi_cpufreq, Live 0x0
(1) i2c_i801 11231 0 - Live 0x0
(1) i2c_core 31276 2 i2c_i801, Live 0x0
...
          NOTE: THIS FIGURE WOULD NEED TO BE 40 TIMES BIGGER TO SHOW CONTENT FOR ALL 64 HOSTS
```

Figure 7.5 Parallel `cat` - Attributed Output without Line Equivalence Aggregation (~1.5 hosts)

The figure shows how the contents of the Linux `/proc/modules` file, which lists the currently loaded kernel modules, would be printed using `pcat` on a cluster of 64 hosts if no line equivalence aggregation was used. For clarity, we have shown all lines from the same source together, although in practice lines from different sources may be interleaved. To fully display the contents from all 64 cluster hosts, this figure would have needed to be forty times larger.

```
(0,1,64) nfs 398778 2 - Live 0x0
(0,1,64) lockd 74270 1 nfs, Live 0x0
(0,1,64) fscache 46859 1 nfs, Live 0x0
(0,1,64) nfs_acl 2647 1 nfs, Live 0x0
(0,1,64) auth_rpcgss 44895 1 nfs, Live 0x0
(0,1,64) sunrpc 244046 18 nfs,lockd,nfs_acl,auth_rpcgss, Live 0x0
(0,1,64) acpi_cpufreq 7955 1 - Live 0x0
(0,1,64) freq_table 4881 1 acpi_cpufreq, Live 0x0
(0,1,64) mperf 1557 1 acpi_cpufreq, Live 0x0
(0,8,8) nvidia 11933029 26 - Live 0x0 (P)
(0,1,64) i2c_i801 11231 0 - Live 0x0
(0,8,8) i2c_core 31276 2 nvidia,i2c_i801, Live 0x0
(1,1,7),(9,1,7),(17,1,7),(25,1,7),(33,1,7),(41,1,7),(49,1,7),(57,1,7) i2c_core 31276 2 i2c_i801, Live 0x0
(0,1,64) iTCO_wdt 13662 0 - Live 0x0
(0,1,64) iTCO_vendor_support 3088 1 iTCO_wdt, Live 0x0
(0,1,64) e1000e 219500 0 - Live 0x0
(0,1,64) i7core_edac 18184 0 - Live 0x0
(0,1,64) edac_core 46773 1 i7core_edac, Live 0x0
(0,1,64) ext4 364410 1 - Live 0x0
(0,1,64) mbcache 8144 1 ext4, Live 0x0
(0,1,64) jbd2 88834 1 ext4, Live 0x0
(0,1,64) ahci 40455 2 - Live 0x0
```

Figure 7.6 Parallel `cat` - Line Equivalence using Strided Ranges (64 hosts)

The figure shows the results of using `pcat` with line equivalence and strided range membership lists. Compared to Figure 7.5, we see that the entire aggregated contents from all 64 hosts fits within the same space as that required for printing a single host's file contents. Further, the aggregation reveals that one of every eight hosts in the cluster contains an Nvidia GPU, which results in the `nvidia` module being loaded and the `i2c_core` module depending on `nvidia`.

ing events, and gathering host or process information. Leveraging `grep` on distributed files helps identify configuration differences, correlate distributed events, and monitor resource use. Thus, we have developed `pgrep`. For simplicity, it currently supports text string searches rather than regular expressions. `pgrep` employs a modified version of our line-equivalence aggregation. This version introduces an aggregation function parameter that denotes the search string.

We compared `pgrep` on TBON-FS files to standard `grep` on files served by NFS for the same number of files searched. In each experiment, we measured the completion latency in seconds and the size of the generated output for matching lines. We searched files stored in both disk and memory, and used searches that returned few or many unique matches. **services-udp** searched for the abundant string "`udp`" in the `/etc/services` file. **meminfo-Total** searched for "`MemTotal`" in `/proc/meminfo`. This search returns a single line that is the same across all hosts in a homogeneous environment such as Atlas. **meminfo-Free** searched `/proc/meminfo` for "`MemFree`". Since the amount of free memory is highly variable at runtime, this search returns many unique lines. Figure 7.7 shows a few lines of the `pgrep` output for each search.

As shown in Figure 7.9, `grep` exhibits the expected linear scaling in latency and output size. Due to `pgrep`'s equivalence aggregation, latency is logarithmic and the number of output lines is simply the number of unique lines across all hosts. As a result, `pgrep`'s output is smaller in size and easier to interpret than `grep`, and provides up to an order of magnitude reduction in cases with significant similarity across files. For `pgrep`, latency includes the time for three sub-tasks: (1) initialize the file group using `gopen` and `gbindaggr`, (2) read the files using the equivalence aggregation, and (3) print the aggregated results. In Table 7.8, we report the sub-task latencies observed during `pgrep` searches on groups of 4096 files. We note that for smaller files the `pgrep` latency is dominated by the time to initialize the group file.

(a) services-udp

```
(0,1,4096)  novastorbakcup  308/udp                    # Novastor Backup
(0,1,4096)  irc             194/udp
(0,1,4096)  ni-ftp          47/udp                     # NI FTP
(0,1,4096)  hdap            263/udp                    # HDAP
(0,1,4096)  link            245/udp    ttylink
...
```

(b) meminfo-Total

```
(0,1,4096)  MemTotal:     15528448 kB
```

(c) meminfo-Free

```
(1080,1,3),(1084,1,2)  MemFree:      15098640 kB
(3419,3,2)  MemFree:      15107204 kB
(2762,1,3)  MemFree:      15103764 kB
(2872,1,1)  MemFree:      15096484 kB
(2627,1,1)  MemFree:      15086596 kB
(2675,1,1)  MemFree:      15099176 kB
(1560,5,2)  MemFree:      15104080 kB
(2440,1,2),(2443,1,1)  MemFree:      15102052 kB
(2722,1,1)  MemFree:      15121824 kB
(2931,1,2)  MemFree:      15095204 kB
(1728,1,8)  MemFree:      15113272 kB
...
```

Figure 7.7 Line Equivalence Output from `pgrep`
(a) Searching `/etc/services` for "`udp`" results in one output line per match at all 4096 servers.
(b) Searching `/proc/meminfo` for "`MemTotal`" results in one output line representing all 4096 servers.
(c) Searching `/proc/meminfo` for "`MemFree`" results in one output line for each unique value.

	SUB-TASK LATENCY (MILLISECONDS)		
SEARCH	GROUP INIT	GROUP READ	PRINT OUTPUT
services-udp	102	780	4
meminfo-Total	113	7	< 1
meminfo-Free	118	21	< 1

Table 7.8 `pgrep` Sub-task Latencies for File Groups of Size 4,096

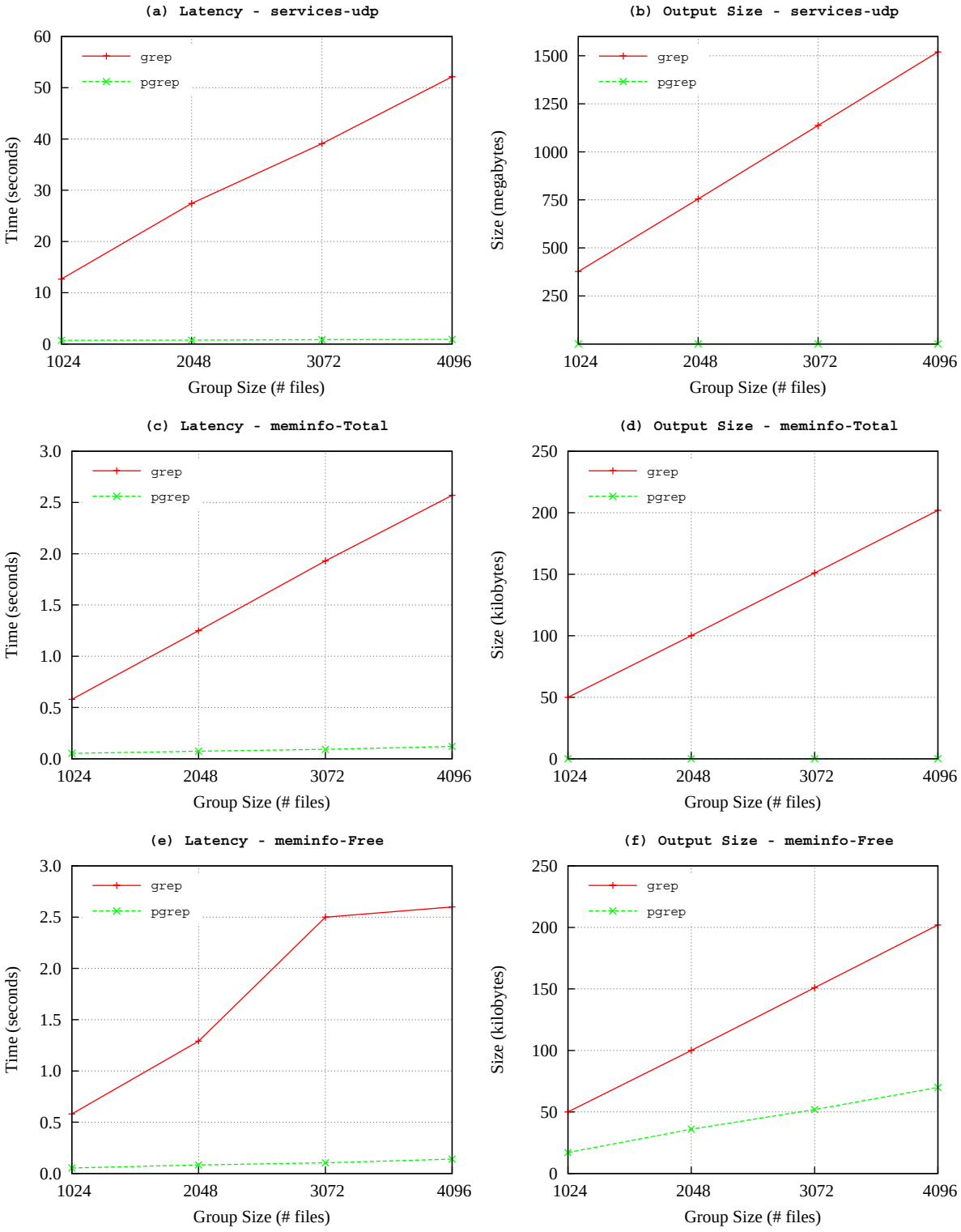

Figure 7.9 Parallel `grep` Scalability
Latency and output size for three searches using `pgrep` on TBON-FS files and `grep` on NFS files.

7.2.3 Parallel `tail`

The `tail` utility with the "`-f`" option can be used to follow system log activity in real-time. Existing software enhances this use of `tail` to include monitoring multiple files [75] and multiple hosts [121], and to highlight interesting lines of output [47,75]. Combining all three enhancements, we developed `ptail` for following large distributed file groups. To improve correlation of events across hosts and reduce output, we extended our line equivalence aggregation with an option to strip host-specific information (e.g., host name and process IDs) from lines using the `syslog` message format. Correlation of distributed events can benefit applications such as identifying misconfigured network services (e.g., many clients of the service notice a problem and generate an identical error) and security (e.g., distributed intrusion or denial-of-service attacks).

We used a synthetic log generator to evaluate `ptail`. The log generator controls the rate and the percentage of equivalent entries. `ptail` significantly reduces the number of output lines by combining equivalent log entries. For a group of size G hosts, L log entries generated per host, and a percentage P of equivalent events, the total number of output lines is reduced to $L((1-P)G+P)$ from the total lines generated $L \times G$. Aggregated output eases analysis and reduces the storage needed to keep logs.

7.3 Process Monitoring

For many administration tasks, it is useful to know which processes are using the most resources. `top` is a simple yet powerful utility for displaying resource utilization by processes on a single host. We created `ptop` to provide similar functionality for many distributed hosts. `ptop` gathers information from files in the Linux proc file system. Custom data aggregations are used to calculate summaries and support the sorting and filtering capabilities of `top`. To give greater insight into distributed resource use, we added two new grouping facilities that summarize across processes executing the same program, both for a specific user and across all users. When grouping, one can view total, average, or maximum utilization. A screenshot of `ptop` executing on Thunder is shown in Figure 7.10. In the fig-

```
ptop - Thu Apr 24 22:46:36 2008
1024 hosts up 96430.24 days, load average: 0.30, 0.13, 0.08
Tasks: 338112 total, 4347 run, 333765 sleep, 0 stopped, 0 zombie
CPU: 4096 cpu(s), 78.72% user, 0.86% sys, 0.00% nice, 19.65% idle, 0.76% wait
Mem:  8441839552k total,1071405120k used,7370434432k free, 169090944k buffers
Swap: 17182572544k total,  71227968k used,17111344576k free, 200221184k cached
    USER             %CPU            %MEM            COMMAND
------------    -------------  -------------  -----------------
       brim1       1.48 @4096      0.06 @4096   tbonfs-server
        root       0.13 @1024      0.01 @1024   spd
        root       0.00 @856       0.00 @856    ksoftirqd/1
        root       0.01 @392       0.00 @392    elan4_mainint
        root       0.00 @884       0.00 @884    ksoftirqd/2
        root       0.03 @92        0.00 @92     ptlrpcd
        root       0.01 @416       0.00 @416    ksoftirqd/0
        root       0.00 @1016      0.00 @1016   ksoftirqd/3
        root       0.00 @436       0.01 @436    syslog-ng
        root       0.00 @272       0.00 @272    kjournald
         ntp       0.00 @980       0.03 @980    ntpd
        root       0.00 @404       0.01 @404    munged
        root       0.00 @60        0.00 @60     ll_ping
        root       0.00 @1020      0.01 @1020   lrmmond
        root       0.00 @124       0.00 @124    ep_comms
        root       0.00 @724       0.00 @724    irqbalance
        root       0.00 @68        0.00 @68     kqswnal_sched
        root       0.00 @1008      0.00 @1008   ldlm_cn_17
        root       0.00 @1008      0.00 @1008   ldlm_cn_16
        root       0.00 @1008      0.00 @1008   ldlm_cn_15
```

Figure 7.10 `ptop` Running on Thunder

Similar to `top`, `ptop` provides summary information for the state of processes and resource use in the upper portion of its display and specific process information in the lower portion. However, in `ptop` summaries are calculated across all processes from all servers, and aggregate information for groups of processes running the same program are displayed. As shown in the screenshot, `ptop` is currently monitoring over 330,000 processes located across the 1,024 hosts of Thunder, which contains 4,096 processors and over 8 terabytes of memory. We also see that the group of 4,096 TBON-FS server processes is using 1.5% of the CPU and 0.06% of memory on average. To generate all of the data shown in this `ptop` display, over one million Linux `/proc` files were read and aggregated.

ure, `ptop` is displaying the top-twenty CPU users and reporting averages for CPU and memory utilization. On the otherwise unloaded cluster hosts, we see that the group of 4,096 `tbonfs-server` processes is using the most CPU at 1.5% on average across all hosts.

Using `ptop`, one can answer many interesting questions: what application is using the most resident memory pages, what is the average CPU utilization for my parallel application's processes, and which users are playing solitaire? Furthermore, a parallel version of `ps` could easily be constructed

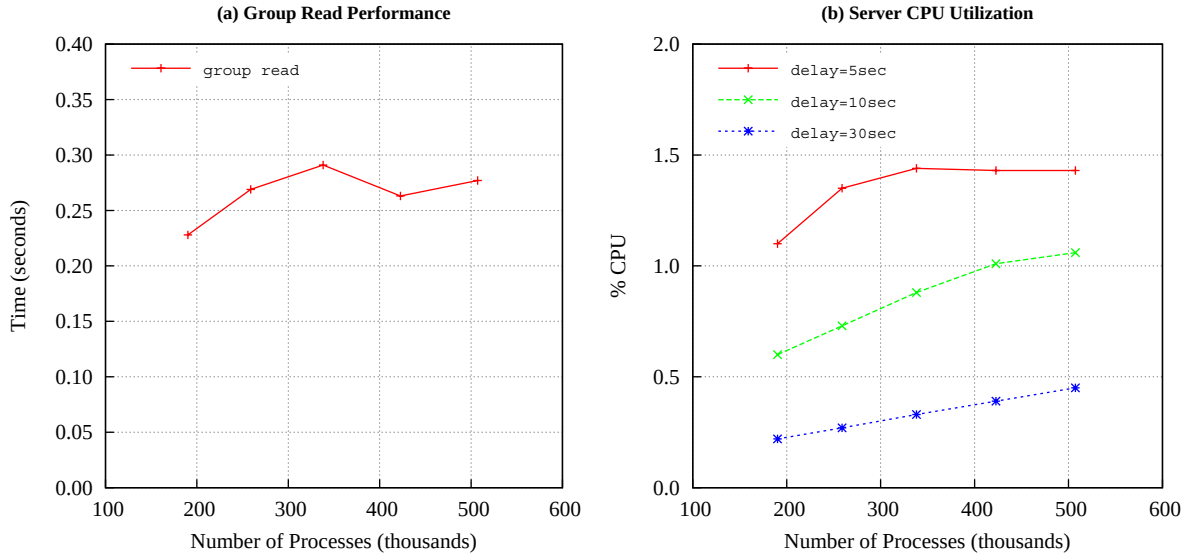

Figure 7.11 Parallel `top` Scalability
(a) Average time to for `ptop` to complete a group `read` operation that aggregates process information from Linux `/proc` files.
(b) Average CPU utilization at each TBON-FS server during use of `ptop` with varied refresh delays.

from the `ptop` source, as the functionality of `ps` is a subset of that of `top`.

