UNIVERSITY OF CALIFORNIA, SAN DIEGO

DISTRIBUTED APPLICATION MANAGEMENT

A Dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in
Computer Science

by

Jeannie Raye Albrecht

Committee in charge:

Professor Amin Vahdat, Chair
Professor Alex C. Snoeren, Co-Chair
Professor Rene Cruz
Professor Jeanne Ferrante
Professor Bill Lin

2007

The Dissertation of Jeannie Raye Albrecht is approved, and it is acceptable in quality and form for publication on micro-film:

_____

_____

_____

_____
Co-Chair

_____
Chair

University of California, San Diego

2007

EPIGRAPH

“Lots of folks confuse bad management with destiny.”

–Kin Hubbard

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First I would like to thank my advisors, Dr. Amin Vahdat and Dr. Alex C. Snoeren, for their support and guidance. I would also like to acknowledge my committee and other faculty members who have been influential over these past six years, including Dr. Jeff Chase, Dr. Rene Cruz, Dr. Jeanne Ferrante, Dr. Bill Lin, Dr. Stefan Savage, and Dr. Geoff Voelker. Additionally I would like to thank Dr. Rodney Tosten for his advice and endless support, and for convincing me to pursue a Ph.D. in the first place. In addition to faculty, I also want to thank all of my research colleagues who I have worked with during my graduate career, including Ryan Braud, Darren Dao, Michael Frederick, John Jersin, Dr. Dejan Kostić, Dr. David Oppenheimer, Dr. Adolfo Rodriguez, and Nikolay Topilski. I want to especially thank Christopher Tuttle whose early work on the development of Plush provided a foundation for much of the work in this thesis.

I never would have made it through six years of graduate school without the support of my friends at Duke and UCSD, including Alvin AuYoung, Allister Bernard, Dr. Andrew Danner, Laura Grit, Chris Kanich, Chip Killian, Priya Mahadevan, John McCullough, Dr. Patrick Reynolds, Sara Sprenkle, Kashi Vishwanath, Dr. Ken Yocum, and the rest of the crew in CSE 3140. I also want to acknowledge Barath Raghavan, my good friend and surf buddy, who taught me to "always swim against the current." In addition, I wish to thank the various members of the Duke Chakra and UCSD Psycho ultimate frisbee teams, who have provided countless hours of stress relief and entertainment. I especially want to acknowledge the "clutch" Psychos, including Shelley DuBois, Holley Haynes, Kelly Jarvis, Goldy Thach, and Laura Wishingrad.

Last, but certainly not least, I acknowledge and thank my family, especially Brian, Albie, and my mom, for their unconditional support and love throughout all of my endeavors. I also want to thank my favorite canine companion, Lucy, for always keeping me company and making me smile. And finally, I thank my fiancé, David Irwin, whose love, support, encouragement, and motivation have helped me overcome all obstacles.

Chapters 2 and 7, in part, are a reprint of the material as it appears in the ACM Operating Systems Review, January 2006, Albrecht, Jeannie; Tuttle, Christopher;

Snoeren, Alex C.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 2, 5, and 6, in part, are a reprint of the material as it appears in the USENIX Annual Technical Conference, 2006, Albrecht, Jeannie; Tuttle, Christopher; Snoeren, Alex C.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapters 2, 3, 4, and 6, in part, have been submitted for publication of the material as it appears in the Large Installation System Administration Conference, 2007, Albrecht, Jeannie; Braud, Ryan; Dao, Darren; Topilski, Nikolay; Tuttle, Christopher; Snoeren, Alex C.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it will appear in the ACM Transactions on Internet Technology, May 2008, Albrecht, Jeannie; Oppenheimer, David; Patterson, David; Vahdat, Amin. The dissertation author was one of the primary investigators and authors of this paper.

VITA

2007	Doctor of Philosophy in Computer Science University of California, San Diego San Diego, CA
2003	Master of Science in Computer Science Duke University Durham, NC
2001	Bachelor of Science in Math and Computer Science Gettysburg College Gettysburg, PA

PUBLICATIONS

J. Albrecht, D. Oppenheimer, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. *ACM Transactions on Internet Technology (TOIT)*, 8(2), 2008.

J. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, and A. Vahdat. Remote Control: Distributed Application Configuration, Management, and Visualization with Plush. In *Proceedings of the Large Installation System Administration Conference (LISA)*, 2007. In Submission.

J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Loose Synchronization for Large-Scale Networked Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2006.

J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. PlanetLab Application Management Using Plush. *ACM Operating Systems Review (OSR)*, 40(1), 2006.

D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. In *Proceedings of the IEEE Symposium on High Performance Distributed Compuuting (HPDC)*, 2005.

FIELDS OF STUDY

Major Field: Computer Science

Studies in Systems and Networking
Professors Amin Vahdat and Alex C. Snoeren

ABSTRACT OF THE DISSERTATION

DISTRIBUTED APPLICATION MANAGEMENT

by

Jeannie Raye Albrecht
Doctor of Philosophy in Computer Science
University of California, San Diego, 2007
Professor Amin Vahdat, Chair
Professor Alex C. Snoeren, Co-Chair

Support for distributed application management in large-scale networked environments remains in its early stages. Although a number of solutions exist for subtasks of application deployment, monitoring, and maintenance in distributed environments, few tools provide a unified framework for application management. Many of the existing tools address the management needs of a single type of application or service that runs in a specific environment, and these tools are not adaptable enough to be used for other applications or platforms. To this end, we present the design and implementation of Plush, a fully configurable application management infrastructure designed to meet the general requirements of several different classes of distributed applications. Plush allows developers to specifically define the flow of control needed by their computations using application building blocks. Through an extensible resource management interface, Plush supports execution in a variety of environments, including both live deployment platforms and emulated clusters. Plush also uses relaxed synchronization primitives for improving fault tolerance and liveness in failure-prone environments. To gain an understanding of how Plush manages different classes of distributed applications, we take a closer look at specific applications and evaluate how Plush provides support for each.

Chapter 1

Introduction

No one can deny the success of today's Internet. What started in the late 1960's as a small government-sponsored research project designed to link together four computers across the United States now connects over one billion people worldwide [112]. Further, the applications that run on the Internet have become an integral part of our lives. The pervasiveness of Internet applications has led to advancements in many aspects of our society—including medicine, education, finance, and entertainment—and these advancements have changed the way we work, learn, play, and share information. We now manage our investments, make travel arrangements, send email, read breaking news stories, learn about new medical treatments, hold conference calls, take online college courses, purchase movies and music, and search the Web directly from applications running on computers connected to the Internet in our homes and offices.

Most applications deployed on the Internet today, including those previously mentioned, run simultaneously on thousands or even millions of computers spread around the world. In general, the goal of these *distributed applications* is to connect users to shared *resources*, where we define a resource as any computing device attached to the Internet capable of hosting an application. By combining the computing power and capabilities of distributed resources, applications are able to satisfy the ever-increasing demand of their users. Some of the most popular distributed applications, such as Web search engines like Google [44], use over 450,000 computers to host their

service and meet user demand [68]. Additionally, other distributed applications, such as content distribution networks like CoDeen [79] and Coral [37], rely on the geographic diversity of hundreds of computers acting as caches to provide users with lower latency retrieval times for commonly accessed Web content. As the Internet continues to become more pervasive and spreads into more remote parts of our planet, the user demand for these services will increase. To satisfy this demand, the number and geographic diversity of the computers needed by these services will also continue to grow.

While distributed applications offer many benefits with respect to increased computing power and geographic diversity, they also introduce new challenges associated with managing computations and services running on hundreds or thousands of computers. For example, consider the task of deploying a distributed application, which involves installing the required software and starting the computation. When running a computation or service on a single resource, it is easy to download any needed software and verify that the correct version is installed. However, when running a distributed application, ensuring that hundreds of computers around the world are all running the correct version of the required software is a cumbersome and tedious task. This task is further complicated by the heterogeneity—in terms of both hardware and software—of the computers hosting the application. The second subtask in deploying an application after installing any needed software is starting the computation. For applications running on a single resource, starting an execution is trivial, and usually is a matter of running a single command. In distributed applications, however, starting a computation requires synchronizing the beginning of the execution across a distributed set of resources, which is especially difficult in wide-area settings due to the unpredictable changes in network connectivity among the resources hosting the application.

In addition to application deployment, there are many other challenges involved with keeping an application running in distributed environments, such as failure detection and recovery. In applications that run on a single resource, monitoring an execution and reacting to failures typically consists of watching a single process (or small set of processes) and addressing any problems that arise. In distributed applica-

tions, monitoring an execution consists of watching hundreds or thousands of processes running on resources around the world. If an error or failure is detected among these thousands of processes, recovering from the problem may require stopping all processes and restarting them again. The challenges associated with these tasks can be frustrating to developers, who end up spending the majority of their time managing executions and trying to detect and react to failures, rather than developing new optimizations and enhancements for increased application performance. In the remainder of this thesis, we take a closer look at the challenges associated with managing large-scale distributed applications, and discuss ways to address the problems that arise in distributed computing environments.

1.1 Limitations of the Internet

Distributed applications leverage the combined computing power and availability of multiple distributed resources to provide better scalability than applications that run on a single resource. They also can provide better fault tolerance since the probability of multiple computers failing simultaneously is typically less than the probability of a single computer experiencing a failure. However, merely distributing an application across several resources does not solve all problems. Internet design decisions made nearly 40 years ago are beginning to limit the potential of many distributed applications. For example, many Internet applications perform poorly when confronted with unexpected "flash crowds." The Internet was not designed to support sudden bursts of activity, and the ability to handle crowds, particularly in crisis situations, is essential in many applications. Currently, the Internet only achieves two to three nines of availability [51], which translates to more than eight hours of downtime per year. The telephone system, on the other hand, provides five nines of availability, implying that the telephone system experiences less than six minutes of downtime per year [9]. Thus, distributed applications that run on the Internet must provide additional application-level support for increased availability. Ideally, a distributed application should provide at

least as much availability as the telephone system if we are ever to rely on it for time critical operations.

In addition to availability, the Internet does not provide enough security for many applications. Malicious users regularly release new worms and viruses that have the ability to take over personal computers or render them useless [98]. Aside from worms and viruses, the development of advanced phishing schemes make it difficult for users to feel safe when performing confidential financial transactions, and Internet-based identify theft is becoming a more serious problem each day [33]. In order for Internet applications to achieve their maximum potential, users must be able to trust that the transactions completed online are safe and secure from hackers and thieves. The current design of the Internet does not provide any capabilities for stopping malicious users or supporting this level of trust in applications, which can limit the utility of applications. In order to overcome this limitation, distributed applications that store sensitive or confidential information need customized application-level support for security. Additionally, applications must take steps to prevent the spread of viruses and worms.

Aside from security, the Internet's design also makes it difficult to support the addition and integration of new technologies and devices. Adding new and unconventional resources to the Internet, such as cell phones or wireless sensors, is an error-prone and unreliable process. One of the main problems is that the Internet does not support mobility, so non-stationary devices like phones and wireless sensors require additional functionality that was not originally included in the design of the Internet. Further, tracking down problems with network misconfigurations or outages is a task we often still reserve for network engineers or IT (Information Technology) specialists. This is largely due to the fact that many of the protocols in use today were not designed to be easy to use, and thus the complexity of diagnosing problems or configuring new hardware is often intimidating to users without a background in computer networks. As a result, distributed applications must provide their own mechanisms and abstractions that simplify the tasks involved with configuring network devices. The applications should expose the needed information to help track down problems easier, so that the average

user need not worry about network-level details in order to successfully connect and manage their network-capable devices.

1.2 Developing and Evaluating Distributed Applications

The challenges and limitations discussed in the previous section make the development and evaluation of new distributed applications difficult. The applications must provide application-level support for availability, security, and extensibility to address the limitations in the Internet's design. Thus, before releasing a new distributed application to the public for general use, it must be thoroughly evaluated in realistic conditions to ensure that it achieves the desired levels of performance, availability, security, and extensibility. Researchers and developers currently exploit three different options for testing their applications before a public release: simulation, emulation, and live deployment. Each of these evaluation techniques allows developers to test different aspects of their applications, and in this section, we explore the advantages and disadvantages of all three approaches.

1.2.1 Simulation

Network simulation is often the first approach used when evaluating a new distributed application. Some of the common network simulators include ns-2 [76] and p2psim [60]. Network simulation does not actually involve the network at all, but rather a set of programs that run on a single computer and simulate the behavior of hosts and routers connected to a wide-area network. If a developer wishes to test their application in a simulator, it usually requires rewriting the application code to adhere to the APIs of the target simulation platform. Since simulation usually only involves a single resource, deploying and running an application simply involves executing a single command on the computer hosting the simulator.

One of the main benefits of network simulation is cost. Network simulation is the most inexpensive way to test a distributed application because typically only one

computer performs the evaluation. Another advantage of simulation is reproducibility. The application developer has complete control over the simulated network environment, and thus running an experiment repeatedly using the same simulated network conditions yields the exact same results each time. Network simulators also give users flexibility with respect to network topologies— application developers can create any arbitrary topology. Barring processor or memory limitations on the computer hosting the simulation, topologies scale to large sizes in network simulators as well, since there are no real network resources to act as the bottleneck. Lastly, another key benefit of simulation is that it can be used in early stages of application development. For example, while it may not be worthwhile to deploy an unfinished application on hundreds of computers worldwide to test one aspect of its design, this is feasible and easily accomplished using a simulator.

Some of the main disadvantages of using a network simulator for testing a distributed application stem from the fact that there is no real network involved in the simulation. Thus, it is difficult to accurately model cross traffic, resource contention, or failures that may occur in the wide-area. As a result, simulation environments limit the ability of the developer to test the application under realistic network conditions. Additionally, most network simulators ignore hardware and operating system properties, making it challenging to evaluate how the application performs in heterogeneous computing environments. The most significant disadvantage of simulators is that they do not run real application code. Hence, the application must be rewritten for the target simulation platform, which runs the risk of introducing problems that do not exist in the real code, or worse, masking problems that do exist.

1.2.2 Emulation

Another alternative for evaluating a distributed application is emulation. Some common and widely-used emulators include ModelNet [103] and Emulab [110]. When using emulation, computers connected to a local-area network (LAN) emulate the behavior of a (potentially) larger set of resources spread across the wide-area. This is

often accomplished using traffic shaping mechanisms in the network. Thus, when one computer sends data packets to another in an emulation environment, emulated network queues subject the individual packets to delay and loss rates that correspond to a wide-area network topology. In addition, most emulators allow multiple emulated hosts to reside on the same physical computer. Therefore, using only a handful of physical computers, developers emulate topologies with several hundred emulated resources. Unlike simulations, emulation platforms typically support running unmodified application code. Running an application in an emulated environment involves installing software and simultaneously starting processes on several hundred emulated resources. This task is typically made simpler through the use of scripts provided by the emulation platform for executing parallel tasks across all emulated resources.

In many ways, emulation provides the same benefits as simulation with the added ability of specifying more variable network conditions, and, most importantly, the ability to run real application code. Emulated topologies define network links that take latency, bandwidth, and loss rate into account. Some emulation platforms also provide mechanisms for injecting cross traffic into the network. Although not as cheap as simulation, emulation is still relatively inexpensive since a small set of computers can emulate a large-scale network topology. Like simulation, emulation results are reproducible, and some environments support replaying results at faster or slower speeds for detailed offline analysis. Emulation environments give developers total control over the conditions to which their application is exposed, allowing for a thorough performance analysis.

The main disadvantage of emulation is that it is difficult to emulate realistic wide-area network conditions. Although some emulators provide support for injecting cross traffic into the network, accurately modeling typical Internet traffic conditions is still an open area of research. However, recent advances may address this limitation in the near future. The Emulab project recently developed Flexlab [93], a emulation environment that supports integrating a wide variety of network models, including models obtained directly from real, wide-area networks. Another disadvantage of emulation

environments in general is that they are sometimes overly reliable and predictable as compared to volatile wide-area networks, where failures are common and expected. Scalability can be a limiting factor in emulation, as well, depending on the amount of resources available for hosting the emulation and the application's needs with respect to processor and network capacity. Based on our experiences, however, we believe that scalability in emulation is usually not a significant problem.

1.2.3 Live Deployment

The third method for evaluating distributed applications is live deployment. This involves running applications on resources connected to a "real" network. The resources are either physical computers or clusters of virtual machines (such as VMware[107] and Xen [10]), and the network is either a local-area or wide-area network. As in emulation, application code does not have to be modified during live deployment. Running applications in live deployment environments involves installing software on each machine separately, and simultaneously starting the execution across all resources. Application-specific scripts and environment-specific toolkits often simplify the task of running applications in live-deployment environments. Some common live deployment platforms include computational grids [35], local site clusters, distributed data centers, and PlanetLab [11, 83]. PlanetLab is a collection of over 700 (Linux) computers connected to the Internet at 345 sites in 25 countries. One of the main design goals of PlanetLab is to enable the introduction of new technologies into the Internet. PlanetLab exposes many new challenges associated with managing distributed applications running at scale across volatile wide-area networks, and thus is the focus of much of the work in this thesis.

One of the biggest advantages of live deployment environments is that the resources hosting the application are typically connected to the Internet, automatically exposing the application to realistic wide-area network conditions. Note that this is not entirely true in site clusters and clusters of virtual machines that reside on local-area networks, however, where the network conditions of the LAN may not be representative

of those seen across the wide-area. In these settings, developers may inject cross traffic using techniques and models similar to those used in emulation. For wide-area live deployment environments like PlanetLab, computational grids, and distributed data centers, application developers also benefit from the geographic diversity of the resources. Perhaps the most important advantage to live deployment is that real software runs on real machines connected to a real network.