To evaluate `ptop`, we measured average latency to collect and aggregate process information and average CPU utilization at TBON-FS servers. On Thunder, we ran `ptop` for 60 seconds with and without command grouping, using delay intervals of 5, 10, and 30 seconds (`top`'s default is 5 seconds) and reporting the top 100 processes. Performance with and without grouping was indistinguishable, so we report the results for the grouping case. Figure 7.11(a) shows that `ptop` can aggregate resource utilization for group files representing hundreds of thousands of distributed processes (several hundred processes per host) in less than 300 milliseconds. The latency scales logarithmically compared to the group file size. Since seven group files are read each time `ptop` updates its display, `ptop` requires just over two seconds per update. As a result, `ptop` can provide the same default delay interval as `top`.

Figure 7.11(b) shows that TBON-FS server CPU use was under 0.5% for the 30 second interval in all experiments. Thus, we believe `ptop` can be used for low-impact, continuous monitoring. Based on

		Lines of Code		
Parallel Tool	Development Time	Total	Group File Operations	Custom Data Aggregation
pcp	1 day	219	24	N/A
psync	1 week	820	60	252
pchecksum	1 day	364	37	115
pcat, phead	2 days	318	20	160
ptail	2 days	410	22	205
pgrep	1 day	356	20	191
ptop	1 week	1,455	83	834
tbonfs_bench	1 day	492	31	255

Table 7.12 Development Time and Lines of Code for Parallel Tools

ptop's low-overhead performance, we were motivated to develop a benchmarking tool called tbonfs_bench that samples the CPU and memory utilization of a set of distributed processes. In our Ganglia case study presented in the next chapter, we used tbonfs_bench to collect the resource use of TBON-FS processes. Thus, we can *use trees to monitor trees*.

7.4 Summary

Overall, group file operations appear to be quite easy to use for creating new scalable tools for managing distributed systems. The performance of these parallel command-line tools generally matched our expectations for the first demonstrations of the idiom and and its associated TBON-FS implementation. The results showed the potential for scalability provided by group file operations and TBON-FS, but also revealed that group file initialization, particularly the latency of gopen, required further attention and performance optimization. The improved performance for gopen previously presented in Chapter 6 was a result of this optimization effort.

To help quantify ease-of-use, we report the approximate development time and relevant source

code metrics related to our use of group file operations in Table 7.12. Source code metrics are shown in terms of lines of code (excluding blank lines and comments) measured using cloc [27]. We show the total lines of code, as well as lines corresponding to use of group file operations and the implementations of custom data aggregation functions. The two most complex tools, `psync` and `ptop`, were developed in just one week each. The other tools required just one or a few days to develop due to the simplicity of using group file operations to accomplish the desired task, and because we were able to re-purpose or extend previously developed custom data aggregations. Developing custom data aggregations was by far the most time-consuming activity, and the associated aggregation function code represents a large percentage of the total code for each parallel tool. We expect this to be the case for most new tool development using group file operations.

Chapter 8

Case Study: Ganglia Distributed Monitoring System

Our second case study is designed to evaluate the ease of integrating group file operations and TBON-FS within an existing middleware system, as well as the resulting benefits to performance and scalability. In this study, we chose to use the Ganglia Distributed Monitoring System [67] for two reasons. First, Ganglia is open-source, widely-used, and known to have problems with monitoring large clusters containing many thousands of hosts. Thus, any improvements from our work will benefit many existing users, and should enable new deployments on large clusters. Second, Ganglia gathers much of its metric data from files and computes summary statistics over this data from many hosts, which equates nicely with the goals of scalable group file operations.

We discuss the Ganglia architecture and its scalability barriers in Section 8.1. The architecture of Ganglia-tbonfs, our modified version of Ganglia that uses group file operations to collect and aggregate metric data, is presented in Section 8.2. We evaluate the performance of Ganglia-tbonfs versus Ganglia 3.0.4 in Section 8.3. The chapter concludes in Section 8.4 with a summary of our integration effort and its benefits for performance and scalability, and a short discussion of potential further improvements.

8.1 Architecture of the Ganglia Distributed Monitoring System

Ganglia is designed to monitor resource utilization of distributed hosts and provide system administrators with a history of the measured data for individual hosts and groups of hosts. Common metrics monitored by Ganglia include the utilization of the processors, memory, network, and local disks at each host. Ganglia aggregates data to provide statistical summaries such as minimum, maximum, and average values for clusters and grids. A cluster is a group of hosts, and a grid is group of clusters or other grids. Thus, Ganglia provides a hierarchical monitoring infrastructure that can support organizations that have many distributed systems and hierarchical administrative structures, such as is common in government research labs and universities.

Metric histories are supported by using round-robin databases (RRDs) to store measured metric values. An RRD is a fixed-sized database that stores data for a period of time broken into fixed-length intervals [104]. Because they are fixed-size, RRDs use a first-in first-out (FIFO) strategy for data storage. As new data is added to an RRD that is fully populated, the new data overwrites the oldest data. To provide a long metric history, several RRDs can be used for each metric as shown in Figure 8.1. Each RRD records metric data at a different granularity. At the finest level of detail, data may be kept for the last day broken into fifteen-minute intervals. Further RRDs contain coarser-grained data that represents an aggregation over a smaller granularity, such as the days in the last month and the months in the last two years. Ganglia uses default periods and intervals for each metric that are designed for low-overhead monitoring, but also gives the system administrator the ability to define custom periods and intervals for metrics.

As shown in Figure 8.2, Ganglia uses a hierarchical architecture that matches the organizational hierarchy of the systems being monitored. An aggregator process (gmetad) is associated with each cluster and grid. The gmetads are organized as a tree, with grid parents collecting metric data from their child clusters and grids. Monitoring processes (gmond) are run on each host. Within a cluster, the

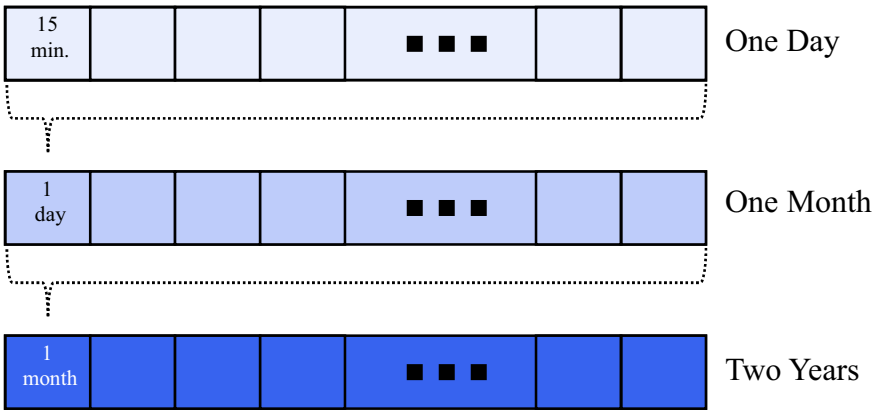

Figure 8.1 Metric Data Histories Stored in Round-Robin Databases
A round-robin database is a fixed size buffer that holds metric data collected during period of time. The period is broken into intervals of fixed-length, such as fifteen-minute intervals within a span of one day as shown in the top RRD. To provide a long metric history, multiple RRDs can be used that provide finer-grained data for recent periods, and more coarse data for long-term periods. RRDs holding long-term data are constructed by aggregating the data from shorter periods.

gmonds use IP multicast as a publish-subscribe mechanism for replicating newly measured metric data. Metric data sent from each gmond on the multicast socket is received and stored at the other gmonds, which results in data for all hosts being available at any host. The gmetad for a cluster selects a representative gmond that it contacts to collect the data for all cluster hosts. Metric data is encoded using XML when transferred between parent and child gmetads, and between a cluster gmetad and a representative gmond.

There are three scalability barriers in the Ganglia architecture. First, as noted by Ganglia's designers [67], its use of IP multicast is not scalable for large clusters containing over 2000 hosts. Because metric data published by every gmond is delivered and cached at all gmonds in the cluster, every gmond incurs network, memory, and processing utilization that grows with the total volume of data for all cluster hosts. Second, metric data for all hosts is transmitted from the representative gmond to the cluster gmetad. Since this XML data has size that grows in proportion to the number of hosts in the cluster, the time required to parse the XML to extract the per-host data increases linearly with the clus-

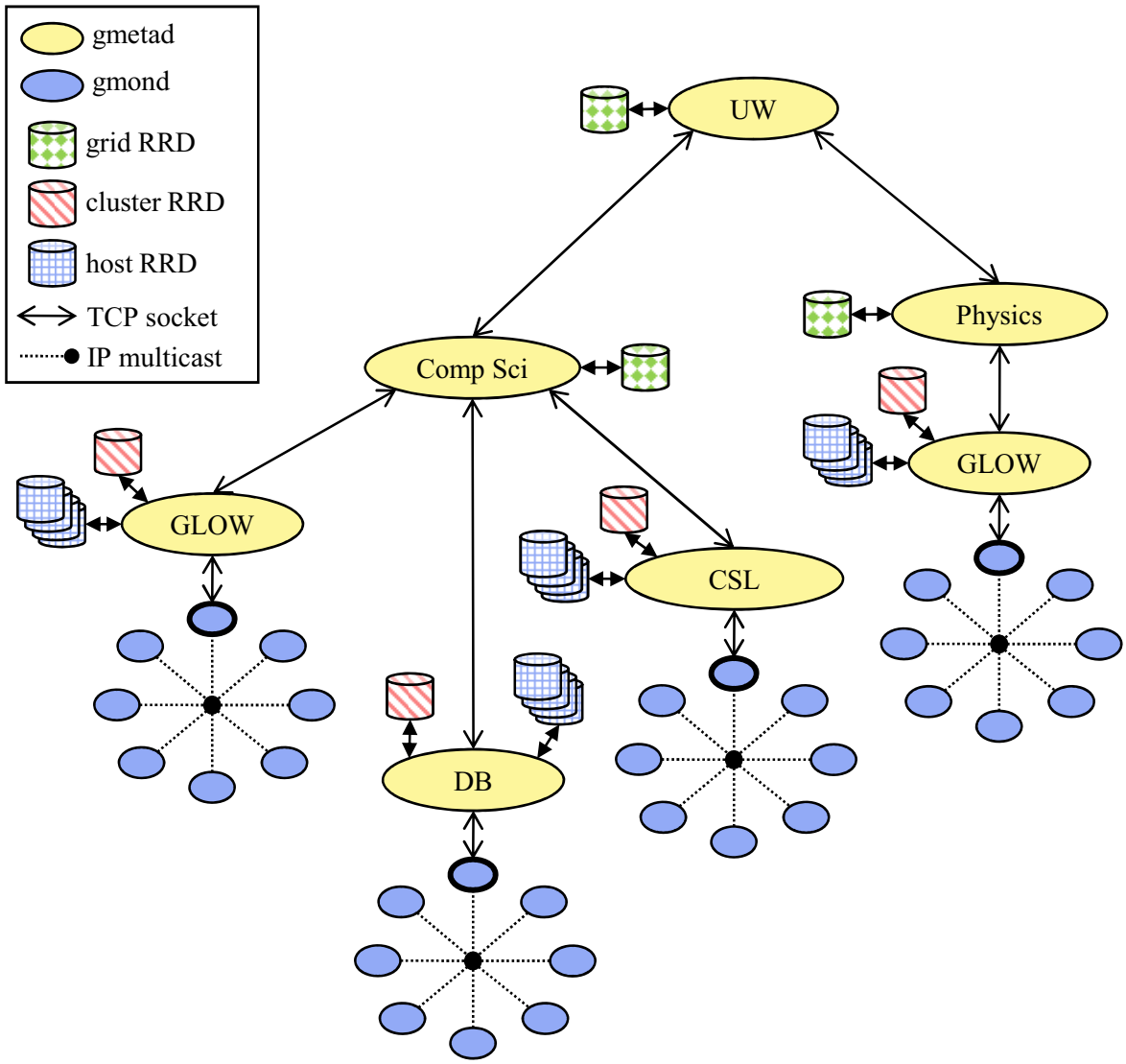

Figure 8.2 Ganglia Monitoring and Data Storage Architecture

Ganglia uses two types of processes: monitor processes (gmond) on each host and aggregator processes (gmetad) associated with each cluster and grid that compute summaries and store metric data in round-robin databases (RRDs) for each host, cluster, and grid. Aggregator processes are organized as a tree whose structure often corresponds to an organizational structure. The figure shows an example structure that could be used for monitoring hosts, clusters, and grids within departments of the University of Wisconsin. Within a cluster, gmonds distribute metric data amongst themselves using IP multicast. A cluster gmetad contacts a representative gmond (indicated by the thick outline) via TCP to collect data for all cluster hosts. Grid gmetads also use TCP to collect summary data from their children clusters and grids.

ter size. Third, the cluster gmetad iterates over the metric data for all hosts to compute metric summaries for the cluster. Grid gmetads are less problematic, since most grids contain a relatively small number of child clusters or grids. Thus, grid gmetads are not likely a bottleneck when collecting and aggregating metric data from their children and storing the metric summaries.

8.2 Ganglia-tbonfs: A Scalable Design for Monitoring Large Clusters

Ganglia-tbonfs addresses the two main scalability barriers in the original Ganglia architecture by making the entire communication architecture tree-based and distributing the work of cluster gmetads across processes in the tree. As shown in Figure 8.3, Ganglia-tbonfs replaces the existing IP multicast among gmonds with tree-based communication to TBON-FS servers. Group file operations are used by the root gmetad process to read and aggregate metric data from files on the server hosts. For metrics whose data is not already found in files, such as disk utilization, we created synthetic files to provide the necessary data. These synthetic files are implemented by a FUSE [115] user-level file service. At the time of the Ganglia case study, our FINAL synthetic file services had not yet been devised, which led to our dependence on FUSE.

The use of group file reads for metric collection results in a move from the original push architecture used within a cluster to a pull architecture. In the original design, each gmond consulted a local configuration file to know when to collect and multicast data for each metric. Then, the gmetad for a cluster would poll the representative gmond on a regular basis, every 15 seconds by default, to collect recently updated data. In Ganglia-tbonfs, these two actions are combined into periodic collections of metric data initiated by the root gmetad. The root gmetad obtains the metric collection intervals from a configuration file and identifies *collection groups*, which are sets of metrics sharing the same interval. The collection intervals for each metric can still be customized, but different intervals cannot be used at separate hosts. In practice most Ganglia users do not use different intervals at separate hosts, so this restriction has little impact.

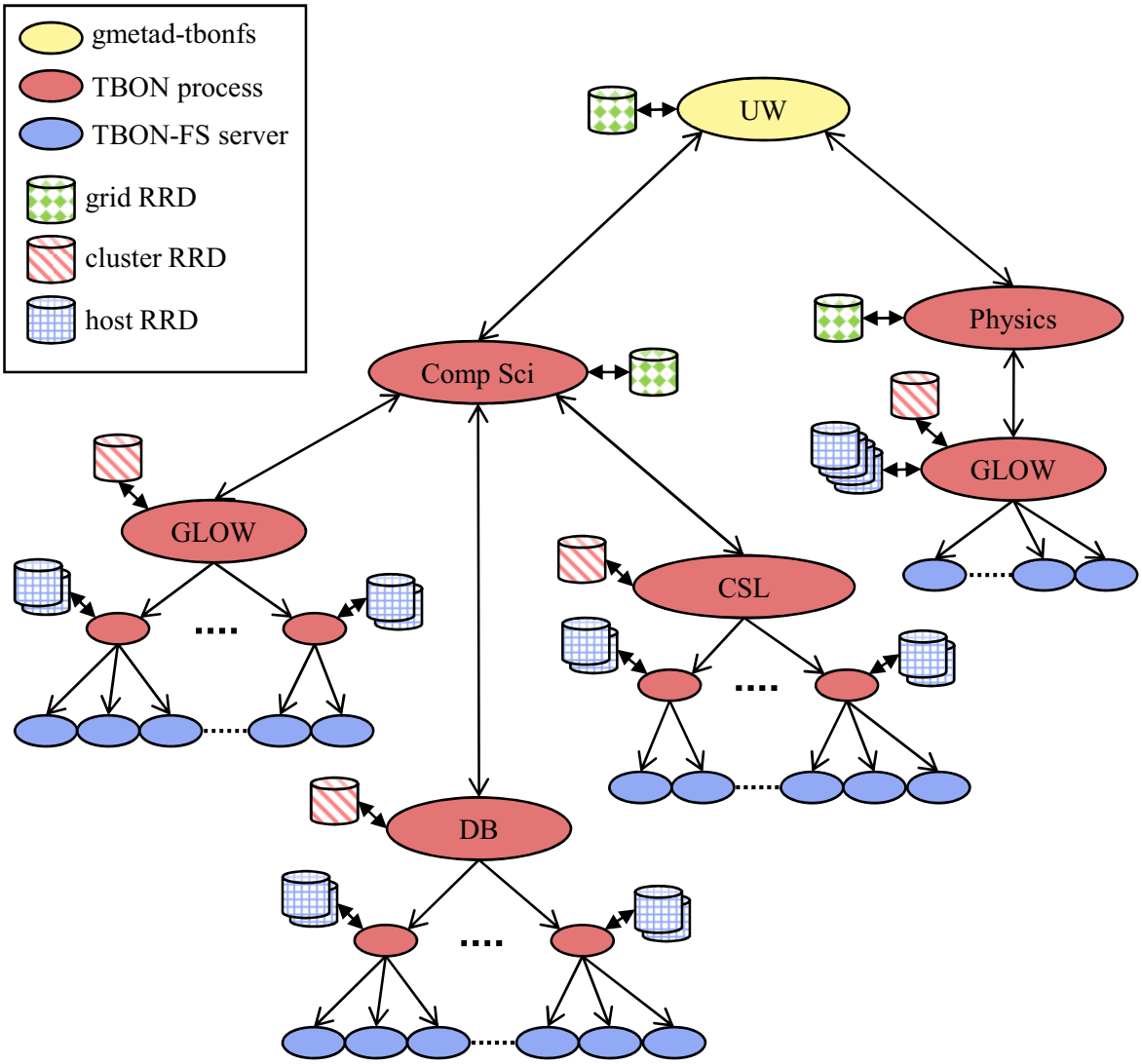

Figure 8.3 Ganglia-tbonfs Monitoring and Data Storage Architecture
The Ganglia-tbonfs architecture substitutes TBON-FS servers for all gmonds, and TBON internal processes for non-root gmetads. The root gmetad uses group file operations to collect metric data. Aggregations running within TBON processes are used to compute metric summaries for clusters and grids, and also to store data within RRDs. In contrast to the original Ganglia design, Ganglia-tbonfs allows for placing host metric RRDs at multiple hosts, which helps to parallelize the overhead of writing the metric data for each host.