The disadvantages of using live deployment are often discouraging for application developers. One problem is that live deployment is expensive. Especially for wide-area platforms, few researchers have the funding necessary to obtain hundreds or thousands of resources to spread around the world for testing purposes. Until recently, live deployment was typically only achieved by companies who could afford to create distributed data centers for hosting their applications, or at small scales by researchers who obtained accounts on computers from colleagues at other universities and companies. Fortunately, recent developments have made live deployment more accessible to a wider range of developers. By pooling together individual resources from many sites worldwide, shared testbeds like PlanetLab and several computational grids now allow researchers to run their applications on real machines across the wide-area. In addition to the cost, another disadvantage is that live deployment does not yield reproducible results. Since live deployment environments subject applications to real Internet traffic, researchers have little control over the conditions to which their application is exposed, making it difficult to analyze results. Future live deployment environments may provide ways to address this limitation. For example, GENI is a live deployment environment that plans to provide a mechanism for resource containment and experiment reproducibility in a global large-scale testbed [39].

1.2.4 Summary

Simulation, emulation, and live deployment are three different techniques for deploying and evaluating distributed applications. Each one offers developers a way to test different aspects of their applications during different stages of its development.

Thus rather than choosing one technique for evaluation, developers typically use a combination of the three in order to ensure that an application is suited for public release. For example, simulation is well suited for early development, perhaps even during the initial design phase. Simulation allows developers to test the behaviors of protocols in a controlled and predictable environment. After completing a thorough analysis of protocol behavior and coming up with an acceptable application design, most developers then move on to emulation. Because emulation environments run unmodified application code, these platforms are best suited for evaluation during early code development. Emulation affords developers the ability to thoroughly test their implementations in large-scale predictable settings. Upon the completion of a series of emulation experiments, the code should be in a stable and functional state, and developers should be confident that their implementation works in the desired way. The last stage of evaluation during application development is live deployment. Live deployment arguably provides the most realistic evaluation conditions, and thus it is important to thoroughly evaluate an application's performance using this technique. However, live deployment is also the least predictable technique, and therefore most developers complete several simulation and emulation experiments to gain a full understanding of the behavior of their application before running across a real network.

1.3 Distributed Application Management Overview

In the preceding section, we discussed the advantages and disadvantages of simulation, emulation, and live deployment in the context of distributed application development and evaluation. In addition to the advantages and disadvantages previously discussed, another key tradeoff between these techniques is the complexity of the tasks associated with managing applications. This complexity is largely dependent on the amount of control afforded to the developer and the number of resources in use for each evaluation technique. In general, it becomes more difficult to manage applications as the developer's control of the environment decreases and the number of resources

increases. Thus, managing applications during simulation is trivial, since developers have total control over the environment, and the execution usually only involves a single resource. At the opposite end of the spectrum, managing applications during live deployment can be very complicated, since these platforms offer developers little or no control over the environment, and also potentially contain hundreds of resources spread around the world.

Although all three evaluation techniques should be used during application development, in the remainder of this thesis we focus on the challenges associated with managing distributed applications running on multiple resources in emulation and live deployment environments. We are especially interested in wide-area live deployment environments, since they are the least predictable and most challenging with respect to application management. In this context, it is also important to realize that we are interested in the problems associated with distributed application management *in general*—that is, we are not limiting our work to a single type of distributed application or a single deployment environment. Instead, we discuss the challenges associated with managing a broad range of distributed applications running on a variety of computing platforms. In particular, we consider applications with varying execution times—ranging from computations that last a few minutes or less to services that run for years—as well as applications with varying computational demands—ranging from computationally intensive scientific applications to bandwidth intensive network applications. Chapter 2 discusses the different types of applications in more detail. In the remainder of this section we give a high-level overview of the challenges associated with distributed application management.

Managing distributed applications involves deploying, configuring, executing, and debugging software running on multiple computers simultaneously. Particularly for applications running on resources that are spread across the wide-area, distributed application management is a time-consuming and error-prone process. After the initial deployment of the software, the applications must detect and recover from the inevitable failures and problems endemic to distributed environments. To achieve required levels

of availability, applications must be carefully monitored and controlled to ensure continued operation and sustained performance. Operators in charge of deploying and managing these applications face a daunting list of challenges: discovering and acquiring appropriate resources for hosting the application, distributing the necessary software, and appropriately configuring the resources (and re-configuring them if operating conditions change). It is not surprising, then, that a number of tools have been developed to address various aspects of the process in distributed environments, but no solution yet flexibly automates the application deployment and management process across all environments.

Presently, most researchers who want to evaluate their applications in wide-area distributed environments take one of two management approaches. On PlanetLab, service operators address deployment and monitoring in an *ad hoc*, application-specific fashion using customized scripts. Grid researchers, on the other hand, leverage one or more software toolkits for application development and management. These toolkits often require tight integration with not only the environment, but the application itself. Hence, applications must be rewritten to adhere to the specific APIs in a given toolkit, making it nearly impossible to run the application in other environments. In emulation environments, the management approach largely depends on the platform. Some emulation environments provide toolkits and web interfaces for manipulating applications running on emulated resources, while others rely on researchers to write their own management scripts.

Despite the fact that applications must adhere to specific APIs and are not easily run in other environments, toolkits such as those commonly used in grid environments are highly regarded due to their ability to shield developers from the significant complexity associated with executing, configuring, and managing large-scale distributed computations. In particular, grid workflow management systems are especially popular. These systems allow researchers to provide a specification of a distributed computation—a high-level description of what resources a particular application requires, its individual phases of computation, and the dependencies between the

phases—as part of the application logic that actually implements the computation. In this manner, developers can avoid "hardwiring" particular configuration details such as the characteristics of individual hosts or network links in the application itself. To this point, however, such toolkits have seen little use in non-grid environments like PlanetLab or in publicly available emulation environments like ModelNet or Emulab. The toolkits are closely tied to the grid environments for which they were designed, and thus are not easily extended to support other computing platforms.

1.4 Hypothesis and Goals

Although the resource-specific low-level tasks associated with application management vary in complexity depending on the target environment and number of resources in use, at a high-level, the goals of application management across all deployment platforms are largely similar. Thus if we provide a way to help developers cope with the intricacies of managing different types of resources—ranging from emulated "virtual" hosts to real physical machines spread around the world—it should be easy for them to seamlessly run applications in a variety of environments using the same management interface. Fundamentally the only elements of the application's execution that change are the underlying resources. Similarly, this management interface should not be tied to a specific application; the interface must be extensible and customizable to support many different applications. To this end, we believe that a unified set of abstractions for shielding developers from the complexities and limitations of networked environments—including the Internet—can be applied to a broad range of distributed applications in a variety of execution environments. These abstractions help developers manage and evaluate distributed applications, to ensure that the applications achieve the desired levels of availability, scalability, and fault tolerance.

The primary goal of this thesis is to understand the abstractions and define the interfaces for specifying and managing distributed computations run in *any* execution environment. We are not trying to build another toolkit for managing distributed

applications. Rather, we hope to define the way users thinks about their applications, regardless of their target deployment platform. We took inspiration from classical operating systems like UNIX [95] which defined the standard abstractions for managing applications: files, processes, pipes, etc. For most users, communication with these abstractions is simplified through the use of a shell or command-line interpreter. Of course, distributed computations are both more difficult to specify, because of heterogeneous hardware and software bases, and more difficult to manage, because of failure conditions and variable host and network attributes. Further, many distributed computing platforms do not provide global file system abstractions, which complicates the way users typically manage their data.

As an evaluation of our hypothesis, we present Plush [86], a generic application management infrastructure that provides a unified set of abstractions for specifying, deploying, and monitoring different types of distributed applications in a variety of computing environments. The abstractions in Plush provide mechanisms for interacting with resources, defining computations and services, and achieving synchronization without making any strong assumptions about the application or the execution environment. Plush users describe distributed computations using an extensible application specification language. In contrast to other application management systems, however, the language allows users to customize various aspects of deployment and management based on the needs of an application and its target infrastructure without requiring any changes to the application itself. Users can, for example, specify a particular resource discovery service to use during application deployment. Plush also provides extensive failure management support to automatically detect and adapt to failures in the application and the underlying infrastructure. Users interact with Plush through a simple command-line interface or a graphical user interface (GUI). Additionally, Plush exports an XML-RPC interface for programmatically integrating applications with Plush if desired.

In order to verify that our hypothesis is correct, in this thesis we show how Plush manages applications that fall into three different classes: short-lived computations, long-lived services, and parallel grid applications. We believe that these classes

encompass the majority of distributed applications, and by showing that Plush provides support for each class, we prove that the abstractions in Plush can in fact be applied to a broad range of distributed applications. Further, we also show how Plush interacts with resources from a variety of computing environments without making any strong assumptions about the underlying infrastructure. In particular we show how Plush supports execution using virtual resources, emulated resources, and real physical resources. Lastly, since one of our goals is to develop a framework that other users can apply to a variety of applications in different environments, we discuss the usability features of the Plush user interfaces, and summarize feedback received from various Plush users at different institutions.

1.5 Contributions

In summary, Plush is an application management framework that provides extensible abstractions for managing resource discovery and acquisition, software distribution, and process execution in a variety of distributed environments. Combinations of Plush application "building blocks" specify a customized control flow for different types of distributed applications. Once an application is running, Plush monitors the execution for failures or application-level errors for the duration of its lifetime. Upon detecting a problem, Plush performs a number of user-configurable recovery actions, such as restarting the application, automatically reconfiguring it, or even searching for alternate resources. For applications requiring wide-area synchronization, Plush provides several efficient synchronization primitives. In particular, Plush implements a new set of barrier semantics designed for increased performance and robustness in failure-prone environments. Plush users interact with their applications through three different user interfaces. The remaining chapters that follow describe each of these features in detail. In particular, this thesis makes the following contributions.

- Chapter 2 distills the general requirements of any distributed application management infrastructure. In addition, we identify three different classes of distributed

applications: short-lived computations, long-lived services, and parallel grid applications. By outlining the requirements of each of these classes of applications, we determine the common elements involved with managing distributed applications. The list of requirements provides motivation for the design of Plush. We conclude Chapter 2 with a discussion of related work.

- Chapter 3 builds on the general requirements outlined in Chapter 2 and discusses the design and implementation of Plush. The architecture of Plush consists of three key components: the application specification, core functional units, and user interface. The Plush application specification defines a set of application building blocks that allows users to specify the flow of control for distributed applications. The core functional units define some of the key abstractions in the internal design of Plush, including barriers and processes. Plush provides several different user interfaces for users to interact with their applications, and we discuss each interface in detail. We also describe how these components work together to manage distributed applications. In addition, Chapter 3 includes a detailed discussion on fault tolerance and scalability, which are two of the key challenges in the design and implementation of Plush.

- Chapter 4 details the Plush resource matcher. One of our key design goals in Plush is to support execution in a variety of environments. In order to achieve this goal, Plush must provide support for many different types of resources, including PlanetLab hosts, virtual machines, and emulated machines. In addition, we discuss the abstractions that Plush uses to support execution on different types of resources by interacting with several external resource management frameworks using a common API. We also describe the process that the Plush resource matcher uses to find and maintain the best set of resources available for hosting an application.

- Chapter 5 describes the design and implementation of partial barriers in Plush. A key challenge when managing applications in potentially volatile, wide-area environments is ensuring that the applications continue to make forward progress

in the execution, even in the face of failures. Traditional barrier-based synchronization primitives are too strict to achieve good performance and ensure forward progress in wide-area environments. Partial barriers relax these traditional synchronization semantics to improve performance across the wide-area. Chapter 5 discusses the relaxations, and explains how Plush uses partial barriers during application management.

- Chapter 6 evaluates Plush's ability to manage different types of applications by studying several specific distributed applications. In particular, we evaluate example applications from multiple classes of distributed applications, as discussed in Chapter 2. The purpose of Chapter 6 is to show that Plush is effective in providing useful abstractions for developers to manage their executions, and also to show how the performance of the applications actually improves due to the fault tolerance and added reliability mechanisms that Plush provides.

- Chapter 7 describes some potential areas for future work with respect to the development of Plush, makes general conclusions about distributed application management, and discusses some lessons that we learned during the development of Plush. Chapter 7 also includes several comments about the usefulness of Plush from users at different institutions around the world.

Chapter 2

Requirements for Managing Distributed Applications

To better understand the requirements of a distributed application controller, we first consider how different types of applications are typically run on PlanetLab. We then use the needs of these applications to distill a list of general requirements that a distributed application management infrastructure must support.

2.1 Classes of Distributed Applications

We start by describing three distinct classes of distributed applications: short-lived computations, long-lived Internet services, and parallel grid applications.

2.1.1 Short-lived Computations

One common type of distributed application that runs on PlanetLab is the interactive execution of short computations. The computations range from simple to complex, but many high-level characteristics of the applications are the same. In particular, the computations only run for a few days or less, and the executions are closely monitored by the user (*i.e.*, the person running the application). To run an application, first the user finds and gains access to machines capable of hosting the application. When

running a short-lived application, most users strive to find powerful machines with good connectivity at the time when the application is started. After locating suitable machines, the user installs the required software on the selected hosts, runs the application to completion, and collects any output files produced for analysis. If an error or failure is detected during execution, the application is aborted and restarted.

Now we consider a specific scenario that further examines the process of running a short-lived computation in a distributed environment. Suppose a user wants to test a new file-distribution application on 50 PlanetLab nodes scattered around the world. In general, file-distribution applications on the Internet are not short-lived. However, suppose that the user is developing a new file-distribution application, and wants to test its performance (*i.e.*, time required for all hosts to download a specific file) across the wide-area for files of different sizes. PlanetLab is often used in this manner for testing new applications before making them publicly available. Before running the application, the user must gain access to PlanetLab resources. Authentication on PlanetLab is based on public-key cryptography, and access to PlanetLab machines is achieved through SSH login using RSA authentication. To obtain SSH login privileges, the user registers with PlanetLab Central (PLC) to obtain a user account, and creates an SSH key pair. After uploading the public key to the PLC database, the user associates their PLC account with a specific PlanetLab *slice*. A slice is a named set of distributed PlanetLab resources, forming the basis for both resource allocation and isolation on PlanetLab. The user binds the newly created slice to a number of physical PlanetLab machines (*e.g.*, using a Web interface), causing the user's public key to be copied to the nodes and authorizing the user to login to the machines [25, 85].

After gaining access to PlanetLab resources, the next step is to find a suitable set of machines scattered around the world to host the application. Resource discovery tools like SWORD [3, 77] are commonly used to help streamline the process on PlanetLab. In our example file-distribution application, suppose the user wants 50 machines with fast processors, low load, and high pairwise available bandwidth to maximize performance. After finding machines that meet these requirements, the user transfers any

required software to the 50 chosen machines. PlanetLab does not provide a global file system, so each machine separately and individually downloads and installs the software package. This is accomplished using a file transfer protocol such as scp, wget, Bullet [56], or CoBlitz [80]. After the machines have been prepared with the software, the processes are started on all 50 machines by connecting to each machine separately via SSH and then executing the appropriate commands. As the execution runs, the user periodically checks the status of each host to ensure the application runs correctly. After all hosts have finished downloading the specified file, any remaining processes that did not cleanly exit are killed, and any desired output or log files that were generated on the PlanetLab machines are copied to a central location for analysis.

2.1.2 Long-lived Internet Services

Aside from short computations, another type of distributed application that is often run on PlanetLab is a continuously running service. Unlike short-lived applications, long-running services are not closely-monitored, typically run for months or even years, and provide a service to the general public. Hence, in addition to the tasks described above for obtaining and configuring resources for hosting short-lived computations, service operators must perform additional tasks to maintain the services over an extended period of time. The environments in which services run change over time, exposing the applications to a variety of operating conditions. Further, the machines that host the services are often taken offline for software or hardware upgrades, and therefore the services must recover from these changes. Some common examples of services in use today on PlanetLab include CoDeen [79], Coral [37], and OpenDHT [92].

Since operators generally do not closely monitor long running services, detecting failures is difficult. If the operator does not periodically check for errors and users of the service fail to report outages, it is not uncommon for a problem to go unnoticed by the operator for weeks or even months. When running short-lived applications like the file-distribution example previously described, users running the application quickly notice when a failure occurs, and often treat the failure as an aberrant condition and

discard the results of the run. For long-running services, failures are the rule rather than the exception and, therefore, must be addressed as such. Thus, rather than aborting the application in response to failure, the operator attempts to detect and recover from the failure to restore the service as quickly as possible. The ultimate goal of the service maintainer is to minimize downtime and maximize availability, even when failure rates are high. Additionally, since services run continuously for long periods of time, it is likely that the service itself will need to be upgraded to remain compatible with underlying operating system features. Hence, supporting software upgrades to the service itself is another challenge that must be addressed. Ideally the upgrades will not drastically impact the service's availability, although achieving this goal is often difficult.

For example, suppose a service operator wants to deploy a new resource discovery service on PlanetLab. The application aims to run on as many PlanetLab machines as possible and be highly available to provide accurate information to users of the service. To deploy such a service on PlanetLab, an operator goes through the same process as previously described for establishing authentication, adding nodes to a PlanetLab slice, finding a suitable set of resources, transferring software, and starting the executable. However, unlike short-lived applications that often run for short periods of time on powerful machines with good connectivity, *i.e.*, low-latency and high-bandwidth connections, services run for months or years, and are subjected to changing network conditions resulting in machines with slow or lossy connections in addition to more desirable low-latency, high-bandwidth links. Further, sporadic resource contention on PlanetLab often leads to saturated machines with low amounts of free memory and available processor power. Thus, choosing nodes to host an Internet service on PlanetLab often hinges on avoiding nodes that frequently perform poorly over relatively long time periods rather than choosing nodes that perform well at any given point in time [91].