When it is time to collect data for a collection group, the group files needed for each metric in the collection group are established and read. The aggregations used with group read operations extract the relevant data and then aggregate the per-host data into cluster and grid summaries. Because the compu-

tation to generate summaries is spread across the tree processes, the associated processing necessary at a gmetad is reduced. Further, we use structured binary data for metrics and metric summaries rather than the original XML encoding, and thus avoid the overhead of XML parsing.

To remove the bottleneck that occurs when a cluster gmetad stores metric data from all hosts in RRDs, we extended Ganglia-tbonfs to distribute the per-host metric RRDs across internal tree processes. Spreading the host RRDs across multiple processes helps to parallelize much of the iteration to store data. In the current implementation, all the internal processes at a pre-configured level in the tree store the host metric RRDs for the servers found in their respective sub-trees. Cluster and grid metric RRDs are still placed at the hosts running their associated gmetads.

Our distribution of host metric RRDs also required changes to the web-based user interface Ganglia provides for viewing host, cluster, and grid information. In the original design, a web server process is run on the host containing the RRDs, or the RRDs are placed on a shared file system accessible to the web server. This requirement results from the use of PHP scripts run by the web server to invoke `rrdtool`, which is the utility that reads the RRDs and generates the requested metric history graphs. In Ganglia-tbonfs, we cannot assume it is easy to run a web server on the hosts where internal tree processes are located, nor that those hosts have access to a file system shared with the web server host. Thus, we simplified the design to remove this RRD placement constraint.

To support our distributed RRDs, we replaced the logic that previously directly invoked `rrdtool` from the PHP scripts with queries to the root gmetad to obtain the requested graphs. When a query requests a graph whose data is found in remote RRDs, the root gmetad uses group file operations on a new synthetic file that is created at each monitored host. Group write operations on these files are used to multicast the query parameters. During these group writes, internal processes use special downstream aggregations that allow them to snoop on the write operands. Each internal process examines the write buffer to determine if it owns the RRDs containing the requested data. If so, it invokes `rrd-`

`tool` to generate the graph and store it within a memory buffer in the aggregation function's persistent state. After the group write of the query, the root gmetad performs a group read to obtain the generated graph. The same aggregation used with the group write is used with this group read, so that the internal process can retrieve the cached graph from the aggregation state and add the graph data to the result buffer. Since the data for any single query resides entirely within the RRDs of one internal tree process, only one process will add graph data to the result buffer during the group read.

8.3 Evaluation

Our evaluation compares Ganglia 3.0.4 to Ganglia-tbonfs to determine the performance and scalability effects resulting from incorporating group file operations and TBON-FS. Experiments were conducted on the Thunder cluster located at Lawrence Livermore National Laboratory, a 1024-node cluster using a Quadrics QsNetII Elan4 interconnect, with each node having four 1.4GHz Intel Itanium2 processors and 8GB of memory. Thunder limits each job to using up to 493 nodes.

Our experiments focus on measuring CPU and network utilization at the hosts running the cluster gmetad and monitors. As Thunder nodes do not have local disks, we were unable to test the benefits of distributing the RRDs across the internal tree processes. We ran both versions for thirty minutes using the default metric collection intervals.

We used `tbonfs_bench`, previously introduced in Chapter 8, to collect CPU utilization for all processes. To measure network utilization, we instrumented Ganglia to record total bytes sent or received, and used the built-in MRNet performance metrics that provide total bytes sent or received for Ganglia-tbonfs. When running Ganglia, we measured the performance of the gmetad for the cluster, the representative gmond, and the average performance of all non-representative gmonds. For Ganglia-tbonfs, we measured the cluster gmetad and the average performance of the TBON-FS servers that act as monitors. Each internal tree process ran on one of the monitored hosts. The TBON-FS topologies used during experiments at each scale are shown in Table 8.4.

Number of Monitored Hosts	TBON Topology (fan-out per tree level)
64	1 × 64
128	2 × 64
256	4 × 64
384	6 × 64
492	12 × 41

Table 8.4 Topologies used on Thunder

Figure 8.5 shows the observed CPU utilization by aggregator processes and monitor processes as we increase the number of monitored hosts. Figure 8.5(a) shows the CPU use by cluster aggregator processes. We see that Ganglia-tbonfs reduces the CPU use at the cluster gmetad by 50% versus Ganglia 3.0.4. This decrease is attributable to the use of distributed TBON computation for summarizing metrics for the cluster. Even after this decrease, the Ganglia-tbonfs gmetad is still using 50% of a CPU to store metric data for hosts and the cluster in RRDs. In the curve for the original Ganglia gmetad, we observe a peak in the utilization at 256 hosts, after which point the utilization decreases. We hypothesize that at this point the gmetad has reached its maximum rate of storing metric data to the RRDs.

Figure 8.5(b) shows the CPU use by monitoring processes in both versions. For the original Ganglia, we separate the performance for the representative gmond from the average of all non-representative gmonds. As expected, the CPU use at non-representative gmonds grows linearly with the number of monitored hosts as each process receives and stores metric data from all other hosts. We observe that the CPU use curve for representative gmond follows the same pattern as the gmetad. This behavior supports our hypothesis that the gmetad is busy storing data to RRDs, which results in reduced requests to the representative for metric data. In Ganglia-tbonfs, CPU use starts at roughly 0.4% for 64 hosts and decreases at the larger scales. Again, we believe this decrease is due to reduced requests for

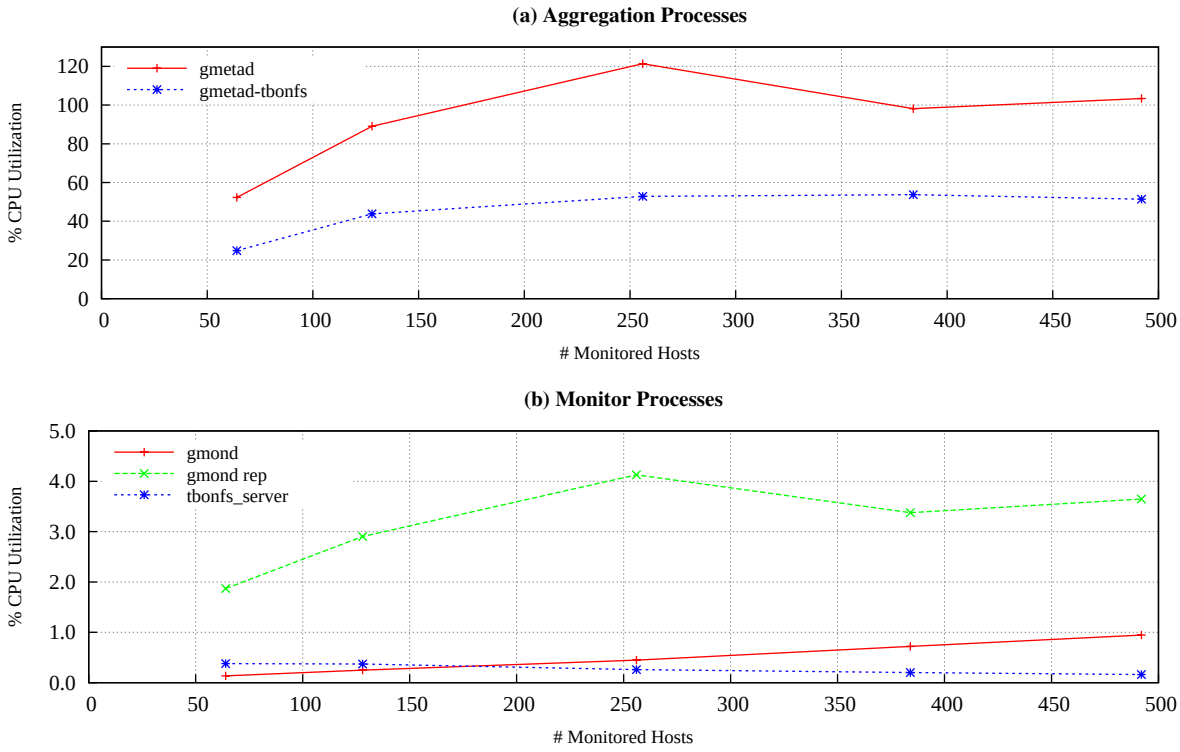

Figure 8.5 CPU Utilization for Ganglia Cluster Aggregators and Host Monitors
(a) Utilization by gmetads within Ganglia 3.0.4 and Ganglia-tbonfs.
(b) Utilization by gmonds in Ganglia 3.0.4 and TBON-FS servers in Ganglia-tbonfs. The 'gmond rep' curve shows the utilization by the representative gmond that sends metric data to the gmetad. The 'gmond' curve is the average of all non-representative gmonds.

metric data from the cluster gmetad as it stores more data in RRDs.

Figure 8.6 shows the network utililization due to reads and writes for aggregators and monitors. Figure 8.6(a) shows that Ganglia-tbonfs reduces the amount of data read at the cluster gmetad by an order of magnitude. It also shows the same peak observed for CPU utilization in the original gmetad. If the gmetad was not busy with storing data to RRDs, we would expect both the original gmetad and gmetad-tbonfs to show increased read rates as the size of the cluster grows. Figure 8.6(b) shows that the original gmetad has no measurable write activity. This graph also shows that the write rate of gmetad-tbonfs is directly tied to the first-level fan-out of the tree, with roughly 1KB of data written per child. All data written by gmetad-tbonfs corresponds to sending requests for reading metric data.

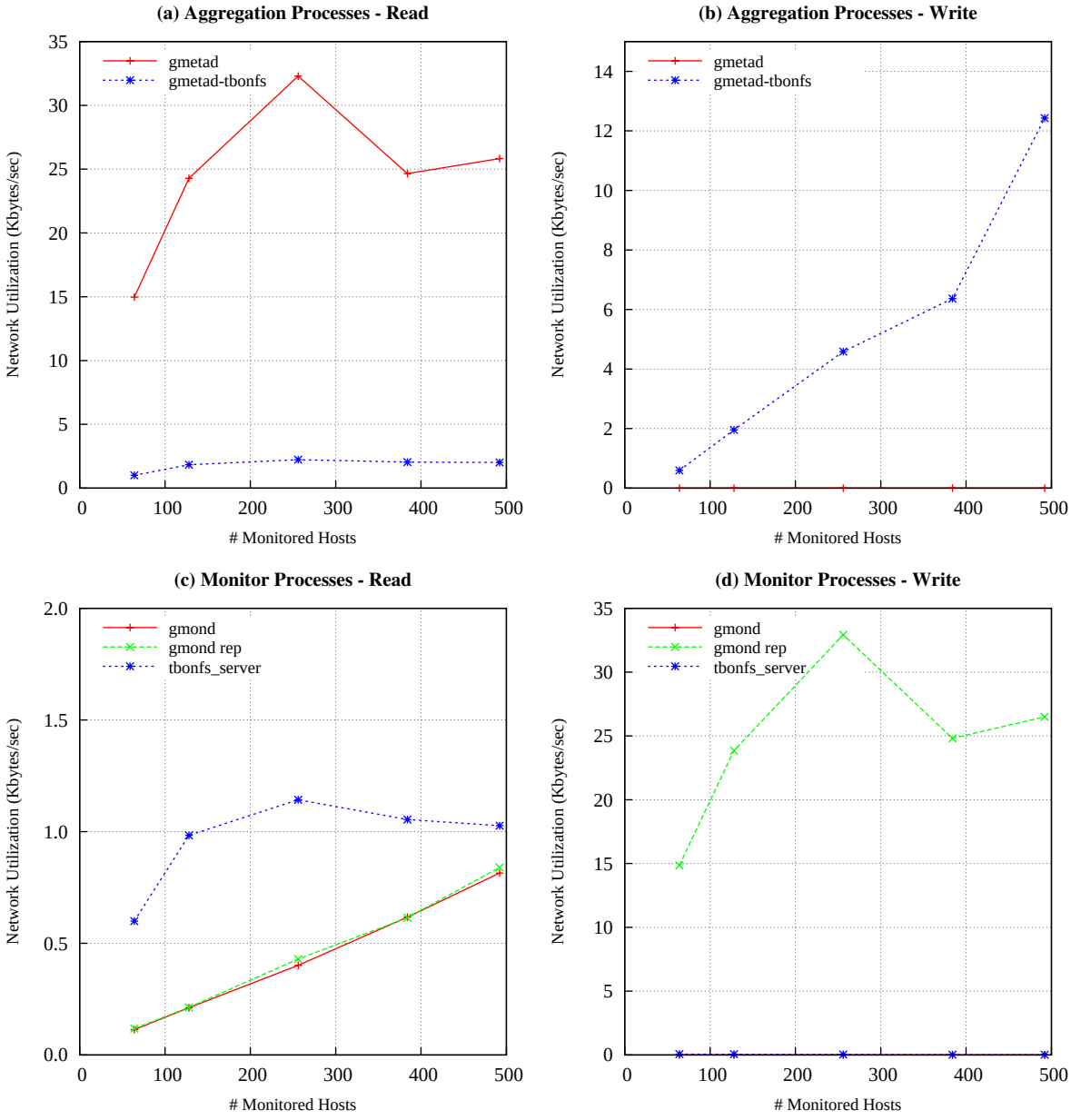

Figure 8.6 Network Utilization for Ganglia Cluster Aggregators and Host Monitors
(a) Utilization due to reads by gmetads within Ganglia 3.0.4 and Ganglia-tbonfs.
(b) Utilization due to writes by gmetads within Ganglia 3.0.4 and Ganglia-tbonfs.
(c) Utilization due to reads by gmonds in Ganglia 3.0.4 and TBON-FS servers in Ganglia-tbonfs.
(d) Utilization due to writes by gmonds in Ganglia 3.0.4 and TBON-FS servers in Ganglia-tbonfs.

The network utilization due to reads by monitoring processes is shown in Figure 8.6(c). As expected, the read rate increases linearly with cluster size for gmonds in the original Ganglia due to the use of IP multicast. Network reads by TBON-FS servers correspond to receiving requests for metric data from the gmetad, which arrive at a rate of approximately 1 KB per second.

The graph of Figure 8.6(d) shows data written by monitor processes. The representative gmond in the original Ganglia sends metric data at nearly the same rate as received at the cluster gmetad. In contrast, the non-representative gmonds and the TBON-FS servers send very little data due to the low-overhead metric collection rates.

8.4 Summary

With no previous knowledge of the Ganglia source code, our initial integration of group file operations and TBON-FS required just two weeks. The speed of the integration was helped by the clean and simple code architecture of Ganglia, but is also a testament to the simplicity of using the group file operation idiom. The group file abstraction helped to easily define metric collection groups, and to associate with each group custom data aggregations that are used to compute metric summaries for clusters and grids.

After this initial integration, we measured the performance of Ganglia-tbonfs on a local cluster. Our experiments at that time and those presented above revealed the severe bottleneck posed by storing data to RRDs at cluster gmetads. Based on this observation, we further developed Ganglia-tbonfs to distribute host RRDs across multiple hosts and eliminate this bottleneck.

Overall, we achieved good performance and scalability benefits from incorporating group file operations. We reduced the CPU and network utilization at cluster gmetad processes by half while stil maintaining the low overhead at monitored hosts. Given the low overhead of monitor processes in Ganglia-tbonfs, we believe the default metric collection rates could be increased to provide finer-grained monitoring data.

Chapter 9

Case Study: TotalView Debugger

Our final case study evaluates the integration of group file operations, TBON-FS, and proc++ within the TotalView parallel debugger. TotalView is a mature piece of software that represents the efforts of many developers over many years. It has a large code base with many functional modules that interact in complex ways. As a result, it is a challenge to understand the entire functionality of the debugger, even for its most senior developers. To successfully modify even a subset of its behavior requires careful analysis to identify the related code and its interactions with other parts of the system. TotalView has also been continuously optimized to improve performance, which lessens the probability of identifying simple changes that yield great benefits (often referred to as “low-hanging fruit”). Instead, performance and scalability gains are likely to require asymptotic improvements to the algorithms employed. TotalView thus represents an ideal study of the true effort required to apply our techniques to improve the scalability of a real tool.

TotalView has been considered the de-facto HPC parallel debugger for over twenty years due to its availability on the highest-performing computing systems of the day: from the Cray supercomputers of the 1980s and the Intel Paragon in the 1990s, through the rise of commodity-hardware clusters in the

Debugger Group Operation	Application Size (# processes)				
	1,728	2,744	4,096	8,000	10,648
Attach to processes	24.150	29.851	56.751	96.350	137.375
Stop processes	0.008	0.012	0.018	0.035	0.047
Single-step processes	0.453	0.812	1.262	2.417	3.335
Insert breakpoint	0.603	0.964	1.483	3.014	4.286
Remove breakpoint	0.077	0.125	0.191	0.402	0.591

Table 9.1 User Time (seconds) for Common Group Operations using TotalView 8.9.0

late 1990s and early 2000s, to today's large integrated IBM BlueGene and Cray systems containing hundreds of thousands of processors and specialized networks. As a result, TotalView has developed a large user base consisting of commercial, government, and academic institutions across the globe.