Once the service is running, the operator periodically monitors the individual processes running on each host for failures. One popular service monitoring technique involves the use of customized scripts or cron jobs that check the status of the application at specified intervals in at attempt to automatically detect and recover from the

more common failure conditions, including network outages, hardware failures, operating system or software incompatibilities, and application errors. Another solution is to monitor application-generated log files for error information. If a problem is found in a log file, it may be possible to restart a misbehaving process before a widespread outage occurs. When a software upgrade to the service is required, the operator has two main options. First, the user can perform a "hot upgrade" on each machine. During a "hot upgrade" the service is upgraded without ever stopping the individual processes. If this is not possible, the second option is to shut the service down on each host, install the new version of the software, and restart the service as quickly as possible.

2.1.3 Parallel Grid Applications

Though there are many different types of grid applications, one of the most common usage scenarios for computational grids is harnessing resources at one or more sites to execute a computationally intensive job. A typical grid application involves gathering data from specific sites, and then processing this data using a compute-intensive algorithm to produce the desired result. Unlike the file-distribution application previously described that embraced the geographic diversity of PlanetLab machines, most grid applications are compute-intensive, and view network connectivity and geographic resource distribution as "necessary evils" to accomplishing their goals [94]. Since grid applications tend to be compute-intensive, many are designed to be highly parallelizable: rather than running on a single machine with one or more processors, the computation is split up and run across several machines in parallel. Parallelization has the potential to increase the overall performance substantially, but only if each machine involved makes progress. The rate of completion for individual tasks is often delayed by a few slow machines or processors. For a researcher running a parallel application, maintaining an appropriate and functional set of machines is crucial to achieving good throughput.

Let us again consider a specific example. Suppose a physicist wants to run EMAN [66] on PlanetLab. EMAN is a publicly-available software package used for reconstructing 3-D models of particles using 2-D electron micrographs. The program

takes a 2-D micrograph image as input and then runs a "refinement" process on the image to create a 3-D model of the particle. The refinement process is run repeatedly until yielding a result with the desired quality. Each iteration of refinement consists of both computationally inexpensive sequential computations and computationally expensive parallel computations. During this process the original micrographs are distributed among several machines that simultaneously run the refinement process computations in parallel. For multiple iterations of refinement, the entire cycle is repeated.

As in the other applications, the researcher running EMAN has to gain access to PlanetLab, find suitable resources, distribute the software and data files, install the software, and start the executables. Unlike the short-lived application or the long-running service described above, however, the performance of EMAN is greatly affected by the computational resources available on the machines hosting the parallel computations of the refinement process. Thus, with each iteration of the refinement process, the researcher running the application wants to use the set of PlanetLab machines that has the most available computational resources. Further, if a machine fails or suddenly becomes overloaded during execution, the machine should be replaced by another with more available resources. Detecting and recovering from these bottlenecks is both difficult and essential to achieving high performance and throughput[1].

2.2 Application Management Requirements

In the preceding section we described the process of executing three different types of distributed applications. Though the low-level details for managing the applications are different, at a high-level the requirements for each example are largely similar. Rather than reinvent the same infrastructure for each application separately, we set out to identify commonalities across all three classes of distributed applications, and build an application control infrastructure that supports all types of applications and execution environments. Based on the example applications in the previous discussion, we now

[1] In reality, PlanetLab is typically not used for running parallel grid applications like EMAN. Dedicated computational grids such as NEESgrid [82] and Teragrid [20] are often used instead. We use PlanetLab in this discussion to provide a better comparison to the previous two example applications.

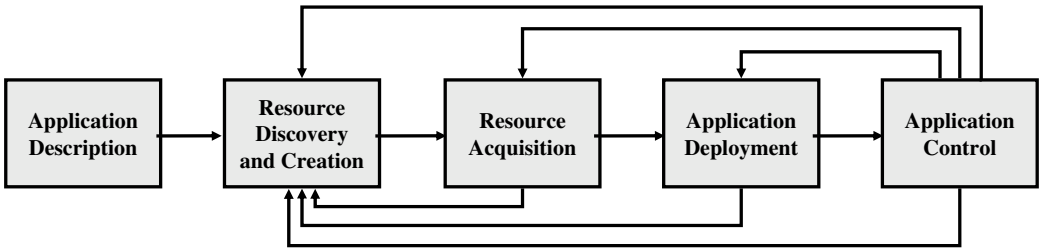

Figure 2.1: Basic requirements and control flow for a distributed application controller.

extract some general requirements for a distributed application management infrastructure. Together, these requirements identify the abstractions needed for defining the flow of control for any distributed application, as shown in Figure 2.1.

2.2.1 Application Description

A distributed application controller must allow the user to customize the flow of control for each application. This *application specification* is an abstraction that describes distributed computations. A specification identifies all aspects of the execution and environment needed to successfully deploy, manage, and maintain an application. It describes the resources required, software needed to run the application (and instructions for how to install it), processes that run on each resource, and environment-specific execution parameters. User credentials for resources are also included in the application specification. To manage complex multi-phased applications like EMAN, the specification supports defining application-specific synchronization requirements. Similarly, distributing computations among pools of resources requires a way to specify a workflow—a collection of tasks that must be completed in a given order—within an application specification. The application controller parses and interprets the application specification, and uses the information to guide the flow of control for the remainder of the application's life cycle.

The complexity of distributed applications varies greatly from simple, single-process applications to elaborate, parallel applications. Thus the challenge in building

a general application management infrastructure is to define a specification language abstraction that provides enough expressibility for complex distributed applications, but is not too complicated for single-process computations. In short, the language must be simple enough for novice users to understand, yet also expose enough advanced functionality to run complex scenarios.

2.2.2 Resource Discovery and Creation

In addition to the application description, another key abstraction in distributed application management is a *resource*. Put simply, a resource is any network accessible device capable of hosting an application on behalf of a user, including physical, virtual, and emulated machines. Because resources in distributed environments are often heterogeneous, users naturally want to obtain a resource set that best satisfies their application's requirements. In shared computing environments, even if hardware is largely homogeneous, dynamic characteristics of a host such as available bandwidth or CPU load can vary over time. The goal of resource discovery in these environments is to find the best *current* set of resources for the distributed application as specified by the user. In environments that support dynamic virtual machine instantiation, these resources may not exist in advance. Thus, resource discovery involves finding the appropriate physical machines to host the virtual machine configurations, and creating the appropriate virtual machines as needed.

Many solutions exist for resource discovery and creation in distributed environments. On PlanetLab, services like CoMon [81] and SWORD [3, 77] provide users with measurements and mechanisms for monitoring and locating hosts that meet application-specific criteria. Distributed computing environments that support virtual machines (*e.g.*, Xen [10] and VMWare [107]) leverage virtual machine configuration and management frameworks such as Shirako [49] and Usher [71] for efficient resource discovery and creation. A distributed application controller must either provide a default mechanism for performing resource discovery directly, or it should interface with these existing external tools. In the latter case, the role of the application control infras-

tructure is to parse the user's request for resources, send the request to an appropriate resource discovery and creation mechanism, and export a common API that allows these services to add and remove resources from the user's application resource pool. If the application control infrastructure provides its own mechanism for resource discovery, it must provide built-in functionality for finding the best set of available resources without contacting an outside service.

2.2.3 Resource Acquisition

Resource discovery and creation systems often interact directly with resource acquisition systems. After locating the desired resources during resource discovery and creation, resource acquisition involves obtaining a lease or permission to use the resources. Depending on the application's target execution environment, resource acquisition can take a number of forms. In best-effort computing environments (*e.g.,* PlanetLab), no advanced reservation or lease is required, so no additional steps are needed to acquire access to the resources. To support advanced resource reservations such as those used in environments where resources are arbitrated by a batch scheduler, resource acquisition involves potentially waiting for resources to become available and subsequently obtaining a "lease" from the scheduler. In virtual machine environments, resource acquisition includes verifying the successful creation of virtual machines, and gathering the appropriate information (*e.g.,* IP address, authentication keys) required for access. If any failures or problems arise while trying to acquire resources, the application controller recontacts the resource discovery and creation mechanism to find a new set of available resources if necessary.

The challenge facing an application controller is to provide a generic resource-management interface that supports execution in all computing environments. The complexities associated with creating and gaining access to physical or virtual resources should be hidden from the user. As the size and popularity of distributed computing environments continue to grow, the process of acquiring resources will become increasingly more complex. Currently, most large-scale distributed environments are centrally

managed, so that obtaining access to resources only involves gaining access to a central site. This central site acts as a trusted intermediary, and has the ability to grant access to resources at all other sites [84]. There are scalability limitations to this centralized approach, however, and as we move toward more decentralized and distributed approaches for managing large-scale networked environments, gaining access to resources may become more complicated and involved. For these reasons, a distributed application controller must be extensible enough to support a variety of methods for resource acquisition.

2.2.4 Application Deployment

Upon obtaining an appropriate set of resources, the application deployment abstraction defines the steps required to prepare the resources with the correct software and data files, and run the executables to start the application. There are really two phases that must be completed during application deployment in the general case. The first phase prepares the resources for execution. This involves downloading, unpacking, and installing any required software packages, checking for software dependencies, verifying correct versions, and basically ensuring that all resources have been correctly configured to run the desired application. Some environments may have a common/global file system, while others may require each resource to separately download and install all software packages. As a result, the application controller must support a variety of file-transfer and decompression mechanisms for each target execution environment, and should react to failures that occur during the transfer and installation of all software. Common file transfer mechanisms include scp, wget, rsync, ftp, CoBlitz [80], and Bullet [56], and common decompression tools include gunzip, bunzip2, and tar. In addition, to further simplify resource configuration, the application controller must interface with package management tools such as yum, apt, and rpm.

The second phase of application deployment begins the execution. After the resources have been prepared with the required software packages, the application controller starts the application by running the processes defined in the application spec-

ification. One key challenge in the application deployment phase is ensuring that the requested number of resources are correctly running the application. This often involves reacting to failures that occur when trying to execute various processes on the selected resources. In order to guarantee that a minimum number of resources are involved in a distributed application, the application controller may need to request new resources from the resource discovery and acquisition systems to compensate for failures that occur during software installation and process execution. Further, many applications require some form of *loose synchronization* across resources to guarantee that various phases of computation start at approximately the same time. Providing synchronization guarantees without sacrificing performance in distributed computing environments is challenging, especially in failure-prone and volatile large-scale networks.

2.2.5 Application Control

Perhaps the most difficult requirement for managing and controlling distributed applications is monitoring and maintaining an application after it starts. Thus, another abstraction that the application controller must define is support for customizable application maintenance. One important aspect of maintenance is application and resource monitoring, which involves probing resources for failure due to network outages or hardware malfunctions, and querying applications for indications of failure (often requiring hooks into application-specific code for observing the progress of an execution). Such monitoring allows for more specific error reporting and simplifies the debugging process. The challenges of application maintenance include ensuring application liveness across all resources, providing detailed error information, and achieving forward progress in the face of failures. In order to accomplish these goals, it is desirable that the application controller have a user-friendly interface where users can obtain information about their applications, and if necessary, make changes to correct problems or improve performance.

In some cases, system failures may result in a situation where application requirements can no longer be met. A robust application management infrastructure must

be able to adapt to “less-than-perfect” conditions and continue execution. For example, if an application is initially configured to be deployed on 50 machines, but only 48 are available and can be acquired at a certain point in time, the application controller should contact the user, and, if possible, adapt the application appropriately to continue executing with only 48 machines. Similarly, different applications have different policies for failure recovery. Some applications may be able to simply restart a failed process on a single resource, while others may require the entire execution across all resources to abort in case of failure. Thus, the application controller should support a variety of options for failure recovery, and allow the user to customize the recovery behaviors separately for each application. For applications that have strict resource requirements, the application controller may need to contact the resource discovery, creation, and acquisition subsystems to obtain new resources for hosting the application and recover from failures.

2.3 Related Work

The functionality required by an application controller as discussed in this chapter is related to work in a variety of areas, ranging from remote execution tools to application management systems. In this section we examine several projects in these areas, and discuss the extent to which they fulfill the requirements outlined in the preceding section. In addition, we also discuss related work that addresses workflow management, resource discovery, creation, and acquisition, and synchronization, since these are key components in distributed application management.

2.3.1 Remote Execution Tools

With respect to remote job execution, there are several tools available that provide a subset of the functionality required for distributed application management. In particular, cfengine [19] is a language-based software configuration system designed to help system administrators manage networks of machines. The goal is to allow users

to describe their configuration requirements in a high-level language, such that a single cfengine file will describe the setup of every machine connected to the network. This saves users from writing customized scripts that attempt to account for the subtle differences that exist across heterogeneous systems.

gexec [23] and pssh [24] are tools designed to perform remote job execution in cluster environments. The goal of both of these projects is to simply run a command on multiple computers simultaneously. The key difference between pssh and gexec is that gexec is designed to run mainly in cluster environments, whereas pssh is designed to run on any set of hosts, including those potentially spread around the world. Rather than requiring a user to maintain separate connections to each machine, gexec and pssh transparently manage the underlying connections and execute commands on behalf of the user. The systems are designed to be robust and scale to over 1000 nodes. vxargs [67] is a similar tool that extends the functionality provided by pssh by providing a wider range of supported commands, and also by providing a user interface for monitoring the execution status of processes.

GridShell [108] and GCEShell [75] are distributed shells designed specifically for grid-computing environments. GridShell incorporates extensions to the Tenex C Shell (TCSH) and Bourne Again Shell (BASH) that transparently support distributed grid-computing operations. GCEShell provides a new set of shell commands that resemble common UNIX operations and are actually implementations for remote Web Services/Open Grid Service Architecture (OGSA) services that are executed on behalf of the user. Both GridShell and GCEShell support script processing, and aim to provide a user-friendly interface for managing remote resources.

The difference between these tools and the requirements of a distributed application controller is that an application controller requires more functionality than just remote job execution. An application controller must be robust and provide mechanisms for failure detection and recovery, as well as automatic reconfiguration due to changing conditions. Additionally, most of the remote execution tools described above require scripts and configuration files that specifically define the set of machines on which to

run commands. A distributed application management infrastructure should support a variety of different types of resources, and not be tied to a specific set of machines.

2.3.2 Application Management Systems

In addition to remote job execution tools, application management projects such as the PlanetLab Application Manager (Appmanager) [48] focus specifically on managing and configuring distributed applications. Appmanager is a tool designed for use on PlanetLab that helps users maintain distributed applications and services. It uses a simple client-server architecture, where the central server maintains a database that stores all pertinent information about the application being run, and the clients are solely responsible for periodically updating the server with the application's status. Appmanager relies on cron jobs running on the clients for maintaining an application and for retrieving status updates. There is no user interface for interacting with the clients running on the remote PlanetLab hosts directly. In order to obtain information about an individual host, the user views a webpage that is periodically updated by the server and displays information about each host involved in the application.

HP's SmartFrog [43] project is a framework for describing, deploying, and controlling distributed applications. It is written entirely in Java, and consists of a collection of distributed daemons that manage applications and a high-level language that describes applications. SmartFrog is a not a turnkey solution to application management, but rather a framework or API for building configurable systems. In order to manage a distributed application using SmartFrog, the user must write the application such that it adheres to the SmartFrog API, or at the very least, build a SmartFrog "wrapper" program around their application. SmartFrog also provides a set of scripts for starting, stopping, and manipulating distributed applications running remotely, and provides mechanisms for detecting and recovering from failures.

There are several other commercially available products that perform functions related to application management and configuration. Namely, Opsware [78] and Appistry [8] provide software solutions for distributed application management. Op-

sware System 6 allows customers to visualize many aspects of their distributed systems, and automates software management of complex, multi-tiered applications. Appistry Enterprise Application Fabric strives to deliver application scalability, dependability, and manageability in grid-computing environments. Both of these tools focus more on enterprise application versioning and package management aspects of application configuration, and less on providing support for debugging, configuring, and interacting with experimental distributed systems.

In grid-computing environments, several different projects address various aspects of application management, including Condor [18] and GrADS [12]. Condor [18] is a workload management system for compute-intensive jobs that is designed to deploy and manage distributed executions in computational grids. Condor is optimized for leveraging underutilized cycles in desktop machines within an organization where each job is parallelizable and compute-bound. GrADS [12] provides a set of programming tools and an execution environment for easing program development in grids. GrADS focuses specifically on applications with variable resource requirements during execution and environments with dynamically-changing resources. vGrADS is a project that is largely based on GrADS. vGrADS aims to accomplish the same goals as GrADS, but provides several extensions related to Virtual Grid [53] environments. Both vGrADS and GrADS were designed to manage compute intensive scientific grid applications that are highly dependent on using the best resources available to obtain results. Thus, once an application starts execution, GrADS/vGrADS maintains resource requirements through a stop/migrate/restart cycle. All of these systems are similar in that they focus on maximizing the application's performance through job migration and rescheduling in an attempt to use the best resources available.

Lastly, the Globus Toolkit [34] is a framework for building grid systems and applications, and is perhaps the most widely used software package for grid development. Although Globus does not directly manage applications, it does provide several components that perform tasks related to application management. With respect to an application specification, the Globus Resource Specification Language (RSL) provides

an abstract language for describing resources, though it does not provide a mechanism for describing entire applications. In the context of resource management, the Globus Resource Allocation Manager (GRAM) processes requests for resources, allocates resources, and manages active jobs in grid environments. Like SmartFrog, Globus is a framework that provides many application configuration options, but each application must be built specifically using Globus APIs to achieve the desired functionality.

Although all of these tools meet many of the requirements described in Section 2.2, none of them meet *all* of the criteria. In particular, these systems provide a flexible and robust way to manage applications in distributed environments, but most are lacking the functionality required to interact with applications from a user-friendly interface. Further, many of these tools are tied to specific computing environments (*e.g.*, Appmanager only works on PlanetLab, GrADS only works in grids), therefore limiting the developer to a specific computing platform for all experimentation and deployment. A more desirable application controller combines the features of these application management systems with the flexibility to execute in different environments and the interactivity of the remote job execution tools previously described.