Despite its widespread use, TotalView has known scalability limitations in debugging very large parallel applications. From a user's perspective, these limitations result in common debugging operations having latencies that increase linearly with the size of the application, as shown in Table 9.1. The table shows the user-observed time for common group debugging operations such as attaching to a parallel application, stopping and stepping process groups, and managing process group breakpoints. TotalView version 8.9.0 (TV8.9), released in November of 2010, was used to collect the data shown in Table 9.1. This was the most recent production version available on the JaguarPF system at the time of our evaluation. Although TV8.9 is fairly responsive at these application sizes, the tasks exhibit latencies that grow linearly with the number of target processes. Given that current HPC systems support applications with hundreds of thousands of processes, and that users of interactive tools are rarely willing to wait for minutes at a time for an operation to complete, TotalView's operational latencies effectively reduce its use to a small fraction of a large system. Unfortunately, this represents a severe impediment to the analysis and debugging of many important parallel applications that only encounter problems at very large scales. Thus, TotalView is an ideal candidate for evaluating the performance

and scalability benefits of integrating group file operations, TBON-FS, and proc++.

In this chapter, we describe the four main activities of our TotalView study:

1. We identify the root causes of non-scalable behavior through measurment and analysis of the software architecture. Section 9.1 describes the architecture of TV8.9, which is used as the basis for our study, and the associated barriers to scalability.

2. We design and integrate solutions based on group file operations that improve the performance and scalability for a core subset of debugger operations on groups of processes and threads. To define the core operations, we assume a common debugging session wherein a user attaches to a parallel application, inserts and removes breakpoints, issues process group control commands, examines process memory and thread register contents, collects thread stack traces, and finally detaches or terminates the parallel application. Our solutions use group file operations on distributed proc++ files, and we have nicknamed the resulting research prototype TV++. We discuss the design of TV++ in Section 9.2.

3. We evaluate the performance of TV++ while debugging very large MPI applications with up to 150,000 processes. For comparison purposes, we use an unmodified version of TotalView (TV-orig) and a prototype of another scalable version of TotalView under development at Rogue Wave that directly integrates MRNet (TV-mrnet). Our evaluation presented in Section 9.3 confirms the scalable performance of group file operations on distributed proc++ files as previously seen in Chapter 6, but also reveals that significant barriers to the debugger's scalability still remain in the form of iterative actions within the client process during group operations.

4. Although a complete overhaul of the client was an untenable task for this study, we summarize the observed inefficiencies and propose remedies that leverage scalable group file operations. We also provide recommendations for improving scalability in areas that we did not directly address, such as presentation of information to users. Our recommendations are presented in Section 9.4.

9.1 Scalability Barriers in the TotalView Architecture

The TV8.9 architecture is shown in Figure 9.2. A user interacts with a debugger client process via a graphical or command-line interface. The client handles user requests for individual and group operations on targets (i.e., processes and threads). A debug server process is deployed on each host where targets are located. Debug servers are lightweight RPC servers that handle requests for control and inspection of specific targets from the client.

The client consists of three major layers of functionality. the *user interface (UI) layer* supports the process, thread and group abstractions exposed by the graphical (GUI) and command-line (CLI) interfaces, and provides debugging operations on these abstractions. The *debugger layer* maps the user-level abstractions to the machine-independent internal abstractions used to model execution and debugging status. This layer implements the individual and group operations provided by the UI layer by making requests of the *tracer layer*, which translates machine-independent requests into operations on specific targets. The tracer layer consists of set of tracer modules, one per remote host. Each module uses RPCs to access the OS process control layer within the debug server on the associated host.

There are three critical scalability barriers in the architecture of TV8.9. Briefly, these scalability barriers are: (1) a 1-to-N communication structure, (2) a design that employs a heavyweight client to keep servers lightweight, and (3) inconsistent treatment of group operations within the layers of the client and servers. We discuss each of these problems in more detail.

Communication between client and servers employs direct channels implemented using TCP sockets. TV8.9 thus has a 1-to-N communication structure that is typical of non-scalable tools, where the tool front-end (viz., the client) has individual communication channels to each tool server. For operations on target groups that are the common use case for a parallel debugger, this structure results in iterative behavior to send requests to servers and to gather operation results.

TV8.9 uses a heavyweight client design to keep debug servers lightweight. By heavyweight and

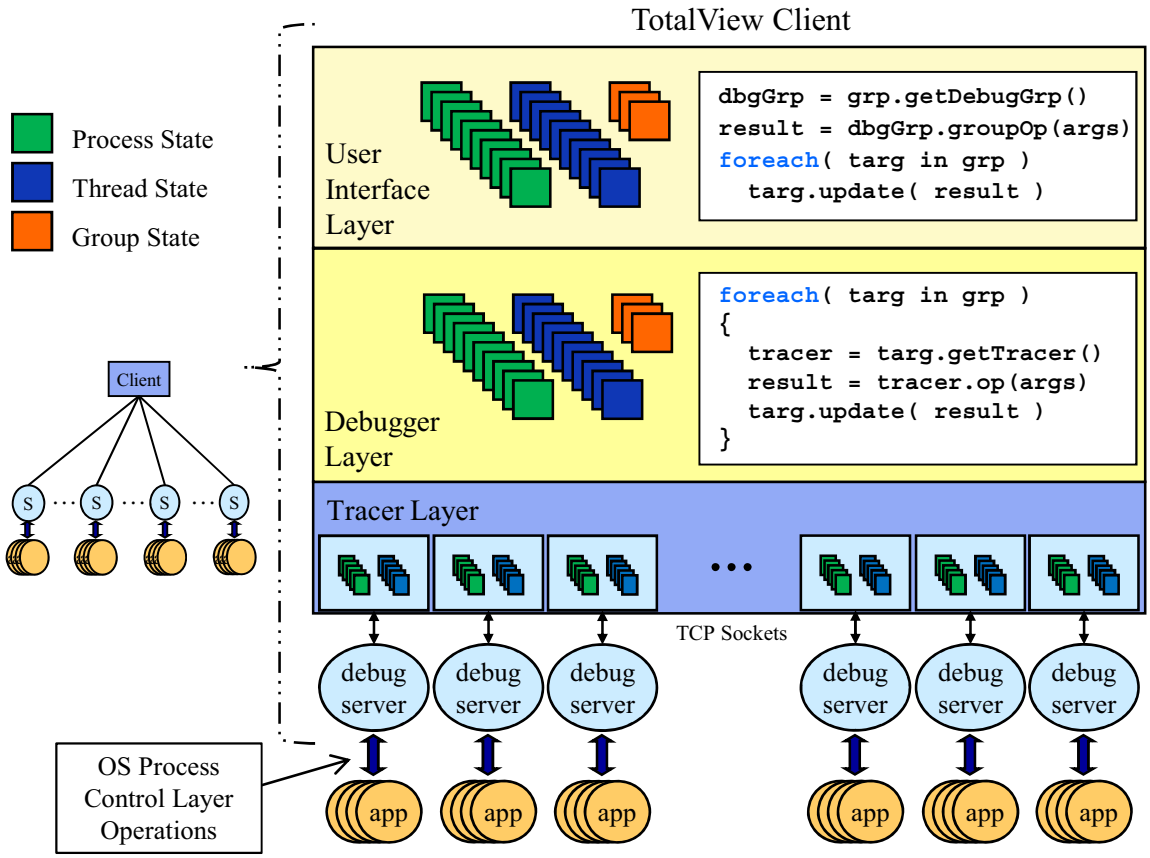

Figure 9.2 TotalView Client and Server Architecture
TotalView has a 1-to-N communication architecture that connects the client to a large number of debug servers. The server on each host containing targets uses the local OS process control layer to support requests from the client for operations on individual processes or threads. State for each process and thread is maintained at all three layers of the client, and group state is maintained at the top two layers. The code boxes within the top two layers show how debugger group operations initiated by a user are implemented. At the UI layer, operations on groups result in calls to group operations within the debugger layer. The debugger layer group operation iterates to call the appropriate tracer level operation for each member and update target state using the result returned by the tracer. The UI layer uses the result returned by the debugger layer group operation, which typically consists of a success or failure value, to update its state for each target.

lightweight, we are referring to the amount of local resource utilization necessary to provide the supported functionality. It is commonly expected that all target host resources will be available for use by parallel application processes. Thus, a lightweight server design has long been favored as a means for lessening the impact on the target application by using a minimal amount of local resources.

TV8.9 servers maintain little state about local targets, and thus use little memory. Further, servers

only use the processor when actively handling a request from the client, and return the results of OS process control layer operations directly to the client without additional processing. Unfortunately, by keeping servers lightweight, the resource utilization of the client is increased. The client maintains state for every host, process, and thread involved in the parallel application. This state is spread across all three layers, and consists of:

1. Execution status for every process and thread – This status denotes whether the target is running, stopped, or held. For processes, the status also includes information about the address space mappings (i.e., the regions of memory containing the code and data for the program executable and dependent libraries) and a list of the process' threads. Additional status information for stopped threads includes the contents of the general-purpose (GPR) registers and the stop reason (e.g., received a signal, or hit a breakpoint).
2. Debugging status for every process and thread – For processes, this status includes the breakpoints and watchpoints that have been applied and a cache of recently read memory locations. For threads, the debugger tracks intermediate states that occur when a thread transitions between the execution states seen by users. For example, intermediates states for transitions from a thread stopped at a breakpoint to running would include states such as restoring the original instruction, single-stepping, and replacing the breakpoint.
3. Host status – This status includes the host's name, IP address, and a set of identifiers indicating the targets that reside on the host.

Together, the execution and debugging status allow for determining the appropriate actions required when the user issues a command for debugging an individual or a group of targets, and provide the necessary context for generating the values for operands to OS process control layer operations.

Initialization and update of state for large numbers of targets requires iterative client actions, and the resulting memory footprint and processing overhead can be substantial. Additionally, because

tracer layer operations often use process- or thread-specific information such as virtual addresses, the client must calculate the correct address for each target before making a request. Such context-specific calculations lead to even more client overhead during group operations.

TV8.9 uses process and thread groups as a primary abstraction. By default, a user is presented with groups representing all processes, all threads, and processes running the same program executable. A user can also form custom process and thread groups. Debugging operations are applied to groups unless the user explicitly selects an individual process or thread target. Despite its focus on groups, the treatment of group operations at the various implementation layers is inconsistent. The most glaring omission is the lack of group operation support in the tracer layer, which forces the debugger layer to iterate in its use of the tracer layer and prevents the use of scalable mechanisms. This omission also hampers interactions with individual servers, since even operations involving many targets on the same server require a separate request and reply for each target. Conversely, some group operations, such as attaching the debugger to a group of processes, are implemented using iteration within the UI layer. Finally, because tracer layer operations are a thin veneer for OS process control operations, many debugging operations such as planting a breakpoint or gathering a thread stack trace require many RPCs between the client and each server. The reliance on OS-level debugging primitives precludes efficient implementations that reduce the number of client-server interactions during such debugging operations.

9.2 A Design for Scalable Group Debugging Operations in TV++

The design of TV++ addresses each of the three scalability barriers in the original architecture. TV++ replaces the existing 1-to-N communication structure and debug servers with TBON-FS. The debugging functionality previously provided by debug servers is supported by using TBON-FS to access the proc++ file service on each target host. The use of proc++ moves some of the burden of target state management to servers, and enables debugger group operations to be implemented using the

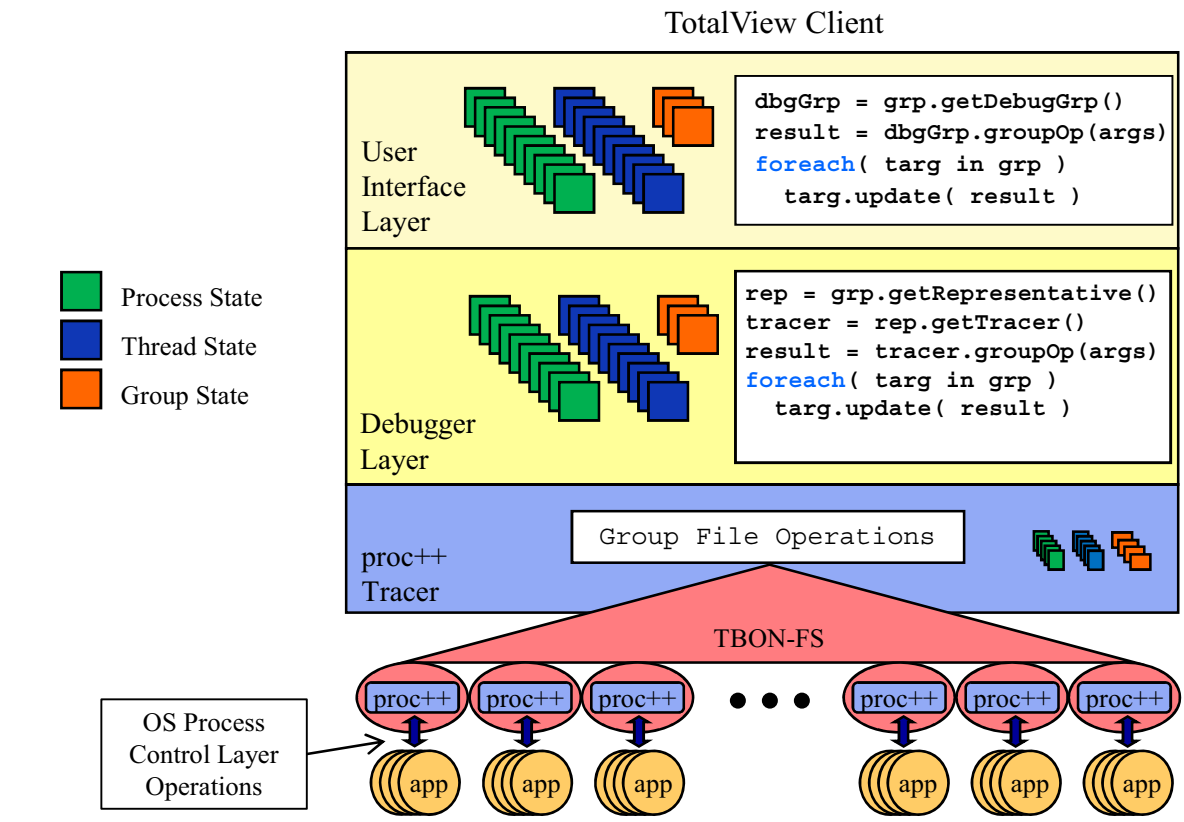

Figure 9.3 TV++ Scalable Architecture
TV++ replaces the existing 1-to-N communication architecture and debug servers with TBON-FS. The client debugger layer and tracer layer at the client have been updated to support new group operations. Within the proc++ tracer module, group file operations on remote proc++ files are used to implement the group debugging operations.

new abstractions proc++ provides to eliminate context sensitivity and reduce client-server interactions. The client design was also modified to support group operations at all layers, thereby allowing the use of group file operations for implementing tracer functionality at the lowest layer. An overview of the resulting TV++ architecture is shown in Figure 9.3. Below, we discuss the key design characteristics of TV++ and the relevant integration activities.

Since the tracer layer encapsulates both remote communication and low-level operating system debug interfaces, we decided to implement a new type of tracer module that would use TBON-FS to access servers that provide debugging support through the proc++ file service. We call this new tracer

the *proc++ tracer*. As is required for all tracer modules in TV8.9, the proc++ tracer maintains information about the remote hosts, processes, and threads under its control. The tracer additionally keeps information on process and thread groups. For each group, this information includes a set of active group file descriptors and the group handle that is used to uniquely identify a group and query its membership. The current proc++ tracer implementation supports operations only on the two most common groups, all processes and all threads, even though proc++ can support arbitrary custom groups. In practice, these two groups were sufficient to demonstrate the functionality required by our target use case.

To remedy the inconsistencies in the support for group operations within the layers of the client, and to enable the use of group file operations for implementing user-level debugger operations on process and thread groups, TV++ adds new group methods (as necessary to support the operations in our target use case). A list of our extensions follows.

- We added a new type of handle that represents a group of processes or threads. The debugger layer passes handles to the tracer layer to identify debug targets.
- Methods used to support attaching the debugger to a group of processes were added to the debugger and tracer layers. Two versions of group attach are supported by the proc++ tracer. The first version takes a list of the target processes to attach to as an input operand. The second version takes the path name of an executable and attaches to all processes on all the server hosts that are running that executable.
- Process group memory read and write operations were added at all layers. Two versions of these operations are supported. The first version operates on virtual memory address, while the second uses the context-insensitive base+offset abstraction provided by the proc++ image files.
- Process group breakpoint insertion and removal were added to the tracer layer. Previously, breakpoint management required sequences of individual address space reads and writes. The new group breakpoint operations utilize the explicit breakpoint support in proc++.

- Process group stop and continue operations were added to the tracer layer.
- Thread group single-step and step-from-breakpoint were added to the tracer layer.

When using the new group methods, any necessary changes to the client state for each group member are made in the same iterative manner as used by the existing software. Though much of this target state is made redundant by the move to using proc++, these updates were necessary to ensure the debugger continued to work as expected. Later in Section 9.4, we propose an approach to state management designed to improve the scalability of allocation and updates.

The proc++ tracer is currently the only tracer module that implements the new group methods. To avoid frequent opening and closing of oft-used group files, the proc++ tracer maintains a set of group file descriptors that are created during the process group attach operation and remain open until the processes are detached or killed. These descriptors correspond to groups of proc++ files used for controlling processes and threads, gathering thread events, and accessing memory and registers.

When servicing a group tracer operation, the tracer queries the target group's information object to see if there exists a cached group file descriptor for the relevant group file. If a cached descriptor is found, it is used to complete the required group file operations for the current tracer operation. Otherwise, the tracer uses `gopen` to establish the file group, and optionally binds the aggregation to be used for group reads. Often, a single group write or group read with default aggregation is all that is required. For these situations, the tracer implements convenience routines for group reads and writes that perform the entire sequence of group file operations (i.e., `gopen`, optional `lseek`, `read` or `write`, and `close`) for a group file whose directory path is given as an operand.

The proc++ tracer uses custom group read equivalence aggregations to provide scalable summaries of program counter register values, thread events, and process address space mappings. A simple equivalence aggregation is used to generate lists of threads that share the same PC register value. Threads are represented using their index within the group file used for accessing the `regs/pc` files in

proc++ thread directories. Strided ranges are used for compact thread lists. A sample of two PC equivalence classes representing 10,000 threads is shown below.

```
(0,1,5000) 0x5adc0

(5000,1,5000) 0x5adc8
```

An event read from the `events` file within a proc++ process directory is a tuple that identifies the eventing thread and the event details, and has the form:

```
((host_id, process_id, thread_id), event_type, event_data, event_addr)
```

The thread associated with the event is given as a 3-tuple containing the host, process, and thread IDs. The `event_type` indicates the reason a thread has stopped, such as due to a signal or hitting a breakpoint. The `event_data` field contains auxiliary data associated with specific event types, such as the signal number or breakpoint ID. The `event_addr` is the value of the PC register, which indicates the address where the thread is stopped. For event types such as process `fork` or `exec` or the loading of a shared library, the address is irrelevant, and `event_addr` is set to zero. To generate event summaries, the tracer uses an aggregation that associates a list of target threads with each unique event. Event uniqueness is determined by comparing all three fields of an event. An example of an event summary for 10,000 threads hitting the same breakpoint, whose ID is 4, is shown below.

```
((0,1920,1920),...,(9999,24365,24365)) (PROCPP_BREAKPOINT, 4, 0x5adc0)
```

To aggregate executable and library address space mappings read from the `as/map` files in proc++ process directories, equivalence classes are formed for images that are mapped at the same base address. Each mapping equivalence class associates a list of processes with the name and base address for an image. Process lists are represented as strided ranges of group file indices, similar to the PC equivalence aggregation. A few sample classes are shown below.

```
(0,1,10000) (myexe, 0x5ac00)

(0,1,10000) (mylib.so, 0x70c00)

(0,1,5000) (libc.so.6, 0x90a00)
```

```
(5000,1,5000) (libc.so.6, 0x90c00)
```

9.3 Evaluation

To evaluate the benefits of using group file operations and TBON-FS in TV++, we measured its performance for debugging an MPI application as we increased the number of processes. As in our evaluation of TBON-FS and proc++ in Chapter 6, we used the IRS benchmark from the ASC Sequoia Benchmark Codes [8]. The application was run in pure-MPI mode and one process was placed on each compute node core. IRS requires the total number of processes to be equal to k^3 for an integer $k \geq 2$.

For comparison, we measured the performance of two other versions of TotalView. The first version, which we refer to as TV-orig, is built from the same source code as TV++. Both TV-orig and TV++ are based upon a snapshot of the TotalView source code taken in February of 2011, a couple months after the release of TV8.9. The code that uses the proc++ tracer and the newly added group methods at the various client layers can be enabled at compile time.

The second version, which we refer to as TV-mrnet, is a prototype version under development at Rogue Wave that directly integrates MRNet (without using TBON-FS or proc++). This version is intended to support debugging of applications with up to 32,000 processes. MRNet is used to replace the existing TCP sockets to each debug server. To avoid extensive server changes, requests from the client and server responses use the existing protocol format. Similar to TV++, extensions were made to the tracer layer to support group operations and enable the use of MRNet's scalable multicast. However, some group debugger operations, including attaching to processes and process group continue, have not yet been converted to use group tracer operations. These operations still use iteration to unicast messages from the client to individual servers. TV-mrnet uses custom data aggregation for compact encoding of PC register values, but no other forms of data aggregation are employed.

All three versions were built on a Linux x86-64 host located at Rogue Wave using the GCC 3.3.3 compiler suite with optimization enabled and without debug information.

Experiments were run on the JaguarPF Cray XT5 system [77] located at Oak Ridge National Laboratory. JaguarPF contains 18,688 compute nodes, each with two six-core AMD Opteron 2435 processors and 16GB of DDR2-800 memory. Nodes are connected in a three-dimensional torus topology by a high-bandwidth, low-latency SeaStar 2+ network. For TV++, we used the same tree topologies as our previous proc++ experiments (see Table 6.7). Because large compute node allocations for interactive debugging are hard to acquire on JaguarPF, we employed the TVscript facility that permits runs of TV8.9 in batch environments. A TVscript is simply a sequence of CLI commands.

We measured performance using TV8.9's built-in performance metric system. This system provides the ability to record the wall time used by a region of sequential code; the region can be explicitly defined using start and end delimiters or implicitly associated with the scope of a function or block. At the end of TotalView's execution, or when explicitly requested by a script command, the performance summaries for recorded metrics are written to a file. For each metric, the performance summary includes the number of times the code region was executed, statistics such as minimum, maximum, and average latency computed from all executions, and a histogram showing the distribution of latencies. The results presented below correspond to the smallest value for average latency observed across a small number of runs at each application size.

A script provided by Rogue Wave for evaluating TotalView's performance when debugging the IRS application was used to drive our experiments. The IRS script runs the debugger through a sequence of eight subtests representing common group debugging actions, and measures the elapsed time for each subtest. The actions performed within each subtest are:

1. Launches the parallel application, attaches to all processes, and waits for all processes to come to a stable state. Time for this subtest can be divided into three primary activities: time to launch the parallel application, time to gather the list of processes, and the time to attach to all processes
2. Create a breakpoint in all processes. This subtest is used three times to insert breakpoints corre-

sponding to three source locations.

3. Run all processes to a breakpoint. This subtest is used three times to run to each of the breakpoints.
4. Step all processes past the first breakpoint.
5. Perform a *step* to run to the next source line on all processes. This subtest is used twice.
6. Remove a breakpoint from all processes.
7. Perform a *next* to run to the next source line without following function calls on all processes. This subtest is used twice.
8. Perform a *run-out* to exit the current function on all processes.

TV++ completely supports the actions of the IRS script, which confirms our choices for core functionality, and demonstrates that TV++ is suitable for basic debugging of real applications.

Our evaluation measured performance during two phases of executing the IRS script. The first phase, which we refer to as *parallel-startup*, corresponds to subtest 1. The second phase, which we call *group-operations*, corresponds to subtests 2 through 8. During each phase, we collected performance data at two levels of detail:

1. *Micro-level performance of proc++ tracer group operations* - We measured the time spent in the tracer group operations. These times include the use of group file operations and the management of tracer state for processes and threads. Measurements use newly added performance metrics for each group tracer operation. We also added metrics to some single-target tracer operations to help identify when they were being used unexpectedly. Results for micro-level measurements are presented in Section 9.3.1
2. *Macro-level performance of debugger group operations* - We measured the time taken by internal routines within the UI and debugger layers that implement group debugger operations initiated by users. For the majority of operations, TotalView already contained the necessary metrics. When more detail was needed to determine where time was being spent in a long running group opera-

tion, we added new metrics to provide the necessary insight. Results for macro-level measurements are presented in Section 9.3.2

9.3.1 Micro-level proc++ Tracer Performance

Figure 9.4 shows the time required by tracer group operations in three classes: (a) group operations used exclusively during parallel-startup, (b) group operations used for controlling the behavior of processes and threads, and (c) group operations used for inspecting the state of processes and threads.

During parallel-startup, the three proc++ tracer group operations shown in Figure 9.4(a) are used. The first operation, “launch servers”, includes the time spent to launch a debug server on each of the target hosts. Server launch occurs as part of the mount of TBON-FS by the proc++ tracer. The second operation, “attach processes”, includes the time to do a group attach to all the processes in the application, as well as the time to `gopen` the process and thread group files that are kept open for the duration of the application. The third operation, “gather address maps”, shows the time to gather and calculate equivalence classes for the address space mappings of the application executable and shared libraries.

The time to launch debug servers dominates the tracer group operations used during startup. On Cray systems, the ALPS parallel runtime environment [57] must be used to co-locate tool processes with application processes. Unfortunately, this means that there is little that can be done to eliminate the linearly increasing time behavior exhibited during server launch. In contrast, the time to attach processes scales fairly well even though the list of processes that is distributed to all servers grows with the application size. Gathering address space mappings for all processes shows sub-linear behavior, but still requires over ten seconds for more than 156,000 processes. We believe that further improvements to the internal data representations used for the mapping equivalence classes in our aggregation can greatly improve the time of encoding and decoding this data at all TBON processes. Related improvements in the format used to record mapping information at the client may also be beneficial.

Figure 9.4(b) shows the performance of proc++ tracer operations used for controlling process and

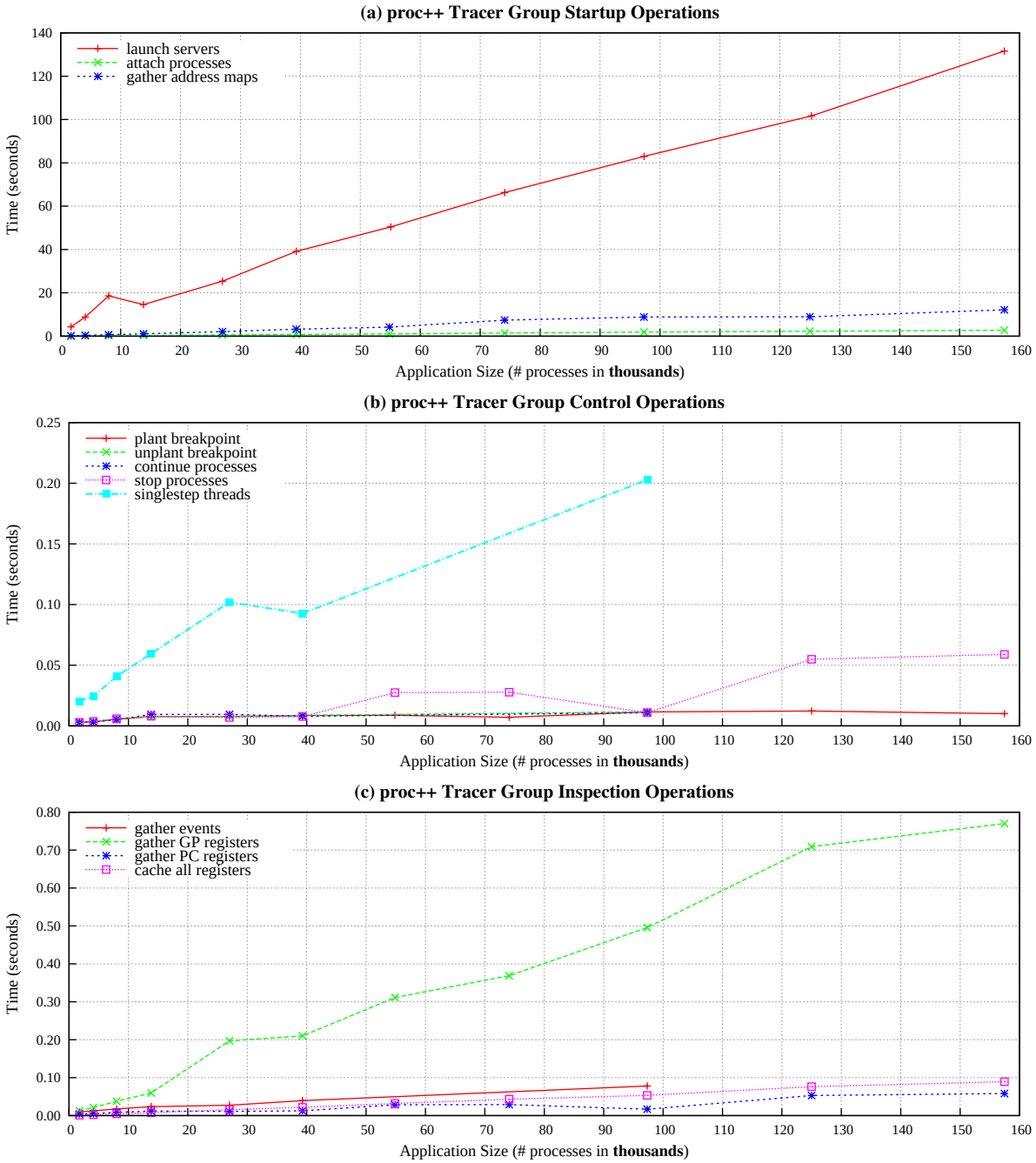

Figure 9.4 proc++ Tracer Group Operation Performance
(a) Latency of group operations used during parallel-startup.
(b) Latency of group control operations.
(c) Latency of group inspection operations.

thread groups. The tracer operations shown include planting and removing breakpoints, stopping and continuing processes, and single-stepping threads. Because they are based on scalable group writes, tracer operations on process groups exhibit excellent performance. Single-stepping a thread group requires both a group write and a group read to gather thread step events. The total latency for single-stepping is dominated by the linear time at the client to record the thread events.

The performance of tracer operations used for process and thread group inspection is shown in Figure 9.4(c). These operations include gathering events from all processes and GPR and PC register contents from all threads. We break out the time the client spends iterating to store individual register contents ("cache all registers") from the time to gather registers to clearly show its contribution. The inspection operations using equivalence aggregation ("gather events" and "gather PC registers") show sub-linear behavior. In contrast, collecting GPR contents using data concatenation scales linearly.

Overall, the performance of the proc++ tracer group operations match that observed during our previous proc++ evaluation. Thus, these results confirm the scalability benefits that can be gained when using group file operations on distributed proc++ files accessed via TBON-FS.

9.3.2 Macro-level Debugger Group Operation Performance

TV8.9 uses the system's parallel runtime environment (e.g., ALPS on Cray) to launch the parallel application, and then extracts the MPI process table from the runtime's starter process (e.g., `aprun` on Cray). The process table maps the host and process ID for each application process to its MPI rank. A rank is a logical identifier used to name processes for communication in MPI. TotalView then brings the processes under its control through a series of four steps. The latency of the main tasks within each step are shown in Figure 9.5 for each of the three versions of TotalView.

One of the most obvious observations to be made from Figure 9.5 is that TV-orig is limited in the number of application processes that can be debugged. In our experiments, the largest application size we could reliably debug using TV-orig was 11,520 processes (just 5% of JaguarPF's capacity). This

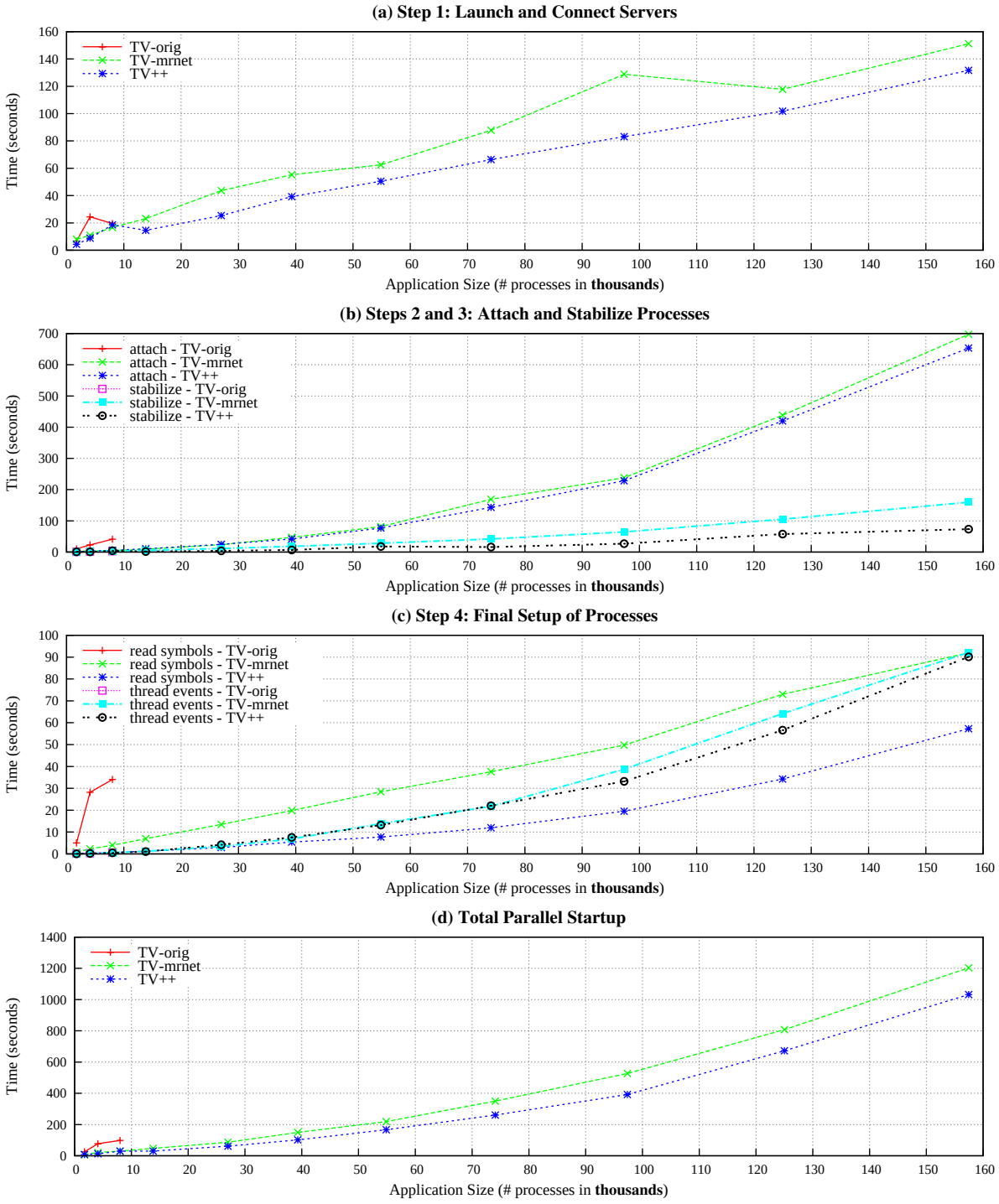

Figure 9.5 TotalView Parallel Startup - IRS

(a) Latency of launching and connecting debug servers (step 1).