2.3.3 Workflow Management

Workflows are a crucial part of many distributed applications, and thus it is important for an application controller to provide support for workflow management. In this section we investigate some of the projects that address the challenges associated with managing workflows in grid environments, starting with GridFlow [26]. GridFlow focuses on service-level scheduling problems. GridFlow users submit grid jobs to a batch scheduler, and GridFlow uses a fuzzy timing prediction technique to estimate a quick solution to resource conflict problems that often arise in shared computing environments. Since the fuzzy time functions in GridFlow are computed quickly, this technique is well suited to time-critical grid applications.

Kepler [65] is a framework for designing and modeling scientific workflows. It builds on the functionality provided by Ptolemy II [88], which provides a set of Java

packages for heterogeneous, concurrent modeling of workflows. Workflows in Ptolemy II are described using a XML-based language called MoML. The goal of Kepler is to help scientists who do not have programming experience execute scientific workflows in grid-computing environments. Hence, Kepler provides an intuitive graphical interface for building workflows rather than requiring an understanding of XML. Although Kepler provides basic functionality for executing workflows, the project focuses on designing workflows, rather than dealing with failures and scheduling problems during execution.

Karajan [106] is part of the Java CoG Kit [105] and is an extensible workflow management framework. It implements abstractions for easy integration with a variety of grid middleware systems including Globus [34] and Condor [18]. Karajan provides extensions that emphasize scalability, workflow structure, and workflow error handling. Workflows are described using a XML-based language that supports both conditional control flows and loops. Karajan supports a simple, non-interactive command-line interface and a graphical user interface that executes basic interactive commands. Like Kepler, Karajan focuses more on defining complex scientific workflows rather than execution, although the framework does provide advanced support for application-specific error handling.

A variety of other tools are described in [116] that have similar goals as GridFlow, Kepler, and Karajan. In general, these tools specialize in managing workflows, specifically in grid environments. They aim to simplify the task of describing a scientific application that uses a workflow for execution, so that scientists without a programming background can make use of grid resources for compute-intensive calculations. Further, most the the systems are designed to work in an environment where access to resources is arbitrated by a batch scheduler. A generic distributed application controller should provide similar functionality for easily describing workflows and supporting batch scheduling environments as these systems. However, since we want to support a variety of distributed applications, additional functionality is required. For example, it would not be possible for an application developer who wants to manage a long running service to use Kepler or Karajan to deploy, monitor, and maintain their

application. When designing a general purpose distributed application controller, additional extensions are needed to manage applications without workflows that are not run in batch scheduled environments.

2.3.4 Resource Discovery, Creation, and Acquisition

Since resource discovery, creation, and acquisition are important aspects of distributed application management, it is useful to examine key work in these areas. Note that an application controller need not implement these functions directly, but should provide an interface that allows applications to interact with the existing services. Instead of completing an exhaustive survey of all work related to resource discovery, acquisition, and creation, we highlight some of the main projects for several different computing platforms, including grid-computing environments, PlanetLab, and virtual machine environments.

With respect to resource discovery, there are several tools designed for grid environments that allow users to find appropriate resources for hosting their applications. Many of these tools are part of larger application management systems that were previously described. In the Globus Toolkit [34], for example, resource discovery is accomplished using the Monitoring and Discovery Service [117]. MDS2 is a framework that uses a combination of Information Providers (IP) for measuring resource usage, Grid Resource Information Services (GRIS) for publishing the measurements, and Grid Index Information Services (GIIS) for aggregating data provided by the GRISes. MDS3 and MDS4 [41] are successors to MDS2 that have similar goals. In general, MDS is a service that provides mechanisms for finding available resources that meet an application's resource specifications. The vGrADS project also has a tool for performing resource discovery and acquisition called vgFAB [53]. vgFAB maintains a database of resource measurements and pre-computes sets of resources that meet certain criteria. Applications managed by vGrADS use vgFAB to "find and bind" to specific virtual grid resources. In the Condor [18] execution management system, applications and resource providers use a resource specification language called ClassAds [62] to ad-

vertise and describe resources. Condor's matchmaker matches resource advertisements with requests using mechanisms called GangMatching [90] and SetMatching [63]. Resource acquisition in many grid environments involves waiting for a batch scheduler to grant permission for an application to use some set of resources. Some of the most commonly used batch schedulers include Sun Grid Engine (SGE) [40], Portable Batch System (PBS) [87], Maui [70], and Load Sharing Facility (LSF) [64].

On PlanetLab, there has also been a number of efforts that address various aspects of resource discovery. Since PlanetLab is a best-effort environment, no additional steps are required to acquire resources. However due to the high amounts of resource contention, services such as SWORD [3, 77] provide users with a way to find resources that best meet the needs of their application given the current operating conditions. SWORD defines an XML-based resource description language where users define queries that describe groups of resources with specific per-node (*e.g.*, load) and inter-node (*e.g.*, latency) properties. In response to user queries, SWORD returns a list of hostnames organized into groups, and ranked according to how well they meet the requested criteria. Similarly, CoMon [81] is a PlanetLab service that measures resource usage across all PlanetLab nodes. The statistics measured by CoMon are obtained using a sensor interface on individual PlanetLab hosts, or viewing a central webpage containing the aggregate of all the collected data for all hosts. The web interface also supports simple queries for hosts that satisfy basic requirements.

The increasing popularity of virtual machine technologies has led to the development of several projects that explore using virtual machines to host network applications. Shirako [49] is a toolkit for building utility service architectures that is based on an extensible resource leasing abstraction. It contains an implementation of Cluster-On-Demand (COD) [22], which is a physical and virtual machine manager that supports the dynamic creation of multiple independent "virtual clusters" in a single physical cluster. Specifically, users submit requests for virtual clusters of machines with desired attributes, and COD instantiates a virtual cluster of Xen machines that contain these attributes. Access to the resources is arbitrated by leases issued by Shirako. Usher [71]

is a related project that focuses on virtual machine scheduling and placement in cluster environments. After a user submits a request for resources, Usher creates virtual machines and clusters, and then uses information gathered by virtual machine monitors running on physical machines to make decisions regarding the placement of the virtual machines. Usher hides the complexity of finding physical machines capable of hosting virtual machines from users. Other similar projects related to virtual machine creation and management include Sandpiper [111], In-VIGO [1], VMPlants [57], The Collective [21], Virtual Workspaces [52], and Virtuoso [97].

2.3.5 Synchronization

Synchronization has been studied for many years in the context of parallel computing, and is an important aspect of distributed application management. For traditional parallel programming on tightly coupled multiprocessors, *barriers* are commonly used to separate phases of computation within an execution, and form natural synchronization points [50]. Given the importance of fast primitives for coordinating bulk synchronous SIMD applications, most massively parallel processors (MPPs) have hardware support for barriers [59, 96]. Barriers also form a natural consistency point for software distributed shared memory systems, often signifying the point where data will be synchronized with remote hosts [54, 13]. In addition to shared memory, another popular programming model for loosely synchronized parallel machines is message passing. Popular message passing libraries such as PVM [38] and MPI [72] contain implementations of barriers as a fundamental synchronization service.

One drawback to traditional barriers is that the throughput of the entire execution is limited by the throughput of the slowest processor. Gupta's work on SIMD programming for tightly coupled parallel processors addresses this limitation using fuzzy barriers [46]. Gupta's approach specifies an entry point for a barrier, followed by a subsequent set of instructions that can be executed before the barrier is released. Thus, a processor is free to be anywhere within a given region of the overall instruction stream before being forced to block. In this way, processors that complete a phase of compu-

tation early can proceed to other computations that do not require strict synchronization before finally blocking. Fuzzy barriers are especially helpful in SIMD programs where there can only be a single outstanding barrier at any time.

Another use of barriers in distributed application management is to perform scheduling and load balancing based on barrier arrival rates. A similar technique is used in Implicit Coscheduling, where the arrival rate at a barrier (and the associated communication) is one consideration in making local scheduling decisions to approximate globally synchronized behavior in a multi-programmed parallel computing environment [31]. Further, performing load balancing by reallocating work upon arriving at a barrier is similar to methods used by work stealing schedulers like CILK [17]. The fundamental difference here is that idle processors in CILK make local decisions to seek out additional pieces of work, whereas all decisions to reallocate work in barrier-based schemes are often made by a central authority or application manager.

Aside from barriers, virtual synchrony [14, 16] and extended virtual synchrony [74] propose communication models for synchronizing large-scale networked systems. These communication systems closely tie together node inter-communication with group membership services. They ensure that a message multicast to a group is either delivered to all participants or to none. Furthermore, they preserve causal message ordering [58] between both individual messages and changes in group membership. This communication model is clearly beneficial to a significant class of distributed systems, including service replication. In the context of distributed application management, however, we have a more modest goal: to provide a convenient synchronization point to loosely coordinate the behavior of applications running on resources in volatile environments. It is important to stress that in this context synchronization is delivered mostly as a matter of convenience rather than as a prerequisite for correctness. Any inconsistency resulting from a relaxed synchronization model is typically detected and corrected by the application, similar to soft-state optimizations in network protocol stacks that improve common-case performance but are not required for correctness. Other related work in consistent group membership/view advancement protocols include Harp [61], Cristian's

group membership protocol [28], and Golding's weakly consistent group membership protocol [42].

The loose synchronization model required for synchronizing computations in distributed applications running in failure-prone environments is related in spirit to a variety of efforts into relaxed consistency models for updates in distributed systems, including Epsilon Serializability [89], the CAP principle [36], Bayou [101], TACT [115], and Delta Consistency [102]. All of these projects recognize the need for relaxed semantics to cope with wide-area inconsistencies and volatility.

2.4 Summary

In this chapter we explored the process of managing distributed applications. By considering three specific examples from different classes of distributed applications that run on PlanetLab, we attempted to extract a general set of requirements for application management. We also described related work that addresses remote execution, application management, workflow management, resource discovery, creation, acquisition, and synchronization. While the set of requirements outlined in this chapter are admittedly challenging, in Chapter 3 we describe Plush, a framework that aims to address these challenges in a streamlined and powerful manner.

2.5 Acknowledgments

Chapter 2, in part, is a reprint of the material as it appears in the ACM Operating Systems Review, January 2006, Albrecht, Jeannie; Tuttle, Christopher; Snoeren, Alex C.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, is a reprint of the material as it appears in the USENIX Annual Technical Conference, 2006, Albrecht, Jeannie; Tuttle, Christopher; Snoeren, Alex C.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, has been submitted for publication of the material as it appears in the Large Installation System Administration Conference, 2007, Albrecht, Jeannie; Braud, Ryan; Dao, Darren; Topilski, Nikolay; Tuttle, Christopher; Snoeren, Alex C.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Design and Implementation of Plush

We now describe Plush, an extensible distributed application controller, designed to address the requirements of large-scale distributed application management discussed in Chapter 2. To directly monitor and control distributed applications, Plush itself must be distributed. Plush uses a client-server architecture, with clients running on each resource involved in the application. The Plush server, called the *controller*, interprets input from the user and sends messages on behalf of the user over an overlay network (typically a tree) to Plush *clients* as shown in Figure 3.1. The controller, typically run from the user's workstation, directs the flow of control throughout the life of the distributed application. The clients run alongside each application component on resources spread across the network and perform actions based upon instructions received from the controller.

Figure 3.2 shows an overview of the Plush controller architecture. Although we do not include a detailed overview of the client architecture, it is symmetric to the controller with only minor differences in functionality. The architecture consists of three main sub-systems: the application specification, core functional units, and user interface. The application specification describes the application. Plush parses the application specification provided by the user and stores internal data structures and objects specific to the application being run. The core functional units then manipulate and act on the objects defined by the application specification to run the application.

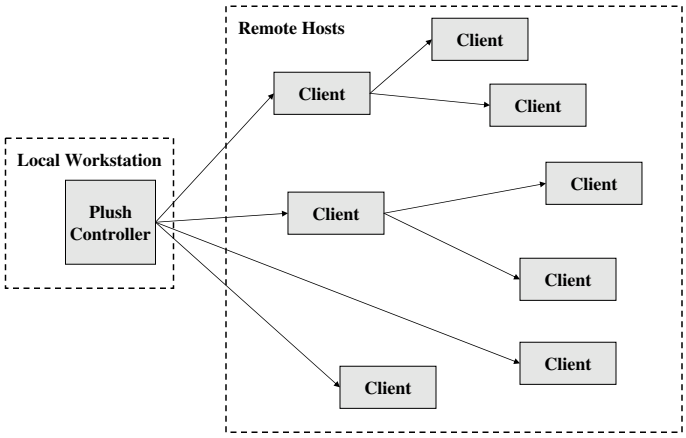

Figure 3.1: Plush controller connected to clients.

The functional units also store authentication information, monitor resources, handle event and timer actions, and maintain the communication infrastructure that enables the controller to query the status of the distributed application on the clients. The user interface provides the functionality needed to interact with the other parts of the architecture, allowing the user to maintain and manipulate the application during execution. In this chapter, we describe the design and implementation details of each of the Plush sub-systems. (Note that the components within the sub-systems are highlighted using boldface throughout the text in the remainder of this chapter.)

3.1 Application Specification

Developing a complete, yet accessible, application specification language was one of the principal challenges in this work. Our approach, which has evolved over the past two years, consists of combinations of five different "building block" abstractions for describing distributed applications. As a preview, Figure 3.3 shows a specific example application that uses these abstractions. We discuss the details of the application later. The five block abstractions are as follows:

1. **Process blocks** - Describe the processes executed on each resource involved in an application. The process abstraction includes runtime parameters, path variables,

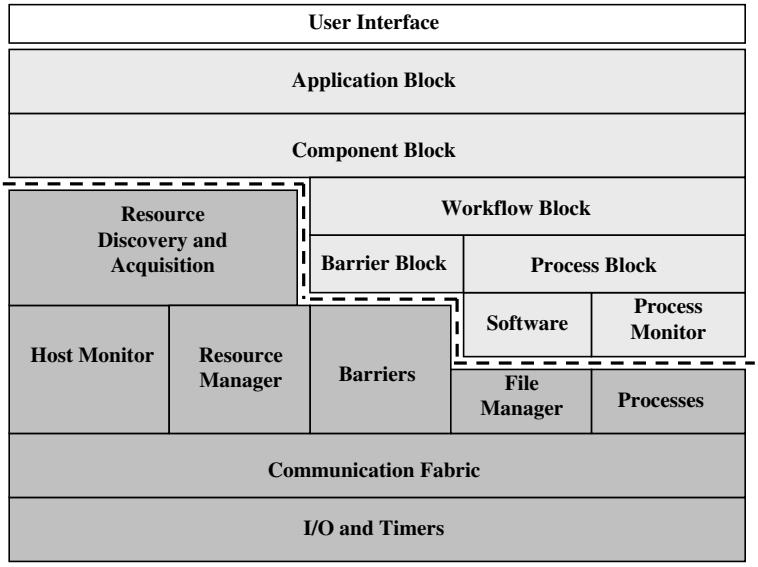

Figure 3.2: The architecture of Plush. The *user interface* is shown above the rest of the architecture and contains methods for interacting with all boxes in the lower subsystems of Plush. Boxes below the user interface and above the dotted line indicate objects defined within the *application specification* abstraction. Boxes below the line represent the *core functional units* of Plush.

runtime environment details, file and process I/O information, and the specific commands needed to start a process on a resource.

2. **Barrier blocks** - Describe the barriers that are used to synchronize the various phases of execution within a distributed application.

3. **Workflow blocks** - Describe the flow of data in a distributed computation, including how the data should be processed. Workflow blocks may contain process and barrier blocks. For example, a workflow block might describe a set of input files over which a process or barrier block will iterate during execution.

4. **Component blocks** - Describe the groups of resources required to run the application, including expectations specific to a set of metrics for the target resources. For example, on PlanetLab the metrics might include maximum load requirements and minimum free memory requirements. Components also define required software configurations, installation instructions, and any authentication information

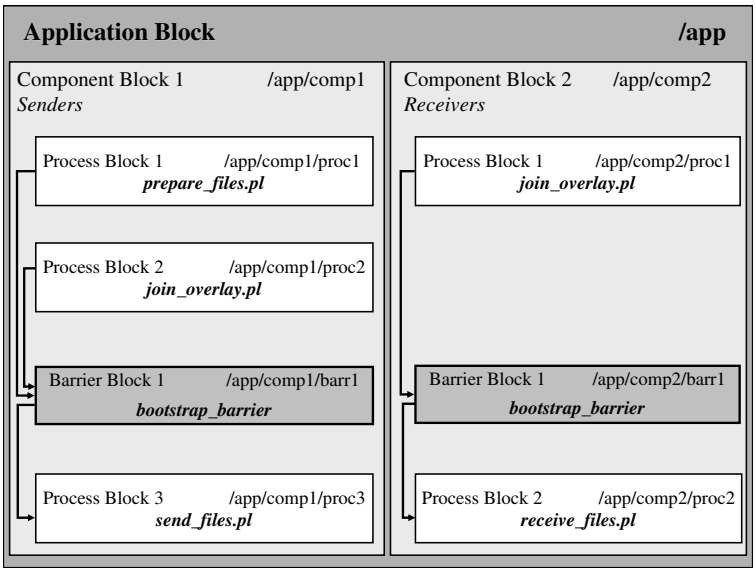

Figure 3.3: Example file-distribution application comprised of application, component, process, and barrier blocks in Plush. Arrows indicate control-flow dependencies. (*i.e.*, Block x → Block y implies that Block x must complete before Block y starts.)

needed to access the resources. Component blocks may contain workflow blocks, process blocks, and barrier blocks.

5. **Application blocks** - Describe high-level information about a distributed application. This includes one or many component blocks, as well as attributes to help automate failure recovery.

To better illustrate the use of these blocks in Plush, consider building the specification for the simple file-distribution application as shown in Figure 3.3. This simple application consists of two groups of resources. One group, the senders, stores the files, and the second group, the receivers, attempts to retrieve the files from the senders. The goal of the application is to experiment with the use of an overlay network to send files from the senders to the receivers using some new file-distribution protocol. In this application, there are two phases of execution. In the first phase, all senders and receivers join the overlay before any transfers begin. Also, the senders must prepare the files for transfer during phase one before the receivers start receiving the files in phase two. In the second phase, the receivers begin receiving the files from the senders. Note that

in the second phase no new senders or receivers are allowed to join the network and participate in the transfer.

The first step in building the corresponding Plush application specification for our new file-distribution protocol is to define an application block. The application block defines general characteristics about the application including the liveness properties and failure detection and recovery options, which determine default failure recovery behavior. For our example, we might choose the behavior "restart-on-failure," which attempts to restart the failed application instance on a single resource, since it is not necessary to abort the entire application across all resources if only a single failure occurs.

The application block also contains one or many component blocks that describe the groups of resources required to run the application. Our application consists of a set of senders and a set of receivers, and two separate component blocks describe the two groups of resources. The sender component block defines the location and installation instructions for the sender software, and includes authentication information to access the resources. Similarly, the receiver component block defines the receiver software package. In our example, it may be desirable to require that all resources in the sender group have a processor speed of at least 1 GHz, and each sender should have sufficient bandwidth for sending files to multiple receivers at once. These types of resource-specific requirements are included in the component blocks.

Within each component block, a combination of workflow, process, and barrier blocks describe the computation that will occur on each resource in the component. Though our example does not employ workflow blocks, they are used in applications where data files must be distributed and iteratively processed. We will consider an example employing a workflow block in Chapter 6.

Plush process blocks describe the specific commands required to execute the application. Most process blocks depend on the successful installation of **software** packages defined in the component blocks. Users specify the commands required to start a given process, and actions to take upon process exit. The exit policies create a Plush

process monitor that oversees the execution of a specific process. Our example defines several process blocks. In the sender component, process blocks define processes for preparing the files, joining the overlay, and sending the files. Similarly, the receiver component contains process blocks for joining the overlay and receiving the files.

Some applications operate in phases, producing output files in early stages that are used as input files in later stages. To ensure all resources start each phase of computation only after the previous phase completes, barrier blocks define loose synchronization semantics between process and workflow blocks. The bootstrap barrier in our example ensures that all receivers and senders join the overlay in phase one before beginning the file transfer in phase two. Note that although each barrier block is uniquely defined within a component block, it is possible for the same barrier to be referenced in multiple component blocks. In our example, we specify barrier blocks in each component block that refer to the same barrier, meaning that the application will wait for all receivers and senders to reach the barrier before allowing either component to start sending or receiving files.

In Figure 3.3, the outer application block contains our two component blocks that run in parallel (since there are no arrows indicating control-flow dependencies between them). Within the component blocks, the different phases are separated by the bootstrap barrier that is defined by Barrier Block 1 in both components. Component Block 1, which describes the senders, contains Process Blocks 1 and 2 that define perl scripts that run in parallel during phase one, synchronize on the bootstrap barrier in Barrier Block 1, and then proceed to Process Block 3 in phase two which sends the files. Component Block 2, which describes the receivers, runs Process Block 1 in phase one, synchronizes on the bootstrap barrier in Barrier Block 1, and then proceeds to Process Block 2 in phase two which runs the process that receives the files from the senders. In the current Plush implementation, the blocks are represented by XML that is parsed by the controller when the application is run. For reference, the XML corresponding to Figure 3.3 is shown in Figure 3.4.

```
<plush>
  <project name="Application Block">
    <application name="Application Block">
      <execution>
        <component_block name="Component Block 1">
          <process_block name="Process Block 1">
            <process name="prepare_files.pl">
              <path>prepare_files.pl</path>
            </process>
          </process_block>
          <process_block name="Process Block 2">
            <process name="join_overlay.pl">
              <path>join_overlay.pl</path>
            </process>
          </process_block>
          <barrier_block name="Barrier Block 1">
            <predecessor name="Process Block 1" />
            <predecessor name="Process Block 2" />
            <barrier name="bootstrap_barrier" />
          </barrier_block>
          <process_block name="Process Block 3">
            <predecessor name="Barrier Block 1" />
            <process name="send_files.pl">
              <path>send_files.pl</path>
            </process>
          </process_block>
        </component_block>
        <component_block name="Component Block 2">
          <process_block name="Process Block 1">
            <process name="join_overlay.pl">
              <path>join_overlay.pl</path>
            </process>
          </process_block>
          <barrier_block name="Barrier Block 1">
            <predecessor name="Process Block 1" />
            <barrier name="bootstrap_barrier" />
          </barrier_block>
          <process_block name="Process Block 2">
            <predecessor name="Barrier Block 1" />
            <process name="receive_files.pl">
              <path>receive_files.pl</path>
            </process>
          </process_block>
        </component_block>
      </execution>
    </application>
  </project>
</plush>
```

Figure 3.4: XML representing the Plush application specification corresponding to Figure 3.3. (Note that the resource definitions are not shown here.)

We designed the Plush application specification to support a variety of execution patterns. With the blocks described above, Plush supports the arbitrary combination of processes, barriers, and workflows, provided that the flow of control between them forms a directed acyclic graph. Using predecessor tags in Plush, users specify

the flow of control and define whether processes run in parallel or sequentially. Arrows between blocks in Figure 3.3, for example, indicate the predecessor dependencies. (Process Blocks 1 and 2 in Component Block 1 will run in parallel before blocking at the bootstrap barrier, and then the execution will continue on to Process Block 3 after the bootstrap barrier releases.) Internally, Plush stores the blocks in a hierarchical data structure, and references specific blocks in a manner similar to referencing absolute paths in a UNIX file system. Figure 3.3 shows the unique path names for each block from our file-distribution example. This naming abstraction also simplifies coordination among resources. Each client maintains an identical local copy of the application specification. Thus, for communication regarding control flow changes, the controller sends the clients messages indicating which "block" is currently being executed, and the clients update their local state information accordingly.

3.2 Core Functional Units

After parsing the block abstractions defined by the user within the application specification, Plush instantiates a set of core functional units to perform the operations required to configure and deploy the distributed application. Figure 3.2 shows these units as shaded boxes below the dotted line. The functional units manipulate the objects defined in the application specification to manage distributed applications. In this section, we describe the role of each of these units.

Starting at the highest-level, the Plush **resource discovery and acquisition** unit uses the resource definitions in the component blocks to locate and create (if necessary) resources on behalf of the user. The resource discovery and acquisition unit is responsible for obtaining a valid set, called a *matching*, of resources that meet the application's demands. To determine this matching, Plush may either call an existing external service to construct a resource pool, such as SWORD [3, 77] for PlanetLab, or use a statically defined resource pool based on information provided by the user. The Plush *resource matcher* then uses the resources in the resource pool to create a match-

ing for the application. We discuss this process in detail in Chapter 4. All resources involved in an application run a Plush **host monitor** that periodically publishes information about the resource. The resource discovery and acquisition unit may use this information to help find the best matching. Upon acquiring a resource, a Plush **resource manager** stores the lease, token, or any necessary user credential needed for accessing that resource to allow Plush to perform actions on behalf of the user in the future.

The remaining functional units in Figure 3.2 are responsible for application deployment and maintenance. These units connect to resources, install required software, start the execution, and monitor the execution for failures. One important functional unit used for these operations is the Plush **barrier manager**, which provides advanced synchronization services for Plush and the application itself. In our experience, traditional barriers [50] are not well suited for volatile, wide-area network conditions; the semantics are simply too strict. Instead, Plush uses partial barriers (described in detail in Chapter 5), which are designed to perform better in failure-prone environments through the use of relaxed synchronization semantics.

The Plush **file manager** handles all files required by a distributed application. This unit contains information regarding software packages, file transfer methods, installation instructions, and workflow data files. The file manager is responsible for preparing the physical resources for execution using the information provided by the application specification. It monitors the status of file transfers and installations, and if it detects an error or failure, the controller is notified and the resource discovery and acquisition unit may be required to find a new resource to replace the failed one.

Once the resources are prepared with the necessary software, the application deployment phase completes by starting the execution. This is accomplished by starting a number of processes on the resources. Plush **processes** are defined within process blocks in the application specification. A Plush process is an abstraction for standard UNIX processes that run on multiple resources. Processes require information about the runtime environment needed for an execution including the working directory, path, environment variables, file I/O, and the command-line arguments.

The two lowest layers of the Plush architecture consist of a **communication fabric** and the **I/O and timer** subsystems. The communication fabric handles passing and receiving messages among Plush overlay participants. Participants communicate over TCP connections. The default topology for a Plush overlay is a star, although we also provide support for tree topologies for increased scalability (see Section 3.3.2). In the case of a star topology, all clients connect directly to the controller, which allows for quick failure detection and recovery. The controller sends messages to the clients instructing them to perform certain actions. When the clients complete their tasks, they report back to the controller for further direction. The communication fabric at the controller knows what resources are involved in a particular application instance, so that the appropriate messages reach all necessary resources.

At the bottom of all of the other units is the Plush I/O and timer abstraction. As messages are received in the communication fabric, message handlers fire events. These events are sent to the I/O and timer layer and enter a queue. The event loop pulls events off the queue, and calls the appropriate event handler. Timers are a special type of event in Plush that fire at a predefined instant.

3.3 Fault Tolerance and Scalability

Two of the biggest challenges that we encountered during the design of Plush was being robust to failures and scaling to hundreds of resources spread across the wide-area. In this section we explore how Plush supports fault tolerance and scalability.

3.3.1 Fault Tolerance

Plush must be robust to the variety of failures that occur during application execution. When designing Plush, we aimed to provide the functionality needed to detect and recover from most failures without involving the user running the application. Rather than enumerate all possible failures that may occur, we discuss how Plush handles three common failure classes—process, resource, and controller failures.

Table 3.1: Process exit policies in Plush.

Exit Policy	**Description**
POLICY_END_APPLICATION	End the application with success
POLICY_FAIL_APPLICATION	End the application with failure
POLICY_RESTART_APPLICATION	Restart the entire application
POLICY_RESTART_PROCESS	Restart only the process
POLICY_CONTINUE	Continue to next step in workflow
POLICY_IGNORE	Log the exit, but do nothing

Process failures. When a resource starts a process defined in a process block, Plush attaches a process monitor to the process. The role of the process monitor is to catch any signals raised by the process, and to react appropriately. When a process exits either due to successful completion or error, the process monitor sends a message to the controller indicating that the process has exited, and includes its exit status. Plush defines a default set of behaviors that occur in response to a variety of exit codes (although these can be overridden within an application specification). The default behaviors include ignoring the failure, restarting only the failed process, restarting the entire application, or aborting the entire application. Table 3.1 lists the process exit policies supported by Plush, as well as a description of their behavior.

In addition to process failures, Plush also allows users to monitor the status of a process that is still running through a specific type of process monitor called a **liveness monitor**, whose goal is to detect misbehaving and unresponsive processes that get stuck in loops and never exit. This is especially useful in the case of long-running services that are not closely monitored by the user. To use the liveness monitor, the user specifies a script and a time interval in the process block of the application specification. The liveness monitor wakes up once per time interval and runs the script to test for the liveness of the application, returning either success or failure. If the test fails, Plush kills the process, causing the process monitor to be alerted and inform the controller.

Resource failures. Detecting and reacting to process failures is straightforward since the controller is able to communicate information to the client regarding the appropriate recovery action. When a resource fails, however, recovering is more difficult. A resource may fail for a number of reasons, including network outages, hardware problems, and power loss. Under all of these conditions, the goal of Plush is to quickly detect the problem and reconfigure the application with a new set of resources to continue execution. The Plush controller maintains a list of the last time successful communication occurred with each connected client. If the controller does not hear from a client within a specified time interval, the controller sends a ping to the client. If the controller does not receive a response from the client, Plush assumes resource failure. Reliable failure detection is an active area of research; while the simple technique we employ has been sufficient thus far, we certainly intend to leverage advances in this space where appropriate.

There are three possible actions in response to a resource failure: restart, rematch, and abort. By default, the controller tries all three actions in order. The first and easiest way to recover from a resource failure is to simply reconnect and restart the application on the failed resource. This technique works if the resource experiences a temporary power or network outage, and is only unreachable for a short period of time. If the controller is unable to reconnect to the resource, the next option is to rematch in an attempt to replace the failed resource with a different resource. In this case, Plush reruns the resource matcher to find a new resource. Depending on the application, the entire execution may need to be restarted across all resources after the new resource joins the Plush control overlay, or the execution may only need to be started on the new resource. If the controller is unable to find a new resource to replace the failed resource and the application description specifies a fixed number of required resources, Plush then finally aborts the entire application.

In some applications, it is desirable to mark a resource as failed when it becomes overloaded or experiences poor network connectivity. The Plush host monitor that runs on each resource is responsible for periodically informing the controller about

each resource's status. If the controller determines that the performance is less than the application tolerates, it marks the resource as failed and attempts to rematch. This functionality is a preference specified at startup. Although Plush currently monitors host-level metrics including load and free memory, the technique is easily extended to encompass sophisticated application-level expectations of resource viability.

Controller failures. Because the controller is responsible for managing the flow of control across all connected clients, recovering from a failure at the controller is difficult. One solution is to use a simple primary-backup scheme, where multiple controllers increase reliability. All messages sent from the clients and primary controller are sent to the backup controllers as well. If a pre-determined amount of time passes and the backup controllers do not receive any messages from the primary, the primary is assumed to have failed. The first backup becomes the primary, and execution continues.

This strategy has several drawbacks. First, it causes extra messages to be sent over the network, which limits the scalability of Plush. Second, this approach does not perform well when a network partition occurs. During a network partition, multiple controllers may become the primary controller for subsets of the clients initially involved in the application. Once the network partition is resolved, it is difficult to reestablish consistency among all clients and resources. While we have implemented a version of this architecture, we are currently exploring other possibilities for handling faults at the controller.

3.3.2 Scalability

In addition to fault tolerance, an application controller designed for large-scale environments must scale to hundreds or even thousands of participants. Unfortunately there is a tradeoff between performance and scalability. The solutions that perform the best at moderate scale typically provide less scalability than solutions with lower performance. To balance scalability and performance, Plush provides users with two topological alternatives for the structure of the control overlay that offer varying levels of scalability and performance.

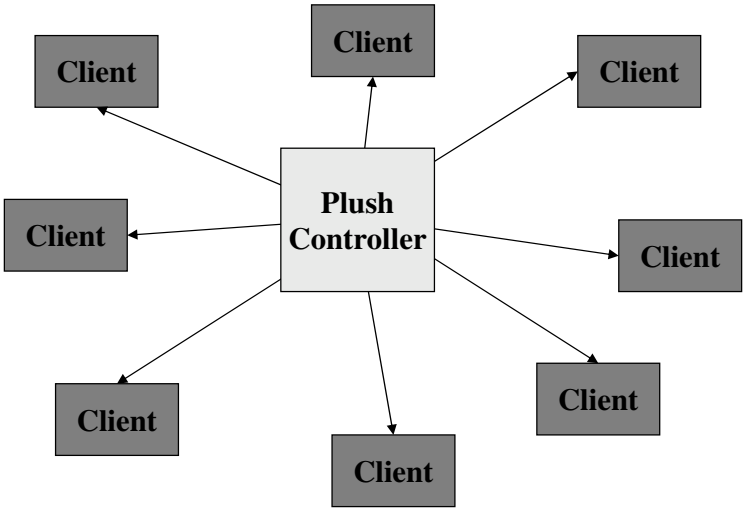

Figure 3.5: Clients connected to Plush controller in star topology.

By default, all Plush clients connect directly to the controller forming a star topology (Figure 3.5). This architecture scales to approximately 300 resources, limited by the number of file descriptors allowed per process on the controller machine in addition to the bandwidth, CPU, and latency required to communicate with all connected clients. The star topology is easy to maintain, since all clients connect directly to the controller. In the event of a resource failure, only the failed resource is affected. Further, the time required for the controller to exchange messages with clients is short due to the direct connections.

At larger scales, network and file descriptor limitations at the controller become a bottleneck. To address this, Plush also supports tree topologies (Figure 3.6). In an effort to reduce the number of hops between the clients and controller, Plush constructs “bushy” trees, where the depth of the tree is small and each node in the tree has many children. The controller is the root of the tree. The children of the root are chosen to be well-connected and historically reliable resources whenever possible. Each child of the root acts as a “proxy controller” for the resources connected to it. These proxy controllers send invitations and receive join messages from other resources, reducing the total number of messages sent back to the root controller. Important messages, such as failure notifications, are still sent back to the root controller. Using the tree topology, we

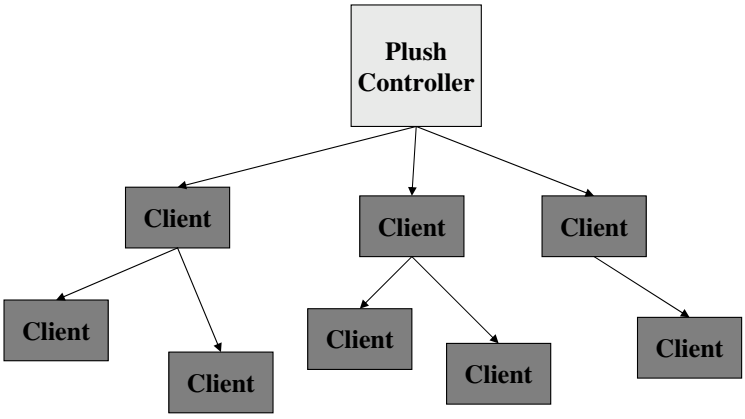

Figure 3.6: Clients connected to Plush controller in tree topology.

have been able to use Plush to manage an application running on 1000 ModelNet virtual hosts, as well as an application running on 500 PlanetLab clients. We believe that Plush has the ability to scale by perhaps another order of magnitude with the current design.