(b) Latency of attaching to parallel application processes (step 2) and waiting for a stable state (step 3).
(c) Latency of reading symbol information for all processes and inserting event breakpoints (step 4).
(d) Cumulative latency for parallel startup.

limitation is due to the use of the `select` system call for monitoring socket connections to debug servers and the system limit for the size of the file descriptor sets that are used as its operands. On JaguarPF and most Linux systems, the descriptor set size is a compile-time constant in the operating system kernel, with 1,024 being the default value. Due to this system limit, TV-orig can only be used to debug applications using slightly less than 1,024 hosts (12,288 processes). This limit on the number of monitored file descriptors can be remedied by using `poll` instead of `select`, but this will not improve the performance at scale. TV++ and TV-mrnet do not have this limitation, due to their use of TBONs for communicating with servers.

In the first step, debug servers are launched on each participating host and communication between the client and servers is established. As shown in Figure 9.5(a), this phase has linear behavior that is primarily attributable to the time required to launch servers using ALPS. TV++ shows a relatively uniform reduction in latency of about 20 seconds versus TV-mrnet. This reduction is due to tuning that was done in TV++ to improve the overhead of initializing host state on the client .

During the second step, the client initializes some state for each process that will be attached, instructs and waits for each server to attach to local processes, and finally updates the state for all processes. The performance of this phase is shown by the “attach” curves in Figure 9.5(b). Both TV++ and TV-mrnet significantly reduce the time for attaching all processes versus TV-orig, due to the benefits of using a TBON to distribute attach requests. Unfortunately, the overall scaling behavior appears quadratic. We attribute this behavior to the need to initialize and update the process state at the client. The cost of state management is only magnified by the fact that process state is spread across all three layers of the client.

Events for processes stopped during attach and newly identified threads are gathered and processed by the client during the third step, and the parallel application is brought to a stable state. The performance for this phase is shown by the “stabilize” curves in Figure 9.5(b). TV++ outperforms TV-

mrnet and TV-orig during this phase by generating the process stop and thread creation events directly within the proc++ tracer during process attach, whereas the other two versions generate these events in the server tracers and must incur the cost of gathering them to the client. The TV++ curve thus is strictly the time spent at the client processing events, which can be quite substantial.

In the fourth step, symbol information for the program executable and dependent shared libraries is collected, and special breakpoints used for internal handling of process fork/exec events, shared library load and unload events, and thread creation and destruction events are inserted into all processes. Figure 9.5(c) shows the performance for these tasks. The "read symbols" curves include the time to read symbol information and insert breakpoints for handling fork/exec and library events. The "thread events" curves show the time required to insert breakpoints for user-level thread library events. Because proc++ automatically generates events for fork/exec and library load/unload, TV++ does not try to insert the internal breakpoints for these events, which results in a large time savings. The "thread events" time is roughly equivalent for TV++ and TV-mrnet, and represents the time to issue a series of memory inspection operations on each process to determine that no user-level thread package is present (since IRS was run without threading).

Figure 9.5(d) shows the cumulative latency of the four startup phases for each version. At the largest scale, improvements in TV++ result in approximately 200 seconds of savings, but the overall startup time is nearly twenty minutes due to all of the client iteration bottlenecks.

In the second phase of the IRS script (subtests 2-8), we measured the performance of group debugging operations used for process control. The group control operations tested are those employed by the IRS script subtests, which includes stop, continue, step, next, and run-out. We also measured the performance of creating and removing breakpoints. These measurements closely approximate the performance that a user would observe during interactive debugging.

Figure 9.6 shows the performance for the group control operations. For each operation, we show

the scaling behavior up to 100,000 processes in the graphs on the left, and magnified views on the right for up to 16,000 processes. The magnified views are useful for seeing the performance of TV-orig. As is evident from the left graphs, each group control operations exhibits linear or worse scaling behavior.

For group stop, shown in Figure 9.6(a, b), the TV++ and TV-mrnet latencies are reasonable at less than 600 and 500 milliseconds, respectively, for nearly 100,000 processes. However, as previously shown in Figure 9.4, the time to actually stop the remote processes using the proc++ tracer is less than 20 milliseconds. Since this is a user-directed group stop, the debugger ensures that all processes have stopped by waiting for stop events from all targets. Thus, the remaining client time is spent waiting for these events and updating process state for each received event. We do not know the exact reason why TV-orig and TV-mrnet outperform TV++ during group stop operations. We believe the cause may be related to the 100 millisecond timeout period used by the proc++ tracer while waiting for events. In retrospect, this timeout period is a poor choice for the low-latency Cray network.

Process group continue operations in TV8.9 are implemented using a weak stop followed by a continue on each process. A weak stop is a stop request that does not update the user visible process state, and is used to ensure that pending events are handled before continuing a process. For processes with threads currently stopped due to hitting a breakpoint, the threads are stepped over the breakpoint during the continue. Due to all of the work that is done for a group continue, it takes approximately ten times as long as a group stop as shown in Figure 9.6(c, d). A majority of this time is spent within the client iterating over target processes and their threads to determine what actions are necessary. In TV++, we have eliminated a large portion of this iteration by having proc++ automatically step threads over breakpoints when a process is continued. The result of this enhancement is a better slope for TV++ versus TV-mrnet. Still, the scaling behaviors for all versions are linear due to the iteration involved in processing events during the weak stop.

TV8.9 implements group step, next, and run-out using temporary breakpoints. A temporary break-

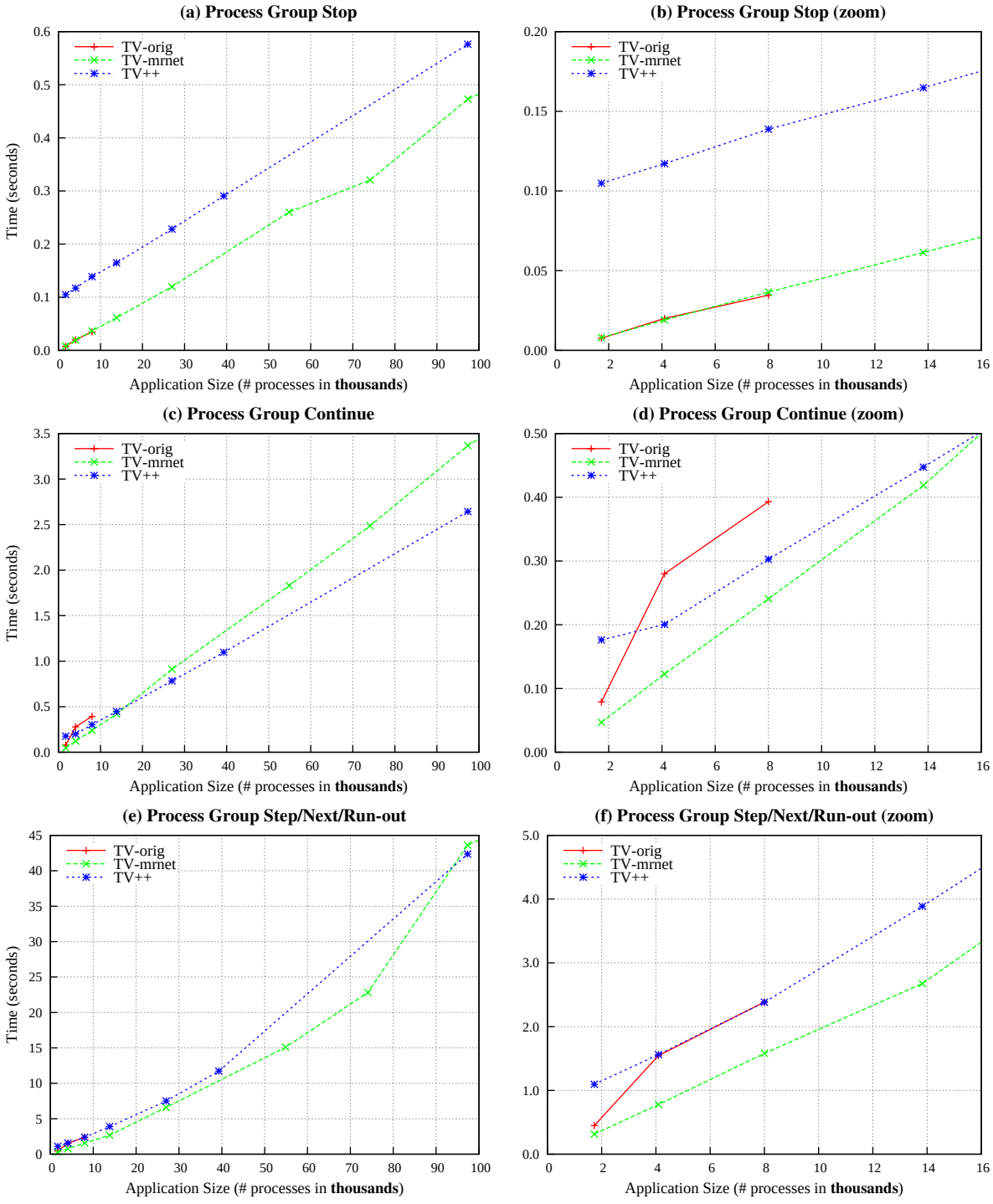

Figure 9.6 TotalView Process Group Control Operation Performance

(a, b) Latency of group stop operations.

(c, d) Latency of group continue operations.

(e, f) Latency of group step, next, and run-out operations.

point is placed at the address of the next machine instruction associated with the next program source line, and at the addresses of instructions whose control flow is not known in advance such as conditional jumps. Threads are then run to these breakpoints, and the debugger checks to see if the current program counters for all the process group's threads have reached the next source line. Threads that have reached the source line are held as the process is repeated for threads that have not. Instructions with unknown control flow are processed using machine instruction single-stepping to stop threads at the control flow target.

In our measurements all of these operations performed similarly, so we report only the results for the fastest group step, which involves the use of a single temporary breakpoint. The latency of this group step is shown in Figure 9.6(e, f). The performance of all three versions is dominated by client processing to analyze instructions and handle temporary breakpoint events, since the cost of creating, planting, and removing breakpoints is comparatively small, as we discuss next.

Figure 9.7(a) shows the latency to create breakpoints at source and address locations. Breakpoint creation is an activity performed strictly at the client, and simply iterates over the process group members to update their state. For source-level breakpoints, this client processing exhibits quadratic scaling behavior. Address-level breakpoint creation, as is used for temporary breakpoints, performs much better but still has linear scaling. At nearly 100,000 processes, creation of a source-level breakpoint takes over two minutes, while an address-level breakpoint requires just under ten seconds.

Figure 9.7(b) shows the time necessary to plant and remove breakpoints in process groups. In TV-orig and TV-mrnet, planting breakpoints requires fetching and storing the existing machine instructions at the target address and then writing the breakpoint instruction, and removing breakpoints writes back the original instruction. TV++ uses the explicit breakpoint support in proc++ that transparently handles the details of replacing instructions. TV++ thus outperforms TV-mrnet, but both still show linear scaling due to an iteration within the client to update the breakpoint state for each process.

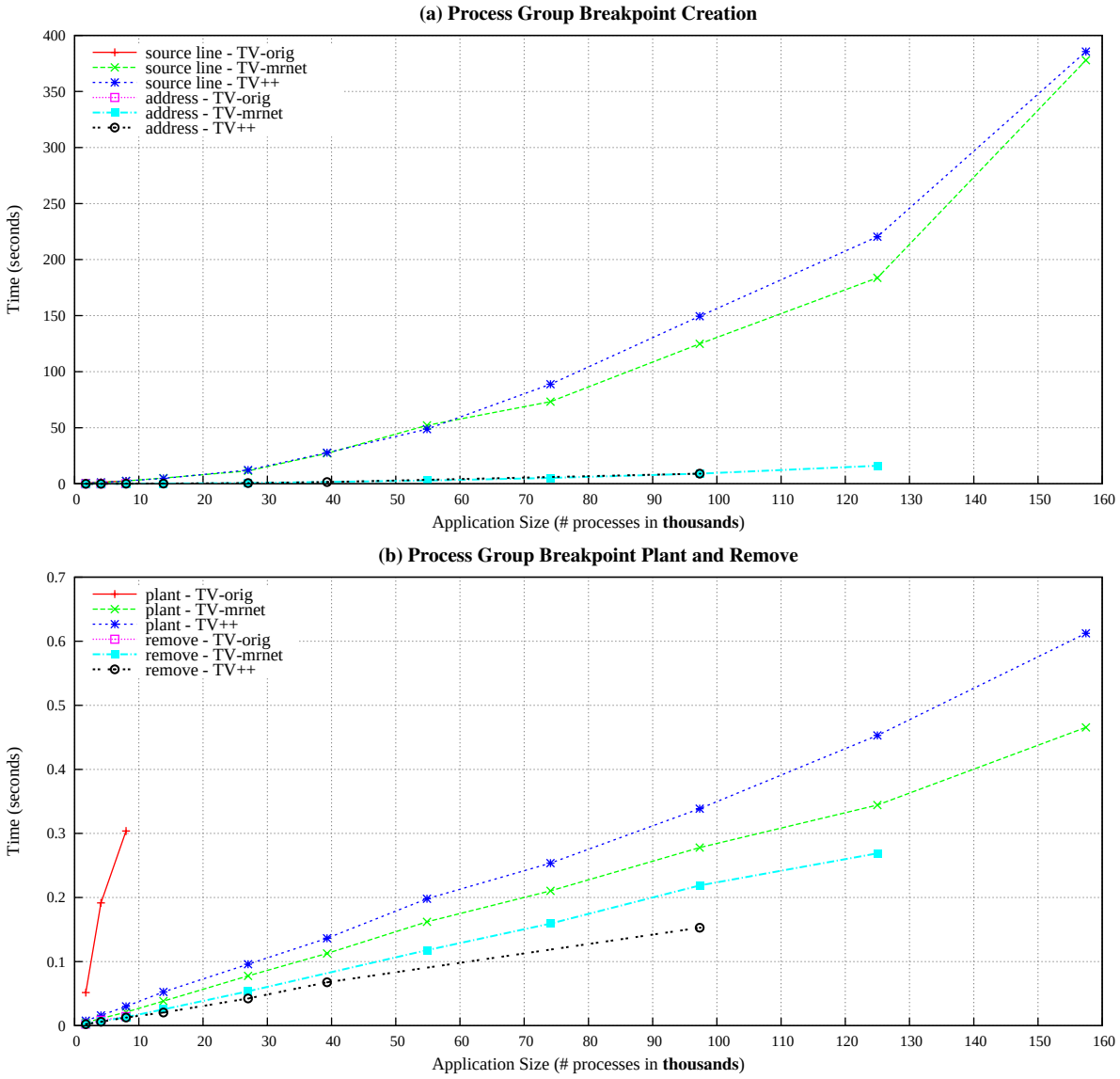

Figure 9.7 TotalView Process Group Breakpoint Performance
(a) Latency of creating a process group breakpoint at a specific source code line or memory address. Although hard to see, TV-orig performs identically to TV++ and TV-mrnet up to 8000 processes.
(b) Latency of planting and removing breakpoints in a group of remote processes.

9.4 Design Recommendations to Improve Scalability

Our evaluation of TV++ has revealed that the performance and scalability gains from integrating group file operations and TBON-FS can easily be hidden by iteration still present in the client. Thus, we can see that Amdahl's law [4], which generally states that benefit from parallelizing the work done

by an application is limited by the fraction of work that remains serial, also applies to the implementation of tools for distributed systems. In this section, we suggest approaches targeted towards reducing the iterative behavior in the client that limits its performance at scale.

Because the client maintains up-to-date status for every target process and thread, and target state is split across three layers, the client spends a large amount of time to initialize and update the state at each layer. There are two common causes for state updates, user-initiated debugging actions and processing of events. For user actions, the client typically updates state while it iterates to make tracer requests for each target. Updates are also applied during the serialized event processing at the client.

Completely eliminating these iterative behaviors in the client would require a substantial re-design of the management of target state. Ideally, target state would be maintained within debug servers and scalable group file operations would be used to collect and aggregate the state for groups of processes and threads. However, moving target state to servers also requires servers to be active in the decisions regarding the appropriate control or inspection tasks for a given debugging operation. For example, the current logic for process continue operations identifies threads at breakpoints that must be stepped over before issuing the continue request. TV++ can provide this functionality at servers since proc++ is breakpoint-aware. Given appropriate knowledge, servers could directly support operations such as source-level step or next and generation of stack traces. Unfortunately, such an overhaul would require a long development effort, which is a big investment for an established product with regular releases.