While the tree topology has many benefits over the star topology, it also introduces several new problems with respect to resource failures and tree maintenance. In the star topology, a resource failure is simple to recover from since it only involves one resource. In the tree topology, however, if a non-leaf resource fails, all children of the failed resource must find a new parent. Depending on the number of resources affected, a reconfiguration involving several resources often has a significant impact on performance. Our current implementation tries to minimize the probability of this type of failure by making intelligent decisions during tree construction. For example, in the case of ModelNet, many virtual hosts (and Plush clients) reside on the same physical machine. When constructing the tree in Plush, only one client per physical machine connects directly to the controller and becomes the proxy controller. The remaining clients running on the same physical machine become children of the proxy controller. In the wide area, similar decisions are made by placing resources that are geographically close together under the same parent. This decreases the number of hops and latency between leaf nodes and their parent, minimizing the chance of network failures.

3.4 Running an Application

In this section, we discuss how the architectural components of Plush interact to run a distributed application. When starting Plush, the user's workstation becomes the controller. The user submits an application specification to the Plush controller. The controller parses the specification and internally creates the objects shown above the dotted line in Figure 3.2.

After parsing the application specification, the controller runs the resource discovery and acquisition unit to find a suitable set of resources that meet the requirements specified in the component blocks. Upon locating the necessary resources, the resource manager stores the required access and authentication information. The controller then attempts to connect to each resource. If the Plush client is not already running, the controller initiates a bootstrapping procedure to copy the Plush client binary to the resource, and then uses SSH to connect to the resource and start the client process. Once the client process is running, the controller establishes a TCP connection to the resource, and transmits an `INVITE` message to the resource to join the Plush overlay (which is either a star or tree as discussed in Section 3.3.2).

If a Plush client agrees to run the application, the client sends a `JOIN` message back to the controller accepting the invitation. Next, the controller sends a `PREPARE` message to the new client, which contains a copy of the application specification (XML representation). The client parses the application specification, starts a local host monitor, sends a `PREPARED` message back to the controller, and waits for further instruction. Once enough resources join the Plush overlay and agree to run the application, the controller initiates the beginning of the application deployment stage by sending a `GO` message to all connected clients. The file managers then begin installing the requested software and preparing the resources for execution.

In most applications, the controller instructs the resources to begin execution after all resources have completed the software installation. (Note that synchronizing the beginning of the execution is not required if the application does not need all resources

to start simultaneously.) Since each client creates an exact copy of the controller's application specification, the controller and clients can exchange messages about the application's progress using the block naming abstraction (*i.e.*, /app/comp1/proc1) to identify the status of the execution. For barriers, a barrier manager running on the controller determines when it is appropriate for resources to be released from the barriers in the application.

If a client detects a failure, the client notifies the controller, and the controller attempts to recover from the failure according to the actions enumerated in the application specification. Since many failures are application-specific, Plush exports optional callbacks to the application itself to determine the appropriate reaction for some failure conditions. When the application completes (or upon a user command), Plush stops all associated processes, transfers output data back to the controller's local disk if desired, performs user-specified cleanup actions on the resources, disconnects the resources from the Plush overlay by closing the TCP connection, and stops the Plush client processes.

3.5 User Interface

Plush aims to support a variety of applications being run by users with a wide range of expertise in building and managing distributed applications. Thus, Plush provides three interfaces which each provide users with techniques for interacting with their applications. We describe the functionality of each user interface in this section.

In Figure 3.2, the user interface is shown above all other parts of Plush. In reality, the user interacts with every box shown in the figure through the user interface. For example, the user can force the resource discovery and acquisition unit to find a new set of resources by issuing a command through one of the user interfaces. We designed Plush in this way to give the user maximum control over the application. At any stage of execution, the user can override a default Plush behavior. The overall effect is a customizable application controller that has the ability to support a variety of distributed applications and computing environments.

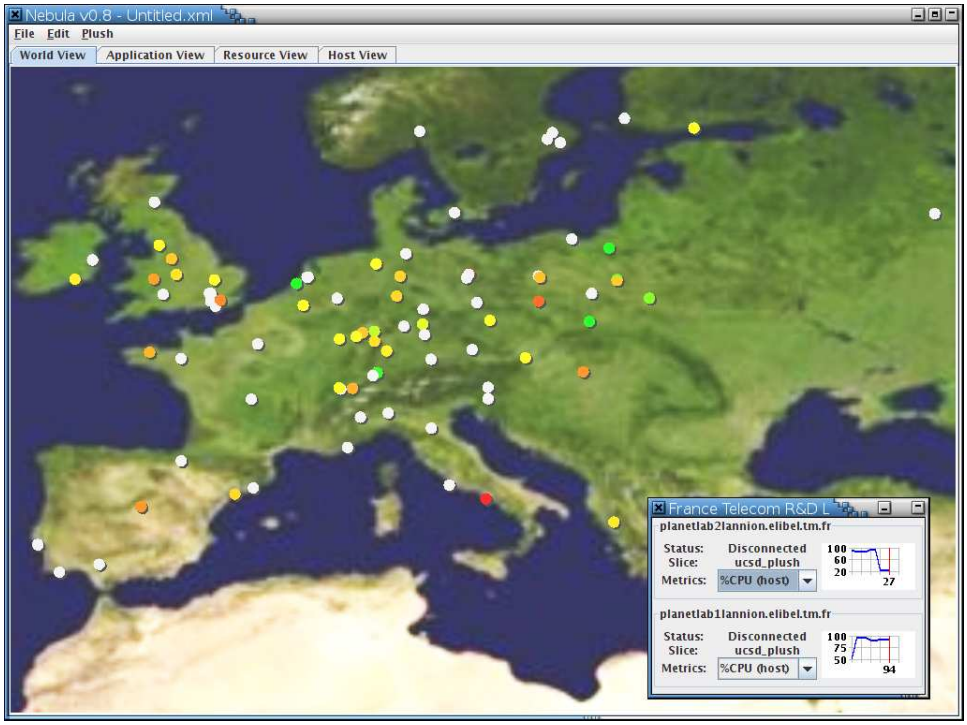

Figure 3.7: Nebula World View Tab showing an application running on PlanetLab sites in Europe. Different colored dots indicate sites in various stages of execution. The window in the bottom right corner displays CPU usage information about selected hosts.

3.5.1 Graphical User Interface

In an effort to simplify the creation of application specifications and help visualize the status of executions running on resources around the world, we implemented a graphical user interface for Plush called Nebula. In particular, we designed Nebula (as shown in Figure 3.7) to simplify the process of specifying and managing applications running across PlanetLab. Plush obtains data from the PlanetLab Central (PLC) database to determine what hosts a user has access to, and Nebula uses this information to plot the sites on the map. To start using Nebula, users have the option of building their Plush application specification from scratch or loading a preexisting XML document representing an application specification. Upon loading the application specification, the user runs the application by clicking the Run button from the Plush toolbar, which causes Plush to start locating and acquiring resources.

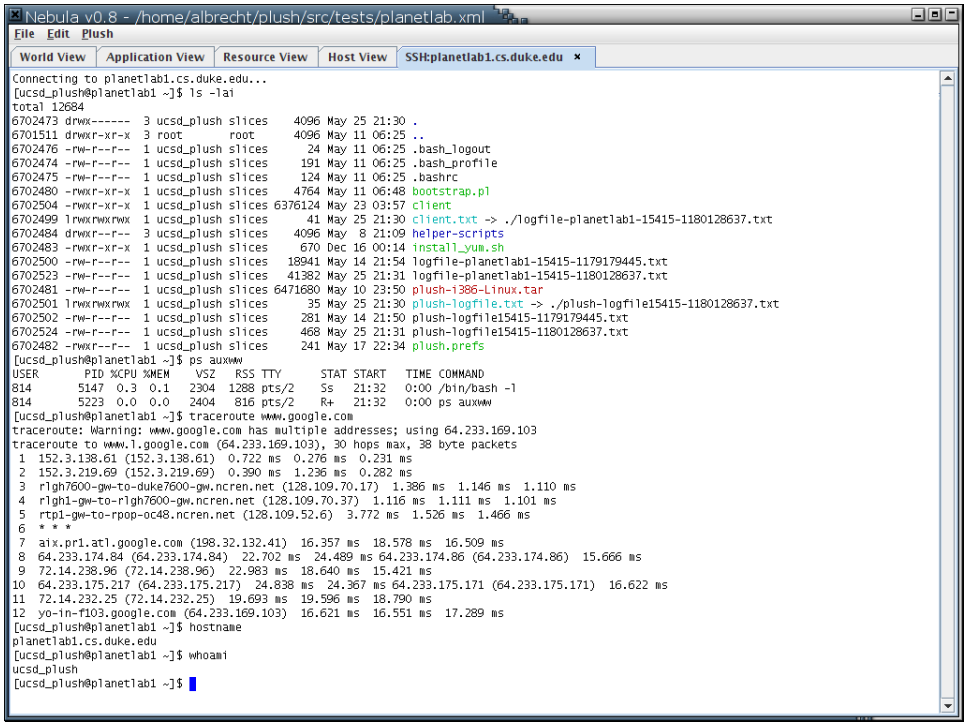

Figure 3.8: Nebula SSH tab displaying an SSH connection to a PlanetLab host.

The main Nebula window contains four tabs that show different information about the user's application. In the "World View" tab, users see an image of a world map with colored dots indicating PlanetLab hosts. Different colored dots on the map indicate sites involved in the current application. In Figure 3.7, the colored dots (ranging from red to green) show PlanetLab sites involved in the current application. The grey dots are other available PlanetLab sites that are not currently being used by Plush. As the application proceeds through the different phases of execution, the sites change color, allowing the user to visualize the progress of their application. When failures occur, the impacted sites turn red, giving the user an immediate visual indication of the problem. Similarly, green dots indicate that the application is executing correctly. If a user wishes to establish an SSH connection directly to a particular resource, they can simply right-click on a host in the map and choose the SSH option from the pop-up menu. This opens a new tab in Nebula containing an SSH terminal to the host, as shown in Figure 3.8. Users can also mark hosts as failed by right-clicking and choosing the Fail option

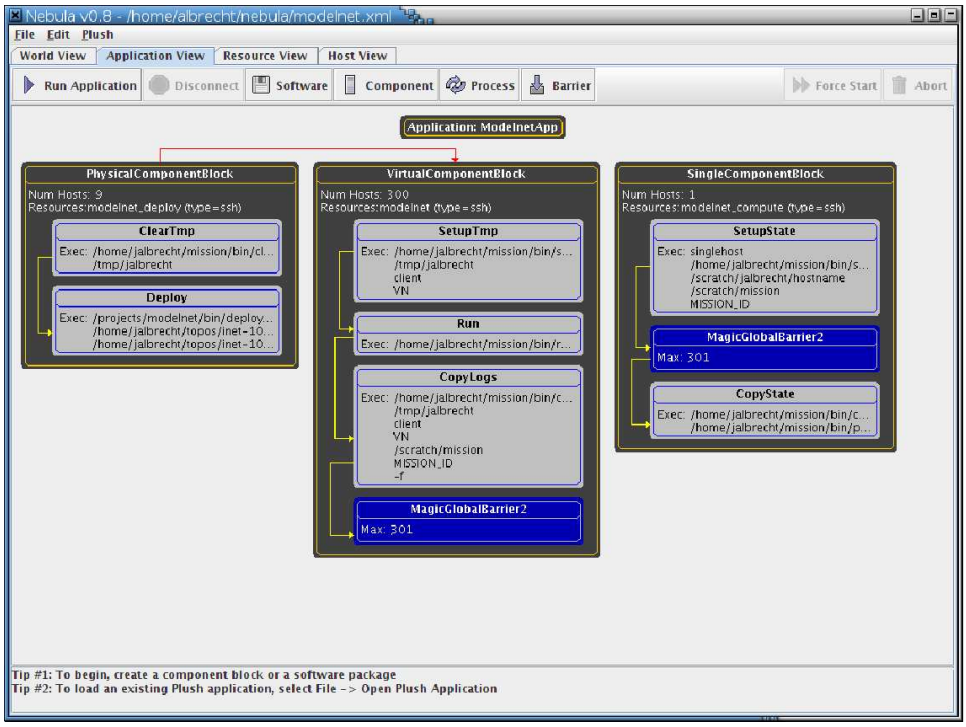

Figure 3.9: Nebula Application View tab displaying a Plush application specification.

from the pop-up menu. Failing a host is helpful if the user is able to determine failure more quickly than Plush's automated techniques. Failed hosts are completely removed from the execution.

Users retrieve more detailed usage statistics and monitoring information about specific hosts (such as CPU load, free memory, or bandwidth usage) by double clicking on the individual sites in the map. This opens a second window that displays real-time graphs based on data retrieved from resource monitoring tools, as shown in the bottom right corner of Figure 3.7. The second smaller window displays a graph of the CPU or memory usage, and the status of the application on each host. Plush currently provides built-in support for monitoring CoMon [81] data on PlanetLab machines, which is the source of the CPU and memory data. Additionally, if the user wishes to view the CPU usage or percentage of free memory available across all hosts, there is a menu item under the PlanetLab menu that changes the colors of the dots on the map such that red means high CPU usage or low free memory, and green indicates low CPU usage or high

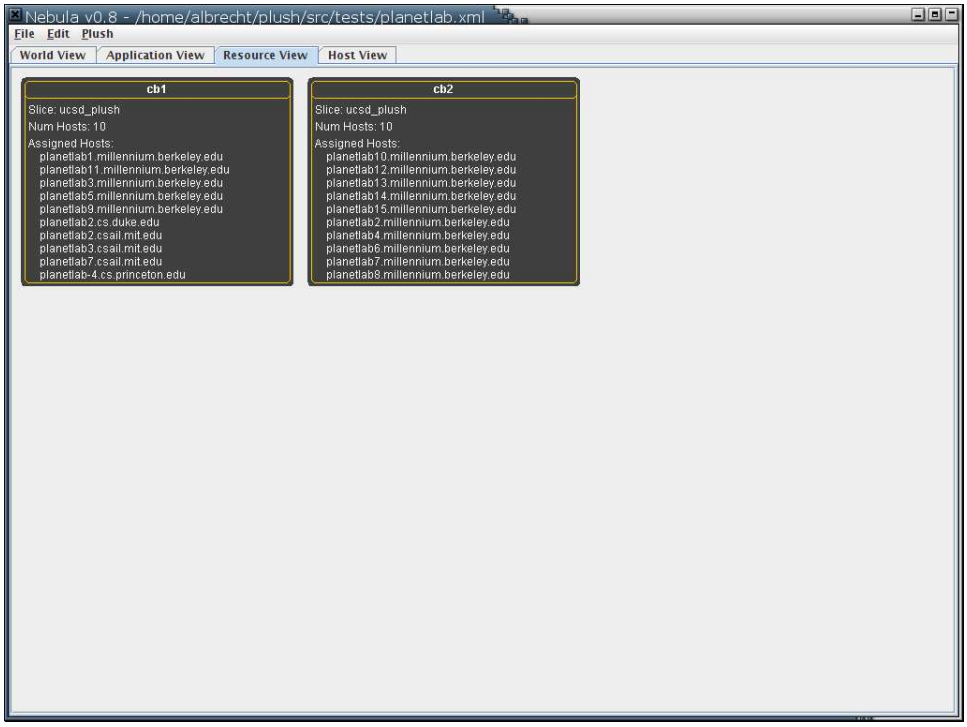

Figure 3.10: Nebula Resource View tab showing resources involved in an application.

free memory. Users can also add and remove hosts from their PlanetLab slice directly by highlighting regions of the map and choosing the appropriate menu option from the PlanetLab menu. Additionally, users can renew their PlanetLab slice from Nebula.

The second tab in the Nebula main window is the “Application View.” The Application View tab, shown in Figure 3.9, allows users to build Plush application specifications using the blocks described in Section 3.1. Alternatively, users may load an existing XML file describing an application specification by choosing the Load Application menu option under the File menu. There is also an option to save a new application specification to an XML file for later use. After creating or loading an application specification, the Run button located on the Application View tab starts the application. The Plush blocks in the application specification change to green during the execution of the application to indicate progress. After an application begins execution, users have the option to “force” an application to skip ahead to the next phase of execution (which corresponds to releasing a synchronization barrier), or aborting an application to termi-

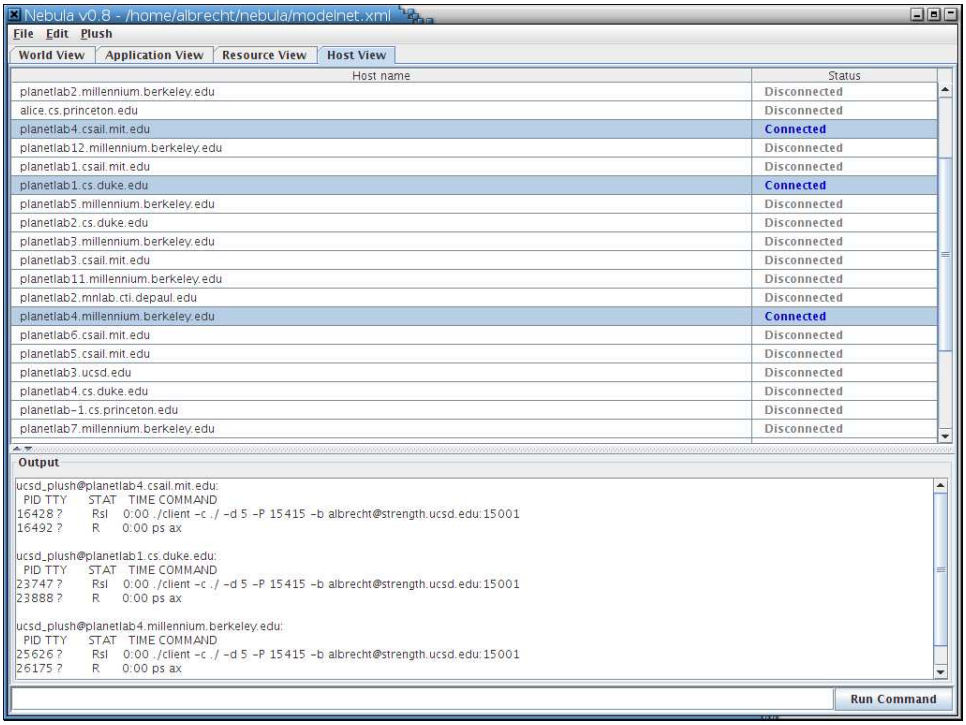

Figure 3.11: Nebula Host View tab showing PlanetLab resources. This tab allows users to select multiple hosts at once and run shell commands on the selected resources. The text-box at the bottom shows the output from the shell commands.

nate execution across all resources. Once the application aborts or completes execution, the user may either save their application specification, disconnect from the Plush communication overlay, restart the same application, or load and run a new application by choosing the appropriate option from the File menu.