A stop-gap measure to improve state management would be the use of bulk allocation and updates. In bulk state management, per-target data structures (or objects) are replaced by a collection of arrays, with one array per field in the original structure. Figure 9.8 shows an example of converting a data structure containing three fields to a collection of three field arrays. Each target is assigned a unique index that is used with all field arrays. Using this data organization, a single large memory allocation per field replaces the iterative allocation of many small structures. A field array can be initialized or

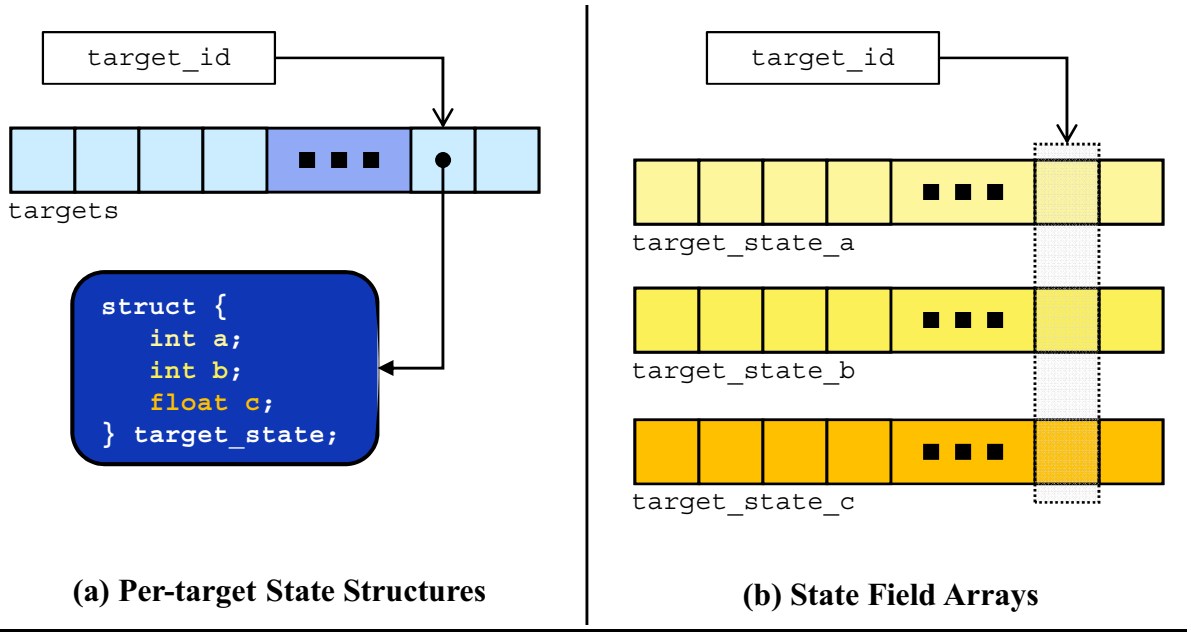

Figure 9.8 Conversion from Per-Target State to Bulk State
(a) The `target_state` structure contains three data fields. Pointers to instances of this structure are maintained for each target in `targets`. The unique ID for a target (`target_id`) is used to index within `targets`.
(b) Each of the three data fields of `target_state` are converted into arrays that hold the values for all targets. The unique ID for a target is used to index within each field array.

updated using a single large memory copy operation, preferably from a buffer generated using a scalable aggregation of remote target data.

A second recommendation for reducing the cost of state management is to avoid data caching. Such a notion may seem counterintuitive to designers of distributed systems, since caching is well-regarded for avoiding the cost of remote data access when data is not expected to change. However, caching for large numbers of targets results in huge memory costs and extra computation to store and invalidate the data for each target. TV8.9 caches the contents of memory and registers, but does not always use all of the cached data. When analyzing data for large groups, it may be much faster to use a scalable group file operation to fetch and aggregate data on demand rather than to iterate over cached data. An interesting area for future research is evaluating the benefits of caching data within TBON internal processes as a means for reducing the latency of on-demand fetching and the load at servers.

Finally, we have not addressed the scalability of displaying process and thread status information in the graphical or command-line user interfaces. It is our belief that attempts to provide per-target data for large groups result in unintelligible displays. The obvious alternative is to display information corresponding to groups of targets. The challenges for constructing such group displays are to identify the groups (i.e., sets of targets that share some characteristic) and to generate compact representations for the associated group membership and data. We believe group file operations could be used extensively for both group identification and generating compact data representations as the basis for scalable group visualizations.

9.5 Summary

The design and integration of group file operations, TBON-FS, and proc++ within TV8.9 was a long process that spanned two years, largely due to the learning curve required to understand a large, complex piece of software. This understanding was crucial to successful modification of the software. The most significant design and development time was spent adding the new group operations at the various client layers. Given these additions, it was fairly easy to use group file operations to implement the group operations. As shown in our micro-level proc++ tracer results, group file operations perform very well at large scales. Unfortunately, as revealed in our macro-level group operation results, the performance benefits are hidden by remaining serial behaviors in the client.

Chapter 10

Conclusion

Many tools and middleware perform group operations on distributed processes and files. Prior work has failed to provide a common solution that both addresses the key scalability barriers in distributed group operations and is easy to use within new and existing software. Our goal for this research was to develop a scalable, easy-to-use, and portable method for performing group operations on distributed processes and files. We desired a method that enables developers to quickly create new scalable software and that is easy to integrate within existing software to improve its scalability. To meet our scalability goals, this method had to employ techniques that are effective on the largest existing and upcoming systems, meaning systems containing at least tens of thousands of hosts and hundreds of thousands of processor cores. In this chapter, we review our technical contributions that meet these goals and discuss opportunities for further investigation and improvement.

10.1 Contributions

Our approach to group operations on distributed processes and files was designed to have three desirable properties: scalability, usability, and portability. We achieve scalability by avoiding linear-time behavior, such as iterations that are proportional to the size of the group, in favor of constant-time

or logarithmic-time behavior. Since data processing time often scales linearly with respect to the size of the data, our approach incorporates methods for reducing or limiting the amount of data that is gathered to a central location for further processing or analysis. To encourage widespread use and improve portability, we based our techniques on familiar and commonly available abstractions and functionality. When extensions to this common base were necessary to achieve scalability, we strove to retain clear semantics and intuitive behavior. Our approach consists of four main contributions:

Group File Operations. We introduced *group file operations*, an intuitive new idiom for eliminating iterative behavior when a single process must apply the same set of file operations to a group of related files. The keys to the idiom are explicit identification of file groups, the ability to name a file group as the target for POSIX I/O operations, and explicit semantics for aggregation of data and status results produced by these operations. Group file operations provide a familiar interface that eliminates forced iteration, thus enabling scalable implementations on groups of distributed files. Given underlying mechanisms for distributed data aggregation, group data and status results can be processed in a distributed fashion that eliminates or reduces the need for centralized analysis or large data storage.

Name Space Composition. Forming file groups efficiently is an important precursor to the use of group file operations. To avoid iteration during group definition, we proposed a scalable technique for defining file groups that relies on composing file system name spaces. Name space composition is supported using a new specification language called *FINAL*, the **FI**le **N**ame space **A**ggregation **L**anguage. FINAL models hierarchical name space composition as operations on trees, and provides flexible composition semantics based on common tree operations such as copying, pruning, and grafting. To support efficient composition of many trees without explicit iteration, FINAL introduces a merge operation over set of trees, and supports customizable resolution for the name conflicts that can occur during a merge. FINAL's merge operation provides the key semantics required for composing directories containing files from independent name spaces. Further, the merge operation maps nicely to scal-

able distributed composition within a tree-based overlay network.

TBON File System. We designed and evaluated *the TBON File System* (TBON-FS*)*, a group file system that leverages the logarithmic communication and distributed data aggregation capabilities of tree-based overlay networks to support scalable group file operations on distributed files and scalable global name space composition. TBON-FS uses MRNet [101], a general-purpose TBON API and infrastructure, for scalable communication of file system operation requests to thousands of independent servers, scalable aggregation of group data and status results from group file operations, and scalable name space composition. TBON-FS gives its single client the ability to construct a private, global view of distributed name spaces that is tailored for group file operations. Using synthetic file services loaded into TBON-FS servers, a client can further extend its use of scalable group file operations to non-file resources. Because TBON-FS is implemented entirely at user-level, it is easy to deploy on a wide variety of distributed systems. Since TBON-FS executes from a single user context, it can rely on existing authentication and file system access control mechanisms.

proc++. We designed and evaluated *proc++*, a new synthetic file system tailored for use in scalable control and inspection of distributed process and thread groups. Similar to existing proc file systems, proc++ provides files that can be used for control and inspection of processes and threads. The primary challenges in designing proc++ were to identify the tool behaviors that were hindered by interactions with existing process control interfaces, and to develop appropriate abstractions that enable group file operations to take advantage of newly provided tool-level capabilities. Abstractions were carefully designed to place functionality at the appropriate layer of the distributed tool hierarchy, which enables the parallelism needed to scale. The resulting abstractions are context-insensitive and naturally encapsulate interaction intensive operations, such as breakpoint management, as a means to reducing the number of interactions between the tool and the process control layer.

We combined these four research contributions to form a scalable platform for group operations on

distributed processes and files. We have demonstrated the performance and scalability benefits of this platform in several new tools for distributed system administration and monitoring and two existing, widely-used software packages. Our experimental results show this platform provides excellent performance for operations on groups with over 200,000 distributed processes (or files). At this scale, group file operations can achieve latencies on the order of 250 milliseconds, which is suitable for interactive tool and middleware tasks. Further, when using scalable data aggregations such as equivalence classes or simple statistical summaries, our results project that group file operations on groups with over one million members should complete in under one second.

10.2 Future Directions

Throughout this dissertation, we have identified future research and development activities that provide opportunities for improving the performance, usefulness, and applicability of our approach to scalable group operations on distributed processes and files. We briefly review the most important activities for group file operations and TBON-FS, and then discuss additional avenues for fruitful investigation.

By extending familiar POSIX I/O operations to produce the group file operation idiom, we believe the learning curve for adoption within new or existing software is significantly reduced. Still, we have identified cases where strict adherence to the existing interfaces and our attempts to mirror the utilitarian style of POSIX in the new operation interfaces results in less than ideal interactions. A study of the inefficiencies in the interfaces of group file operations could help to resolve such problems.

TBON-FS is currently limited to a single type of merge composition that places distributed files that have the same path into a single directory in the global name space. This composition is key to creating the directories that define the membership of group files. More general support for the semantics of the FINAL merge composition requires investigating techniques for caching portions of the composed global name space within the TBON and associated cache invalidation strategies to handle

dynamic server name spaces.

The current design of TBON-FS provides limited fault tolerance, as only failures of internal tree processes are properly handled. Since TBON-FS server process (or host) failures are expected to be common on next generation HPC systems that contain millions of components, and network and host failures are already prevalent in wide-area systems and data centers containing commodity clusters, a thorough examination of the survivability of TBON-FS in the presence of server failures is needed. In particular, we need to study techniques for automatic server restart after failures.

The error semantics of group file operations could also be improved to handle less reliable systems. For example, group file operations are synchronous and will not complete until every group member has finished the requested operation. During periods of temporary network disconnection or extreme server host load, a single member can delay the entire group operation for an extended period of time. We plan to investigate adding customizable timeout semantics to group file operations that would allow for returning partial group results from the non-delayed members.

Our case studies have focused on tools and middleware that needed low latency group operations and performed relatively small data accesses. There are many other classes of distributed operations on groups of files that do not have these properties. For example, large-scale data analysis as found in Map-Reduce style computations or data mining and clustering is used to process huge amounts of data and does not require instantaneous results. Further, analysis results are often too large to be held within the memory of a single client system, which would seem to preclude the use of the group file operations for large-scale data analysis. We would like to study common analyses performed in these systems to identify potential use cases that could be accomplished using group file operations and TBON-FS. Analyses that produce summarized information that is small in size relative to the input data sets are obvious targets, and approaches that analyze data in a pipelined, streaming fashion may also be applicable. Similar to our Ganglia effort, we may be able to use TBON processes to store large analysis

results on disks across many hosts to further reduce the client's burden.

One could also envision building a scalable key-value storage system using group file operations and TBON-FS. A hierarchical key space represented as file paths would be easily supported. Group write operations could be used to automatically replicate and distribute key-value tuples across many servers. Group read operations could quickly satisfy requests for specific keys, ranges of contiguous keys, and even searches over the values of all keys. Current distributed key-value stores are designed to support multiple clients reading and writing data simultaneously, and as such they provide rigorous consistency properties. To support multiple storage clients that may reside on many hosts, new semantics and a design for sharing both the global name space and group file descriptors will likely be required. In particular, mechanisms for ordering group file operations among clients will be crucial to ensuring consistency, as will associated strategies for fault-tolerance during replication.

References

[1] "Akamai Edge Platform: Application Acceleration that Delivers Content and Applications Quickly, Reliably, and Securely", http://www.akamai.com/html/technology/edgeplatform.html, January 2012.

[2] Allinea Software, "Allinea DDT - the debugging tool for parallel computing", http://www.allinea.com/products/ddt, December 2011.

[3] Alexander Ames, Nikhil Bobb, Scott A. Brand, Adam Hiat, Carlos Maltzahn, Ethan L. Miller, Alisa Neeman, Deepa Tuteja, "Richer File System Metadata Using Links and Attributes", *Mass Storage Systems Technologies (MSST2005)*, April 2005.

[4] Gene Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", *AFIPS Conference Proceedings* **30**, 1967, pp**.** 483-485.

[5] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory Lee, Barton P. Miller, and Martin Schulz, "Stack Trace Analysis for Large Scale Applications", *International Parallel & Distributed Processing Symposium*, March 2007.

[6] Dorian C. Arnold, Gary D. Pack and Barton P. Miller, "Tree-based Overlay Networks for Scalable Applications", *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2006)*, April 2006.

[7] Dorian C. Arnold and Barton P. Miller, "Scalable Failure Recovery for High-performance Data Aggregation", *International Parallel and Distributed Processing Symposium*, April 2010.

[8] "ASC Sequoia Benchmark Codes", https://asc.llnl.gov/sequoia/benchmarks/.

[9] Susanne M. Balle, John Bishop, David LaFrance-Linden, and Howard Rifkin, "Ygdrasil: Aggregator Network Toolkit for Large Scale Systems and the Grid", *Para 2004 Workshop on State-of-the-Art in Scientific Computing*, June 2004. Appears as *Lecture Notes in Computer Science* **3732**, J. Dongarra et al. (Eds.), Springer-Verlag, Berlin/Heidelberg, Germany, 2006, pp.207-216.

[10] Mohammad Banikazemi, David Daly, and Bulent Abali, "Sysman: A Virtual File System for Managing Clusters", *22nd Large Installation System Administration Conf. (LISA '08)*, pp. 167-174, November 2008.

[11] Amnon Barak and Oren La'adan, "The MOSIX multicomputer operating system for high performance cluster computing", *Future Generation Computer Systems* **13**, 4-5, March 1998, pp. 361-372.

[12] S. Bhatkar, D. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits", *12th USENIX Sec. Symp.*, August 2003, pp. 105–120.

[13] Andrew D. Birrell, Andy Higsen, Chuck Jerian, Timothy Mann, and Garret Swart, “The Echo Distributed File System”, Research Report 111, Digital Equipment Corporation, September 1993.

[14] Andrew D. Birrell and Bruce Jay Nelson, “Implementing Remote Procedure Calls”, *ACM Transactions on Computing Systems* **2**, 1, February 1984, pp. 39-59.

[15] Daniel P. Bovet and Marco Cesati, **Understanding the Linux Kernel (2nd ed.)**, O’Reilly and Associates, Inc., 2003, ISBN 0-596-00213-0.

[16] D. Breazeal, K. Callaghan, and W.D. Smith, “IPD: A Debugger for Parallel Heterogeneous Systems”, *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 216-218, May 1991.

[17] Ron Brightwell and Lee Ann Fisk, “Scalable Parallel Application Launch on Cplant”, *ACM/IEEE SC 2001 (SC ‘01)*, November 2001.

[18] Michael Brim, Ray Flanery, Al Geist, Brian Luethke, and Stephen L. Scott, “Cluster command & control (c3) tool suite”, *Parallel and Distributed Computing Practices* **4**, 4, December 2001, Nova Science Publishers, pp. 381-399.

[19] Michael J. Brim and Barton P. Miller, “Group File Operations for Scalable Tools and Middleware”, *16th Intl. Conf. on High-Performance Computing (HiPC 2009)*, December 2009.

[20] Michael J. Brim, Barton P. Miller, and Vic Zandy, “FINAL: Flexible and Scalable Composition of File System Name Spaces”, *Intl. Workshop on Runtime and Operating Systems for Supercomputers 2011 (ROSS'11)*, May 2011.

[21] D.R. Brownbridge, L.F. Marshall, and B. Randell, “The Newcastle Connection or UNIXes of the World Unite!”, *Software-Practice and Experience* **12**, 1982, pp. 1147-1162.

[22] Mark Burgess, “A Site Configuration Engine”, *USENIX Computing Systems* **8**, 3, 1995.

[23] Ralph Butler, William Gropp, and Ewing Lusk, “A Scalable Process Management Environment for Parallel Programs”, *Recent Advances in Parallel Virtual Machine and Message Passing Interface - Lecture Notes in Computer Science* **1908**, Springer, September 2000, pp. 168-175.

[24] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur, “PVFS: A Parallel File System for Linux Clusters”, *4th Annual Linux Showcase and Conference*, pp. 313-327, October 2000.

[25] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong, “Freenet: A distributed anonymous information storage and retrieval system”, *Workshop on Design Issues in Anonymity and Unobservability*, pp. 311-320, July 2000.

[26] Clip2 and The Gnutella Developer Forum, “The Annotated Gnutella Protocol Specification v0.4”, http://rfc-gnutella.sourceforge.net/developer/stable/index.html, January 2012.