The third tab is the “Resource View” tab. This tab is blank until an application starts running. During execution, this tab lists the specific PlanetLab hosts that are involved in the execution. If failures occur during execution, the list of hosts is updated dynamically, such that the Resource View tab always contains an accurate listing of the resources that are in use. The resources are separated into components, so that the user knows which resources are assigned to which tasks in their application. A screenshot showing the Resource View tab is shown in Figure 3.10.

The fourth tab in Nebula is called the “Host View” tab, shown in Figure 3.11. This tab contains a table that displays the hostname of all available PlanetLab resources.

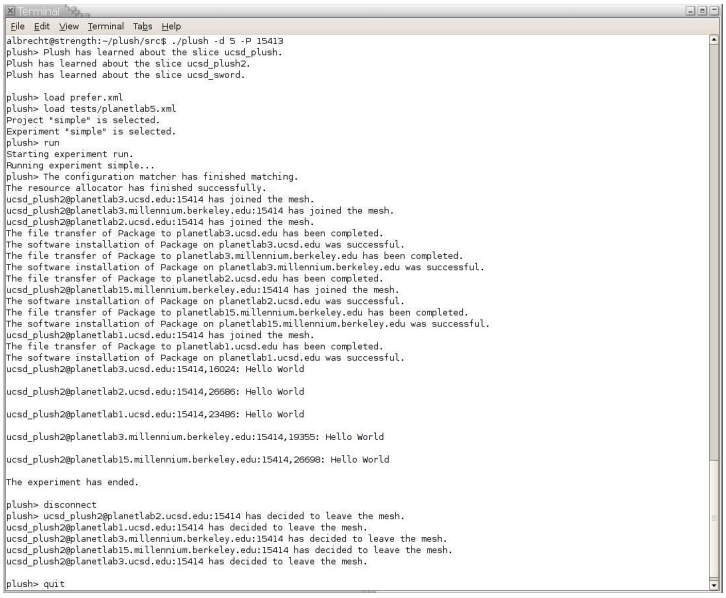

Figure 3.12: Screenshot of Plush command-line interface.

The purpose of the Host View tab is to give users another alternative to visualize the status of an executing application. In the right column, the status of the host is shown. Each host's status corresponds to the color of the host's dot in the World View tab. This tab also allows users to run shell commands simultaneously on several resources, and view the output. As shown in Figure 3.11, users can select multiple hosts as once, run a command, and the output is displayed in the text-box at the bottom of the window. Note that PlanetLab hosts do not have to be involved in an application in order to take advantage of this feature. Plush will connect to any available resources and run commands on behalf of the user. Just as in the World View tab, right-clicking on hosts in the Host View tab opens a pop-up menu that enables users to SSH directly to the hosts.

3.5.2 Command-line Interface

Motivated by the popularity and familiarity of the shell interface in UNIX, Plush further streamlines the develop-deploy-debug cycle for distributed application management through a simple command-line interface where users can deploy, run, monitor, and debug their distributed applications running on hundreds of resources. The Plush command-line (Figure 3.12) combines the functionality of a distributed shell with

Table 3.2: Sample Plush terminal commands.

Command	Description
`load <filename>`	Read an XML application specification
`connect <resource>`	Connect to a Plush client on a resource
`disconnect`	Close all open client connections
`info nodes`	Print summary information about all known resources
`info mesh`	Print the communication overlay status (membership)
`info control`	Print the controller's state information
`info install`	Print summary information on pending installations
`run`	Start executing the application (after loading specification)
`shell <quoted string>`	Run "quoted string" as a shell command on all resources
`slice list <slice_name>`	Show information about a PlanetLab slice
`slice renew <slice_name>`	Renew the specified PlanetLab slice

the power of an application controller to provide a robust execution environment for users to run their applications. From a user's standpoint, the Plush terminal looks just like a shell. Plush supports several commands for monitoring the state of an execution, as well as commands for manipulating the application specification during execution. Table 3.2 shows a subset of the available commands.

3.5.3 Programmatic Interface

Many commands that are available via the Plush command-line interface are also exported via an XML-RPC interface to deliver similar functionality as the command-line to those who desire programmatic access. Using XML-RPC, Plush can be scripted and used for remote execution and automated application management. External services for resource discovery, creation, and acquisition can also communicate with Plush using XML-RPC. These external services have the option of registering themselves with Plush so that the controller can send callbacks to XML-RPC clients when various actions occur during the execution.

```
class PlushXmlRpcServer extends XmlRpcServer {
  void plushAddNode(HashMap properties);
  void plushRemoveNode(string hostname);
  string plushTestConnection();
  void plushCreateResources();
  void plushLoadApp(string filename);
  void plushRunApp();
  void plushDisconnectApp(string hostname);
  void plushQuit();
  void plushFailHost(string hostname);
  void setXmlRpcClientUrl(string clientUrl);
}

class PlushXmlRpcCallback extends XmlRpcClient {
  void sendPlanetLabSlices();
  void sendSliceNodes(string slice);
  void sendAllPlanetLabNodes();
  void sendApplicationExit();
  void sendHostStatus(string host);
  void sendBlockStatus(string block);
  void sendResourceMatching(HashMap matching);
}
```

Figure 3.13: Plush XML-RPC API.

Figure 3.13 shows the Plush XML-RPC API. The functions shown in the `PlushXmlRpcServer` class are available to users who wish to access Plush programmatically in scripts, or for external resource discovery and acquisition services that need to add and remove resources from the Plush resource pool. The `plushAddNode(HashMap)` and `plushRemoveNode(string)` calls add and remove nodes from the resource pool, respectively. `setXmlRpcClientUrl(string)` registers XML-RPC clients for callbacks, while `plushTestConnection()` simply tests the connection to the Plush server and returns "Hello World." The remaining function calls in the class mimic the behavior of the corresponding command-line operations. In Chapter 4 we will examine some specific uses of this API within the context of different resource management frameworks.

Aside from resource discovery and acquisition services, the XML-RPC API allows for the implementation of different user interfaces for Plush. Since almost all

of the Plush terminal commands are available as XML-RPC function calls, users are free to implement their own customized environment specific user interface without understanding or modifying the internals of the Plush implementation. This is beneficial because it gives the users more flexibility to develop in the programming language of their choice. Most mainstream programming languages have support for XML-RPC, and hence users are able to develop interfaces for Plush in any language, provided that the chosen language is capable of handling XML-RPC. To increase the functionality and simplify the development of these interfaces, the Plush XML-RPC server has the ability to make callbacks to programs that register with the Plush controller via `setXmlRpcClientUrl(string)`. Some of the more common callback functions are shown in the bottom of Figure 3.13 in class `PlushXmlRpcCallback`. Note that these callbacks are only useful if the client implements the corresponding functions.

3.6 Implementation Details

Plush is a publicly available software package [86]. The Plush codebase consists of over 60,000 lines of C++ code. The same code is used for the Plush controller and client processes, although there are minor differences in functionality within the code. Plush depends on several C++ libraries, including those provided by xmlrpc-c, curl, xml2, zlib, math, openssl, readline, curses, boost, and pthreads. The command-line interface also depends on packages for lex and yacc (we use flex and bison). For optimal performance, we recommend the use of the Native POSIX Threads Library (NPTL) in Linux environments, as well as the ares package for asynchronous DNS lookups. In addition to the main C++ codebase, Plush uses several simple perl scripts for interacting with the PlanetLab Central database and bootstrapping resources. These perl scripts require the Frontier::Client and Crypt::SSLeay perl modules. Plush runs on most UNIX-based platforms, including Linux, FreeBSD, and Mac OS X, and a single Plush controller can manage clients running on different operating systems. The only prerequisite for using Plush on a resource is the ability to SSH to the resource.

One challenge that arises when running Plush on different platforms is inconsistencies in execution environments. This is particularly problematic when executables are dynamically linked to system libraries. In general we found that statically linking the client executable that runs on potentially remote resources helps to solve this problem in most cases. Statically linking the client is especially helpful on PlanetLab, since PlanetLab machines do not include many libraries by default. One caveat to supporting statically linking executables, however, is the inability to use architecture-specific system calls, such as some cryptographic random number generators. The Plush codebase explicitly avoids architecture-specific system calls to ensure that statically linking executables function correctly.

Nebula consists of approximately 25,000 lines of Java code. Nebula communicates with Plush using the XML-RPC interface described in Section 3.5.3. XML-RPC is implemented in Nebula using the Apache XML-RPC client and server packages. In addition, Nebula uses the JOGL implementation of the OpenGL graphics package for Java. Since Nebula uses OpenGL, we highly recommend enabling video card hardware acceleration for optimal performance. Nebula runs in any computing environment that supports Java, including Windows, Linux, FreeBSD, and Mac OS X among others. Note that since Nebula and Plush communicate solely via XML-RPC, it is not necessary to run Nebula on the same physical machine as the Plush controller. When starting Nebula, users have the option of either starting a local Plush controller or specifying a remote Plush controller process.

3.7 Summary

Motivated by the requirements of Chapter 2, in this chapter we described the design and implementation of Plush, a distributed application management framework. The three main sub-systems of Plush are the core functional units, application specification, and user interface. This chapter described how these sub-systems work together to manage distributed applications, and also discussed how the Plush architecture achieves

fault tolerance and scalability. Finally, we explored three different user interfaces to Plush, which give users several different options for interacting with their applications running on distributed resources.

3.8 Acknowledgments

Chapter 3, in part, has been submitted for publication of the material as it appears in the Large Installation System Administration Conference, 2007, Albrecht, Jeannie; Braud, Ryan; Dao, Darren; Topilski, Nikolay; Tuttle, Christopher; Snoeren, Alex C.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Resource Matcher

Chapter 2 describes the role of resource discovery, creation, and acquisition in the context of application management. To summarize, the main responsibility of a resource discovery and acquisition service is to find a set of resources (called a *matching* in Plush) that meet the application's resource demands. One goal in the design of Plush is to create an architecture flexible enough to work in a variety of computing environments with different types of resources. Thus, rather than reinvent the functionality of existing resource discovery and acquisition services for each target environment, we instead employ an extensible resource discovery and acquisition unit (as shown in Figure 3.2) that supports a variety of resources. This is often accomplished by using the Plush XML-RPC interface for adding and removing resources from the application's *resource pool*. The Plush *resource matcher* then uses the resources in the resource pool and the application's requirements as defined in the application specification to create a resource matching.

In this chapter, we examine how the Plush resource matcher interacts with different types of resources provided by external services to construct a valid matching and run applications. In particular, we examine three distinct types of resources: physical PlanetLab hosts, emulated ModelNet resources, and Xen virtual machines. We also describe the external services used to manage these resources, including SWORD for PlanetLab hosts, Mission for ModelNet resources, and Shirako and Usher for Xen

virtual machines. In doing so, we evaluate how effective Plush is in achieving our goal of supporting execution in a variety of computing environments.

4.1 Plush Resource Pools

Before discussing how a matching is created, we first describe how resource pools are constructed in Plush. Recall that a resource in Plush is a (virtual or physical) machine that can host an application on behalf of the user (*i.e.*, the person running the application). Plush assumes that all resources are accessible via SSH, and requires that passphrase-less authentication has been established *a priori* by using a combination of ssh-agent and public key distribution. A resource pool is simply a grouping of resources that are available to the user and can be reached via passphrase-less SSH authentication.

The simplest way to define a resource pool in Plush is by creating a resource directory file (typically called directory.xml) that lists available resources. This file is read by the Plush controller at startup, and internally Plush creates a *Node* object for each resource. A Node in Plush contains a username for logging into the resource, a fully qualified hostname, the port on which the Plush client will run, and a group name. The purpose of the group name is to give users the ability to classify resources into different categories based on application-specific requirements. We discuss how this name is used when creating a matching in the next section.

The resource file also contains a special section for defining PlanetLab hosts. Rather than specifically defining which PlanetLab hosts a user has access to, the directory file instead lists which *slices* are available to the user. In addition to the slice names, the user specifies their login to PlanetLab Central (PLC), as well as a mapping (called the *portmap*) from slice names to port numbers. At startup, Plush uses this login information to contact PLC directly via XML-RPC. The PLC database returns a list of hostnames that have been assigned to each available slice. The Plush controller uses this information to create a Node object for each PlanetLab host available to the user. The username that is used for logging in to PlanetLab hosts is the slice name, and the port is

```
<?xml version="1.0" encoding="UTF-8"?>
<plush>
  <resource_manager type="planetlab">
    <user>jalbrecht@cs.ucsd.edu</user>
    <port_map slice="ucsd_plush" port="15415"/>
    <port_map slice="ucsd_sword" port="15416"/>
  </resource_manager>
  <resource_manager type="ssh">
    <node hostname="sysnet80.ucsd.edu:15420" user="albrecht" group="local">
    <node hostname="sysnet81.ucsd.edu:15420" user="albrecht" group="local">
    <node hostname="sysnet82.ucsd.edu:15420" user="albrecht" group="local">
  </resource_manager>
</plush>
```

Figure 4.1: Plush resource directory file.

determined using the portmap defined in the directory file. The group name is set to be the same as the slice name by default for PlanetLab hosts. A sample directory.xml file is shown in Figure 4.1.

In addition to the resources defined in a Plush resource directory file, resources are also added and removed by external services at any point during an application's execution. This is typically accomplished using the Plush XML-RPC interface described in Chapter 3. External services that create virtual resources dynamically based on an application's needs, for example, contact the Plush controller with new available resources, and Plush adds these resources to the user's resource pool. If these resources become unavailable, the external service calls Plush again, and Plush subsequently removes the resources from the resource pool. Note that this may involve stopping the application running on the resource beforehand. Additionally, when using the Plush command-line user interface, users have the option of adding resources to their resource pool directly by using the "add resource" command from the Plush shell.

4.2 Creating a Matching

After a resource pool has been created, the Plush resource matcher is responsible for finding a valid matching—a subset of resources that satisfy the application's demands—for the application being managed by Plush. To accomplish this task, the

matcher first must parse the resource definitions for each *component* defined in the the application specification. A Plush component is merely a set of resources. Each component block defined in the application specification has a corresponding component, or set of resources, on which the processes and barriers specified in the component block are run. Component definitions also include required software, desired number of resources, optional external service usage information (discussed in detail later is this chapter), and any "static host" specifications. Static hosts are resources that *must* be used to host an application. If these resources fail or become unavailable, the entire application is automatically aborted.

Figure 4.2 shows a Plush software and component definition. Note that the XML shown is only part of the application specification. Chapter 6 illustrates several complete application specifications that include software and component definitions in addition to the block descriptions discussed in Chapter 3. The software definitions specify where to obtain the required software, the file transfer method as indicated by the "type" attribute for the package element, and the installation method as indicated by the "type" attribute of the software element. In this particular example, the file transfer method is "web" which means that a web fetching utility such as wget or curl is used to retrieve the software package. The installation method is "tar." This implies that the package has been bundled using the tar utility, and installing the package involves running tar with the appropriate arguments. "dest_path" specifies what filename the package is saved as on the resources.

The component definition begins below the software specification in Figure 4.2. Each component is given a unique name, which is used by the component blocks later to identify which set of resources should be used. Next, the "rspec" element defines "num_hosts," which is the number of resources required in the component. The "rspec" element also optionally specifies any "static_host" resources desired. Note that the use of static hosts is not recommended for most applications, since the failure of a static host results in the entire application being aborted. The "software" element within the component specification refers to the "SimpleSoftware" software package that was pre-

```
<?xml version="1.0" encoding="UTF-8"?>
<plush>
  <project name="simple">
    <software name="SimpleSoftware" type="tar">
      <package name="SimplePackage" type="web">
        <path>http://plush.ucsd.edu/software.tar</path>
        <dest_path>software.tar</dest_path>
      </package>
    </software>
    <component name="Group1">
      <rspec>
        <num_hosts>25</num_hosts>
        <static_host>ucsd_plush@planetlab1.cs.duke.edu</static_host>
      </rspec>
      <software name="SimpleSoftware" />
      <resources>
        <resource type="planetlab" group="ucsd_plush"/>
      </resources>
    </component>
  </project>
</plush>
```

Figure 4.2: Plush software and component definition.

viously defined. Lastly, the "resources" element specifies which resource group (recall that each Node object includes a group name) to use for creating the matching. In this case we are interested in PlanetLab hosts assigned to the ucsd_plush slice.