[27] “CLOC -- Count Lines of Code”, http://cloc.sourceforge.net/, October 2011.

[28] “Clumon Performance Monitor”, National Center for Supercomputing Applications (NCSA), http://clumon.ncsa.uiuc.edu/doc-info.html.

[29] "ClusterIt", http://www.garbled.net/clusterit.html, January 2012.

[30] Peter F. Corbett and Dror G. Feitelson, "The Vesta parallel file system", *ACM Transactions on Computer Systems (TOCS)* **14**, 3, August 1996, pp. 225-264.

[31] Timothy W. Curry, "Profiling and tracing dynamic library usage via interposition", *USENIX Summer 1994 Technical Conference*, June 1994.

[32] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS", *SIGOPS Oper. Sys. Rev.* **35**, 5, December 2001, pp. 202-215.

[33] R.C. Daley and P.G. Neumann, "A General-Purpose File System for Secondary Storage", *Proceedings of the November 30--December 1, 1965, fall joint computer conference, part I (AFIPS '65)*, pp. 213-229, December 1965.

[34] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", *6th Symposium on Operating Systems Design and Implementation*, December 2004.

[35] Narayan Desai, Rick Bradshaw, Andrew Lusk, and Ewing Lusk, "MPI Cluster System Software", *11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI 2004)*, September 2004. Appears as *Lecture Notes in Computer Science* **3241**, D. Kranzlmüeller et al. (Eds.), Springer-Verlag, Berlin/Heidelberg, Germany, September 2004, pp.277-286.

[36] Narayan Desai, Andrew Lusk, Rick Bradshaw, and Ewing Lusk, "MPISH: A Parallel Shell for MPI Programs", *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, April 2005.

[37] DOE Exascale Initiative, "A DOE Laboratory plan for providing exascale applications and technologies for critical DOE mission needs", July 2010. http://computing.ornl.gov/workshops/scidac2010/presentations/r_stevens.pdf.

[38] European Exascale Software Initiative, "Investigation Report on Existing HPC Initiatives", September 2010. http://www.eesi-project.eu/pages/menu/publications/investigation-of-hpc-initiatives.php.

[39] D.A. Evensky, A.C. Gentile, L.J. Camp, and R.C. Armstrong, "Lilith: Scalable Execution of User Code for Distributed Computing", *6th IEEE International Symposium on High Performance Distributed Computing (HPDC '97)*, pp. 305-314, August 1997.

[40] R. Faulkner and R. Gomes, "The Process File System and Process Model in UNIX System V", *USENIX*, Dallas, Texas, January 1991.

[41] Joan M. Francioni and Cherri M. Pancake, "High Performance Debugging Standards Effort", *Scientific Programming* **8**, 2000, pp. 95-108.

[42] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr., "Semantic file systems", *13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pp. 16-25, October 1991.

[43] David K. Gifford, Roger M. Needham, and Michael D. Schroeder, "The Cedar File System", *Communications of the ACM* **31**, 3, March 1988, pp. 288-298.

[44] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System", *19th ACM Symposium on Operating Systems Principles*, October 2003.

[45] Andrzej Goscinski, Mickael Hobbs, and Jackie Silcock, “GENESIS: an efficient, transparent and easy to use cluster operating system”, *Parallel Computing* **28,** 4, April 2002, pp. 557-606.

[46] William Gropp and Ewing Lusk, "Scalable Unix Tools on Parallel Processors", *Scalable High-Performance Computing Conference*, pp. 56-62, May 1994.

[47] S. Hansen and E. Atkins, “Automated System Monitoring and Notification With Swatch”, *7th USENIX Conference on System Administration*, pp.145-152, November 1993.

[48] Erik Hendriks, “BProc: The Beowulf distributed process space”, *2002 International Conference on Supercomputing*, pp. 129-136, June 2002.

[49] Robert Hood, “The p2d2 Project: Building a Portable Distributed Debugger”, *ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT ‘96)*, May 1996.

[50] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, “Scale and Performance in a Distributed File System”, *ACM Transactions on Computer Systems* **6**, 1, February 1988.

[51] James V. Huber, Jr., Christopher L. Elford, Danial A. Reed, Andrew A. Chien, and David S. Blumenthal, “PPFS: A High Performance Portable Parallel File System”, *9th ACM International Conference on Supercomputing*, pp. 385-394, July 1995.

[52] IBM Corporation, “IBM LoadLeveler: User’s Guide”, September 1993.

[53] IBM Corporation, “Parallel System Support Programs for AIX: Command and Technical Reference, Volume 1, Version 3 Release 5”, May 2003.

[54] Sitaram Iyer, Antony Rowstron, and Peter Druschel, “Squirrel: A decentralized peer-to-peer web cache”, *PODC 2002*, July 2002.

[55] Jasmina Jancic, Christian Poellabauer, Karsten Schwan, Matthew Wolf, and Neil Bright, “dproc - Extensible Run-Time Resource Monitoring for Cluster Applications”, *International Conference on Computational Science (ICCS 2002)*, April 2002. Appears as *Lecture Notes in Computer Science* **2330**, P.M.A. Sloot et al. (Eds.), Springer, Berlin/Heidelberg, Germany, 2002.

[56] “June 2011 | TOP500 Supercomputing Sites”, http://top500.org/lists/2011/06, June 2011.

[57] M. Karo, R. Lagerstrom, M. Kohnke, and C. Albing, “The Application Level Placement Scheduler”, *Cray User Group*, May 2006.

[58] T. J. Killian, ‘‘Processes as Files,’’ *USENIX Summer Conference Proceedings*, June 1984.

[59] John Kubiatowicz et al., “OceanStore: An Architecture for Global-Scale Persistent Storage”, *ASPLOS 2000*, November 2000.

[60] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz, and Ben Liblit, "Lessons Learned at 208K: Towards Debugging Millions of Cores", *Supercomputing 2008 (SC’08)*, November 2008.

[61] Linux Kernel 2.6.27.5, http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.27.5.tar.bz2.

[62] German Llort, Juan Gonzalez, Harald Servat, Judit Gimenez, and Jesus Lebarta, “On-line detection of large-scale parallel applications’s structure”, *International Parallel & Distributed Processing Symposium (IPDPS 2010)*, April 2010.

[63] Robert Love, **Linux Kernel Development (2nd ed.)**, Novell Press, Indianapolis, Indiana, 2005, ISBN 0-672-32720-1.

[64] S.J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E.D. Milne, and R. Wheeler, “sfs: A parallel file system for the CM-5”, *Summer 1993 USENIX Technical Conference*, pp.291-305, June 1993.

[65] Lucent Technologies, “Introduction to the Plan 9 File Protocol”, 2010, http://plan9.bell-labs.com/magic/man2html/5/0intro.

[66] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, “TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks,” *Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[67] Matthew L. Massie, Brent N. Chun, and David E. Culler, “The ganglia distributed monitoring system: design, implementation, and experience”, *Parallel Computing* **30**, Elsevier B.V., 2004, pp. 817-840.

[68] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam and Tia Newhall, “The Paradyn Parallel Performance Measurement Tool”, *IEEE Computer* **28**, 11, November 1995, pp. 37-46. Special issue on performance evaluation tools for parallel and distributed computer systems.

[69] Ron Minnich, “V9FS: A Private Name Space system for Unix and its uses for Distributed and Cluster Computing”, *1st Conference Francaise sur les Systemes d’Exploitation (CFSE ’1)*, June 1999.

[70] Ron Minnich and Andrey Mirtchovski, “XCPU: a new, 9p-based, process management system for clusters and grids”, *2006 IEEE International Conference on Cluster Computing*, September 2006.

[71] Ryan Mooney, Kenneth P. Schmidt, R. Scott Studham, and Jarek Nieplocha, “NWPerf: A System Wide Performance Monitoring Tool for Large Linux Clusters”, *2004 IEEE International Conference on Cluster Computing (CLUSTER 2004)*, pp. 379-389, September 2004.

[72] Anna Morajko, Oleg Morajko, Tomàs Margalef, and Emilio Luque, “MATE: Dynamic Performance Tuning Environment”, *10th International Euro-Par Conference (Euro-Par 2004)*, August 2004. Appears as Lecture Notes in Computer Science **3149**, Marco Danelutto et al (Eds.), Springer, Berlin/Heidelberg, Germany, January 2004, pp. 98-107.

[73] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, R. Badrinath, and Louis Rilling, “Kerrighed: A Single System Image Cluster Operating System for High Performance Computing”, *9th International Euro-Par Conference (Euro-Par 2003)*, August 2003. Appears as Lecture Notes in Computer Science **2790**, Harald Kosch et al (Eds.), Springer, Berlin/Heidelberg, Germany, January 2003, pp. 1291-1294.

[74] Philip J. Mucci, “DynaProf Users Guide Release 0.8”, http://www.cs.utk.edu/~mucci/dynaprof/dynaprof.html, November 2002.

[75] “MultiTail”, http://www.vanheusden.com/multitail/.

[76] Aroon Nataraj, Allen D. Malony, Alan Morris, Dorian C. Arnold, and Barton P.Miller, “A Framework for Scalable, Parallel Performance Monitoring using TAU and MRNet”, *International Workshop on Scalable Tools for High-End Computing (STHEC 2008)*, June 2008.

[77] National Center for Computational Sciences, “National Center for Computational Sciences >> Jaguar”, http://www.nccs.gov/computing-resources/jaguar/.

[78] B. Clifford Neuman, “The Prospero File System: A Global File System Based on the Virtual System Model”, *Computing Systems* **5**, 1992, pp. 407-432.

[79] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing", *SIGMOD ’08*, 2008.

[80] Hong Ong, Rajagopal Subramaniyan, Chokcahi Leangsuksun, and R. Scott Studham, “Open-WLC: A Scalable Workload Characterization System”, *High Availability and Performance Computing Workship (HAPCW)*, October 2005.

[81] Hong Ong, Jeffrey Vetter, R. Scott Studham, Collin McCurdy, Bruce Walker, and Alan Cox, “Kernel-level single system image for petascale computing”, *ACM SIGOPS Operating Systems Review* **40**, 2, April 2006, pp. 50-54.

[82] “Open | SpeedShop”, http://oss.sgi.com/projects/openspeedshop/.

[83] “OpenPBS Technical Overview”, http://www.openpbs.org/overview.html.

[84] “OpenSSI (Single System Image) Clusters for Linux”, http://www.openssi.org/, August 2006.

[85] Paradyn Parallel Performance Tools, “ProcControlAPI Developer’s Guide, Beta 1“, March 2011, ftp://ftp.cs.wisc.edu/paradyn/releases/release7.0/doc/ProcControlAPI.pdf.

[86] Paradyn Parallel Performance Tools, “StackwalkerAPI Programmer’s Guide, Release 2.0“, March 2011, ftp://ftp.cs.wisc.edu/paradyn/releases/release7.0/doc/stackwalker.pdf.

[87] “Parallel Tools Consortium”, http://web.engr.oregonstate.edu/~pancake/ptools/flyer.html, March 1996.

[88] D. Pase, “Dynamic Probe Class Library (DPCL): Tutorial and Reference Guide, Version 0.1”, *IBM Corporation Technical Report*, Poughkeepsie, NY, June 1998.

[89] “PBS Professional 7.1”, Altair Engineering, http://www.altair.com/software/pbspro.htm.

[90] “PDSH - Parallel Distributed SHell”, Lawrence Livermore National Laboratory UCRL-CODE-2000-009, http://www.llnl.gov/linux/pdsh/pdsh.html, February 2003.

[91] Jan-Simon Pendry and Marshall Kirk McKusick, “Union mounts in 4.4BSD-lite”, *USENIX Technical Conference*, 1995.

[92] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall", *Scientific Programming* **13**, 4, October 2005, pp. 277-298.

[93] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, “Plan 9 from Bell Labs”, *UK UUG Summer 1990 Conference*, pp. 1–9, July 1990.

[94] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, “The Use of Name Spaces in Plan 9”, *5th ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring*, September 1992.

[95] “Platform LSF HPC”, Platform Computing, http://www.platform.com/Products/Platform.LSF.Family/Platform.LSF.HPC.

[96] Chet Ramey, “The GNU Readline Library”, http://www.gnu.org/s/readline.

[97] Herman C. Rao and Larry L. Peterson, “Accessing Files in an Internet: The Jade File System”, *IEEE Trans. on Software Engineering* **19**, 6, 1993, pp. 613-624.

[98] Bernhard Ries, R. Anderson, W. Auld, D. Breazeal, K. Callaghan, E. Richards, and W. Smith, “The Paragon Performance Monitoring Environment”, *1993 ACM/IEEE Conference on Supercomputing*, pp. 850-859, November 1993.

[99] Dennis M. Ritchie and Ken Thompson, “The UNIX time-sharing system”, *Communications of the ACM* **26**, 1, 1983, pp. 84-89.

[100] Rogue Wave Software, “The TotalView Family Brochure: Comprehensive tools for verifying, debugging, and optimizing complex applications”, May 2011. http://www.roguewave.com/products/totalview-family/totalview/resources/brochures-and-datasheets.aspx.

[101] Phillip C. Roth, Dorian C. Arnold, and Barton P. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools", *SC2003*, November 2003.

[102] Philip C. Roth and Barton P. Miller, “On-line Automated Performance Diagnosis on Thousands of Processes”, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ‘06)*, March 2006.

[103] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility", *SIGOPS Oper. Sys. Rev.* **35**, 5, December 2001, pp. 188-201.

[104] “RRDTool”, http://en.wikipedia.org/wiki/RRDtool.

[105] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon, “Design and Implementation of the Sun Network Filesystem”, *Summer 1985 USENIX Technical Conference*, pp. 119-130, June 1985.

[106] Frank Schmuck and Roger Haskin, “GPFS: A Shared-Disk File System for Large Computing Clusters”, *1st USENIX Conference on File and Storage Technologies*, January 2002.

[107] Martin Schulz, Dong Ahn, Andrew Bernat, Bronis R. de Supinski, Steven Y. Ko, Gregory Lee, and Barry Roundtree, “Scalable dynamic binary instrumentation for Blue Gene/L”, *ACM SIGARCH Computer Architecture News* **33**, 5, December 2005, pp. 9-14.

[108] Martin Schulz, John May, and John Gyllenhaal, “DynTG: A Tool for Interactive, Dynamic Instrumentation”, *5th International Conference on Computational Science (ICCS 2005)*, May 2005, pp. 140. Appears as *Lecture Notes in Computer Science* **3515**, Vaidy S. Sunderam et al (Eds.), Springer, Berlin/Heidelberg, Germany, 2005.

[109] Philip Schwan, “Lustre: Building a File System for 1,000 Node Clusters”, *2003 Linux Symposium*, July 2003.

[110] “Simple Linux Utility for Resource Management”, Lawrence Livermore National Laboratory UCRL-WEB-217616, http://www.llnl.gov/linux/slurm/overview.html.

[111] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title, “A Scalable Debugger for Massively Parallel Message-Passing Programs”, *IEEE Parallel and Distributed Technology* **2**, 2, Summer 1994.

[112] Matthew J. Sottile and Ronald G. Minnich, “Supermon: a high-speed cluster monitoring system”, *IEEE International Conference on Cluster Computing (CLUSTER 2002)*, September 2002.

[113] D. C. Steere, "Exploiting the Non-Determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency", *SIGOPS Oper. Sys. Rev.* **31**, 5, December 1997, pp. 252-263.

[114] Michael Stonebraker, “The Case for Shared Nothing”, *Database Engineering* **9**, 1, 1986.

[115] Miklos Szeredi, 'FUSE: Filesystem in user space”, http://fuse.sourceforge.net.

[116] Douglas Thain, Christopher Moretti, and Jeffrey Hemmes, “Chirp: a practical global filesystem for cluster and Grid computing”, *Journal of Grid Computing* **7**, 1, March 2009, pp. 51-72.

[117] “TORQUE Resource Manager 2.0”, Cluster Resources, http://www.clusterresources.com/pages/products/torque-resource-manager.php.

[118] A. Tridgell and P. Mackerras, “The rsync algorithm”, Australian National University Technical Report TR-CS-96-05, June 1996.

[119] Junichi Uekawa, “DSH - dancer’s shell / distributed shell”, http://www.netfort.gr.jp/~dancer/software/dsh.html.en, December 2008.

[120] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel, “The LOCUS distributed operating system”. *SIGOPS Oper. Syst. Rev.* **17**, 5, December 1983, pp. 49-70.

[121] John Walker, “logtail: Watch Multiple Log Files on Multiple Machines”, http://www.fourmilab.ch/webtools/logtail/, November 1997.

[122] Greg Watson and Craig E. Rasmussen, “A Strategy for Addressing the Needs of Advanced Scientific Computing Using Eclipse as a Parallel Tools Platform”, Los Alamos National Laboratory Technical Report LA-UR-05-9114, December 2005.

[123] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad Nayyer Zubair, “Versatility and Unix Semantics in Namespace Unification”, *ACM Trans. on Storage* **2**, 1, February 2006, pp. 74-105.

[124] Erez Zadok and Jason Nieh, “FIST: A Language for Stackable File Systems”, *USENIX Annual Technical Conference*, June 2000.

[125] Victor C. Zandy, "Application Mobility", Ph.D. dissertation, University of Wisconsin-Madison, 2004.

[126] Vic Zandy and Dan Ridge, “First-class C Contexts in Cinquecento”, IDA CCS Technical Report, April 2008. http://cqctworld.org/docs/cqct.pdf.