After creating the resource pool and parsing the component definition in the application specification, the resource matcher has all of the information it needs to create a matching. The matcher starts with the global resource pool, and filters out all resources that are not in the group specified in the component definition. In our example, this includes all hosts not assigned to the ucsd_plush slice. Using the remaining resources in the resource pool, the matcher randomly picks 25 (as specified by num_hosts in our example) Node objects and inserts them into the matching. The Plush controller then begins to configure these 25 resources. If a failure occurs during configuration or execution, the controller requeries the matcher. The matcher sets the "failed" flag in the Node that caused the failure[1], removes it from the matching, and inserts another randomly chosen resource from the resource pool. This process is repeated for each

[1] In addition to setting the failed flag in the Node object, the controller also notes the time at which the flag was set. In the case of long-running applications, failed flags are periodically unset after a sufficient amount of time passes.

failure throughout the duration of the application's execution. Note that resources that are marked as failed are never chosen to be part of a matching.

In shared, wide-area computing environments like PlanetLab, machines often experience high load and increased network congestion, especially during peak times when a conference deadline is approaching. Consequently, it is during these times of high load and resource contention when obtaining a usable set of resources to host an application can be difficult. In these situations the resource matcher's random choosing policy does not always allow users to achieve their desired results. To help address this problem, Plush allows users to specify a set of "preferred hosts" for running their application. Internally, each Plush Node has a numerical preference value assigned to it. In the absence of preferred hosts, all preference values are set to zero. If a resource fails, the preference value for the failed resource is reduced by some number. To increase a resource's preference value, users have the option of using the "prefer regex" command from the Plush command-line interface, which increases the preference value for any resource whose hostname matches the regular expression specified by "regex." Alternatively, the user can load an XML file that specifies the increased (or decreased) preference values for the target resources. When the matcher filters through the resource pool, it automatically chooses resources with the highest preference value first. Using this simple technique, users are able to loosely pick resources that they know to be more reliable, and thus are typically able to achieve better results.

4.3 PlanetLab Resource Selection

The preceding section discusses how Plush resources are internally maintained and organized into resource pools in general. It also describes how the Plush resource matcher uses these resource pools and the component definition section of the application specification to create matchings for the application being run. In this section, we take a closer look at how Plush interacts with a specific resource discovery service—SWORD— to select an optimal set of PlanetLab resources from the resource pool.

4.3.1 SWORD Overview

SWORD [3, 77] is a publicly available service that is designed to perform resource discovery for PlanetLab. In the previous section we alluded to the fact that finding a usable set of resources to host a PlanetLab application during times of high resource contention can be very challenging. SWORD is designed to address this challenge in an application-specific manner without requiring the user to select "preferred hosts" for running their applications. SWORD takes a query describing resource demands for a specific application as input, and returns a set of resources that satisfy these demands. Queries define groups of resources that have specific per-node (*e.g.*, load or free memory on all hosts), inter-node (*e.g.*, all-pairs latency or bandwidth within a group), and inter-group (*e.g.*, all-pairs latency or bandwidth across groups) properties. Additionally, the queries allow users to specify ranges of acceptable values for each attribute, rather than a single value. Associated with this range is a "penalty" value, which basically allows users to rank the importance of various attributes. SWORD returns a list of resources organized by group that have the lowest overall penalty. We look at a specific SWORD query later in this section.

Before describing how Plush interacts with SWORD, it is worthwhile to briefly discuss the evolution of SWORD from an architectural and managerial standpoint. SWORD has been in operation for several years now, and many valuable lessons have been learned about application management and maintenance during this time. At a high-level, the architecture of SWORD has remained largely unchanged. SWORD consists of three main components, as shown in Figure 4.3: the query, the logical database and query processor, and the optimizer and matcher. The query is an XML document that describes application-specific requirements for groups of resources. The logical database and query processor is responsible for parsing the query and maintaining the CoMon [81] measurement data needed to answer the queries[2]. The query processor filters through the measurement data, and determines which resources satisfy the per-node

[2]Originally SWORD used data gathered from ganglia [69] sensors running on all hosts, and an all-pairs-ping service [99] that measured latency between PlanetLab hosts. These services are no longer maintained on PlanetLab, and as a result SWORD currently does not support any inter-node attributes.

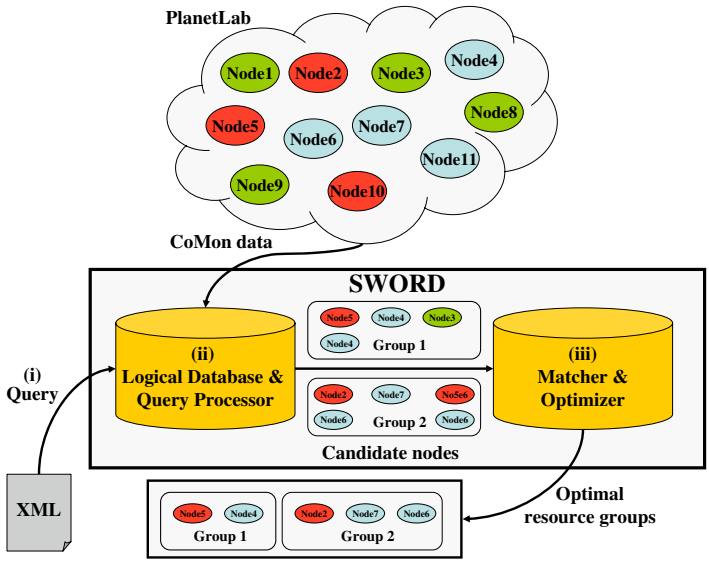

Figure 4.3: High-level overview of the SWORD architecture.

requirements in the query. The role of the optimizer and matcher is to use the measurement data and "candidate nodes" provided by the query processor to determine which groups of resources satisfy the inter-node and inter-group requirements, and also find the groups with the lowest overall penalty relative to the requirements specified in the query.

Although the high-level view of the architecture did not change over the years, the implementation and specific details of the design has undergone significant alterations. Initially, SWORD was designed to be mostly distributed. The optimizer and matcher were never distributed and always ran on a single host, but the logical database and query processor was originally implemented using a distributed hash table (DHT). One challenge with using a DHT is performing distributed range queries to gather the data across several PlanetLab hosts. We experimented with several different techniques, as shown in Figure 4.4. In the end, we found that a centralized approach with replicated servers performed the best with respect to how long it took to get a response to a query. However, even this design suffered from poor performance when PlanetLab was experiencing high load and resource contention. The problem was that although we were trying to perform resource discovery to help users find usable resources, SWORD itself

was running on overloaded PlanetLab machines. Thus SWORD suffered the same fate as other PlanetLab applications, and was essentially rendered useless during peak usage times.

The current version of SWORD is now fully centralized (similar to the Fixed architecture in Figure 4.4), and is run on a machine that is not part of the PlanetLab infrastructure. Data is still periodically gathered from CoMon and stored in the logical database (which is now an XML document), and users now submit queries directly to the SWORD server that subsequently processes the query, determines the candidate nodes, and runs the optimizer and matcher. The risk in this design is fault tolerance (a central server is a single point of failure) and potential scalability limitations; however these problems are easily addressed if they ever cause problems. SWORD presently supports a web-based user interface [100] and an XML-RPC interface for submitting queries and obtaining responses.

4.3.2 Integrating SWORD and Plush

Although setting preferred hosts as described in the previous section helps find suitable resources on PlanetLab, it is not as effective as using SWORD to find the best set of resources available for hosting an application. Hence, we decided to integrate SWORD and Plush, allowing application developers to benefit from the application management features of Plush and the advanced resource discovery features of SWORD. In order to facilitate the integration, we extended both the Plush and SWORD XML-RPC interfaces so that the two systems could communicate easily. Additionally, we modified the Plush application specification parser to recognize when an external service (such as SWORD) should be used. An example of a component definition that includes a SWORD query is shown in Figure 4.5.

One problem with using SWORD and Plush together is recovering from failures. SWORD does not store any state after responding to a query. Thus, if a resource fails, there is no simple way to requery SWORD and obtain a replacement for the failed host without retrieving a completely new list of resources. Further, SWORD does not

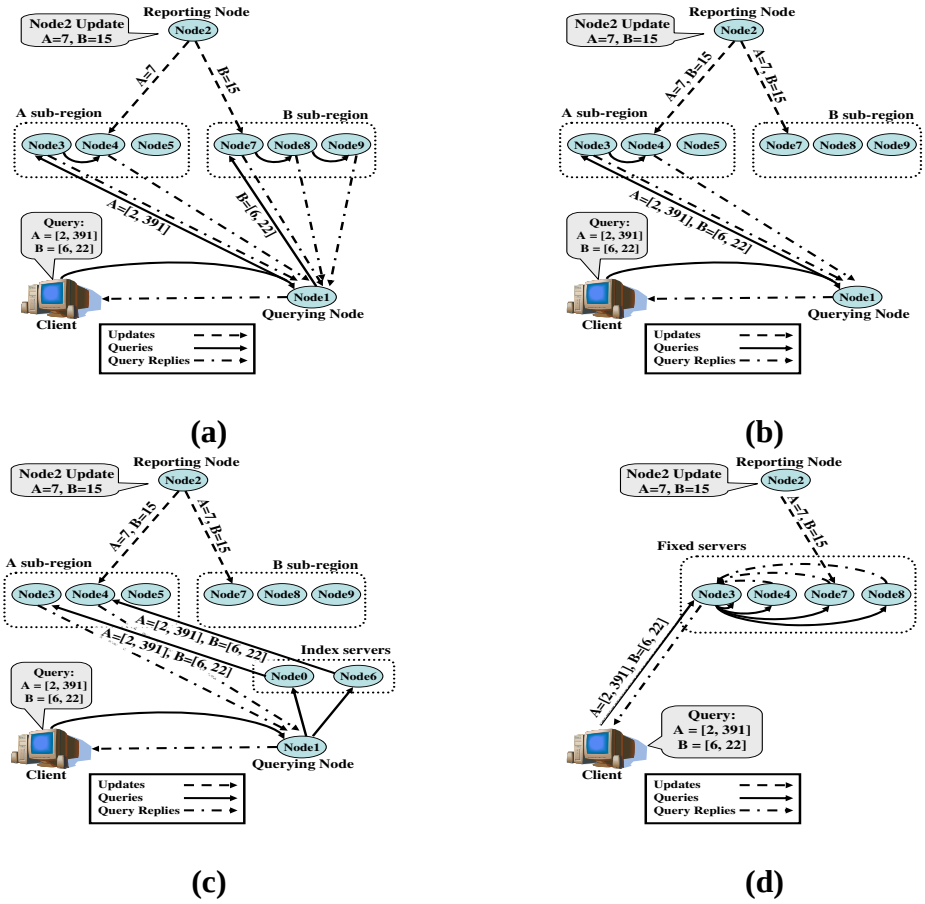

Figure 4.4: Different distributed range search query techniques used in SWORD. (a) Multiquery - small queries sent to many nodes in DHT. (b) Singlequery - large queries sent to only one node in DHT. (c) Index - index servers indicate where to obtain information in DHT. This approach limits the number of hops through the DHT for each query. (d) Fixed - centralized approach with replicated servers that hold all measurement data and respond to queries.

know or care about which hosts are assigned to which slice. Therefore it is entirely possible for SWORD to return a list of resources that are not available to the user. To address this problem, we added the functionality to retrieve the list of SWORD "candidate nodes" (see Figure 4.3) rather than the final optimized matching. Plush itself is capable of creating the matching, thus it is not necessary to use SWORD for this task. The candidate nodes that SWORD returns are guaranteed to meet the application's constraints (recall that inter-node attribute queries are no longer supported since the measurement data does not exist). Plush is able to filter the candidate nodes based

```
<?xml version="1.0" encoding="utf-8"?>
<plush>
  <project name="simple">
    <software name="SimpleSoftware" type="tar">
      <package name="SimplePackage" type="web">
        <path>http://plush.ucsd.edu/software.tar</path>
        <dest_path>software.tar</dest_path>
      </package>
    </software>
    <component name="Group1">
      <rspec>
        <num_hosts>25</num_hosts>
        <sword>
          <request>
            <group>
              <name>Group1</name>
              <num_machines>all</num_machines>
              <oneminload>0, 0, 2, 5, 1</oneminload>
              <gbfree>1, 2, 10, 50, 2</oneminload>
            </group>
          </request>
        </sword>
      </rspec>
      <software name="SimpleSoftware" />
      <resources>
        <resource type="planetlab" group="ucsd_plush"/>
      </resources>
    </component>
  </project>
</plush>
```

Figure 4.5: Plush component definition containing a SWORD query.

on the slice being used to run the application, and then increase the preference value for the Node objects. This allows Plush to easily recover from failures by choosing good replacement resources based on the SWORD candidate set, and also filter out the candidate nodes that are not part of the user's slice.

The XML that appears between the "sword" tags in Figure 4.5 is a complete and unmodified sword query. This particular query defines one group of resources, and, since we want the biggest candidate node set possible, we set the "num_machines" attributes to "all" indicating that we want to know about as many PlanetLab resources as possible. Next, the query specifies two per-node attributes: "oneminload" and "gbfree." oneminload is a measure of the average load for the past minute, and gbfree measures

how many gigabytes of free disk space are available. Any attribute measured by CoMon is supported by SWORD. Notice that there are five numbers specified for each per-node constraint. From left to right, these numbers indicates the absolute minimum, ideal minimum, ideal maximum, absolute maximum, and penalty values associated with the attributes. For example, in the gbfree constraint, we are requesting resources with at least one free GB and ideally two free GB of disk space, and ideally a max of ten free GB and no more than fifty free GB of disk space[3]. For resources with values that fall outside of the ideal range (which is from two to ten GB) but inside the absolute range (which is from one to fifty GB), a normalized penalty of two will be assigned. This means that resources with values close to two and ten GB will have penalties close to zero, and resources with values close to one and fifty GB will have penalties close to two. Resources with values outside of the absolute range are assigned an infinite penalty. Since they do not satisfy the specified per-node constraints, they are not included in the candidate node set that SWORD returns.

When the Plush controller parses the application specification and discovers the "sword" portion of the XML, it immediately sends the query to the SWORD server via XML-RPC. SWORD responds with a list of PlanetLab machines that satisfy the constraints specified in the query. The Plush resource matcher uses this information to increase the preference values for the corresponding Node objects. Additionally, Plush uses the SWORD penalty values to set the preference values according to how well the resources meet the application's demands. Hence, resources with low SWORD penalty values are given high Plush preference values, and resources with high SWORD penalty values are given lower Plush preference values. After setting the preference values, the matcher than proceeds as usual, choosing resources with higher preference values before resources with lower preference values. Plush users also have the option of rerunning the SWORD query periodically to maintain a "fresh" list of good resources.

[3] In reality, we would not likely specify an upper limit for gbfree. We are specifying an ideal and absolute max in this example for illustrative purposes only.

4.4 Virtual Machine Support

In addition to using SWORD for resource selection on PlanetLab, Plush also supports using virtual machine management systems for creating and obtaining resources. In particular, Plush provides an interface for using both Shirako [49] and Usher [71]. Shirako [49] is a utility computing framework. Through programmatic interfaces, Shirako allows users to create dynamic on-demand clusters of resources, including storage, network paths, physical servers, and virtual machines. Shirako is based on a resource leasing abstraction, enabling users to negotiate access to resources. Usher [71] is a virtual machine scheduling system for cluster environments. It allows users to create their own virtual machines or clusters. When a user requests a virtual machine, Usher uses data collected by virtual machine monitors to make informed decisions about when and where the virtual machine should run.

Through its XML-RPC interface, Plush interacts with both the Shirako and Usher servers in a similar manner as SWORD. Unlike SWORD, however, in Shirako and Usher the resources do not exist in advance. The resources must be created and added to the resource pool before the Plush resource matcher can create a matching. To support this dynamic resource creation and management, we again augment the Plush application specification with a description of the desired virtual machines, and then send this description to the corresponding service for resource creation. Just as we included a SWORD query within the component definition of a Plush application specification, we similarly include information about the desired attributes of the virtual machine resources so that this information can be passed on to either Shirako or Usher. As the Plush controller parses the application specification, it stores the resource description, and when the “plushCreateResources” command is issued (either via the Plush command-line interface or programmatically through XML-RPC), Plush contacts the appropriate Shirako or Usher server and submits the resource request. Once the resources are ready for use, Plush is informed via an XML-RPC callback that also contains contact information about the new resources. This callback updates the Plush resource

pool and the user is free to start applications on the new resources. If the resources must be removed at any point, the Shirako or Usher server can contact Plush again via XML-RPC, and the resources will be removed from the resource pool.

4.4.1 Using Shirako

Similar to the how we included the SWORD query in the application specification in the preceding section, if a Plush user wants to obtain Shirako resources for hosting an application, the application specification must again be augmented with a description of the desired resources. The syntax is similar to that of the SWORD query, except that in the case of Shirako, the attributes define the resources that will be created. Figure 4.6 shows an example of a Plush component definition that is augmented with a Shirako resource request. Shirako currently creates Xen [10] virtual machines (as indicated by the "type" flag with value "1" in the resource description) with the CPU speed, memory, disk space, and maximum bandwidth specified in the resource request. If one or more of these attributes is not explicitly defined, Shirako uses default values for creating virtual machines. Also, Shirako arbitrates access to resources using leases. Notice that the resource description contains a lease parameter which tells Shirako how long the user intends to use the resources. Lastly, the resource description specifies which Shirako server to contact with the resource request.

Since Shirako is a lease-based resource management environment, it is possible that the resources will not be available immediately when Plush contacts the Shirako server on behalf of the user. Thus, rather than having Plush block on the XML-RPC call until the resources are available, Plush instead registers a callback with the Shirako server. When the resources become available, the Shirako server contacts the Plush controller with information regarding the newly created resources. This information includes the hostname, group name, username, and Plush client port number. Shirako assumes that the Plush user has registered their SSH key with the Shirako server ahead of time, and when Shirako creates the virtual machines, it installs SSH keys directly onto the resources. This makes accessing the resources via passphrase-less SSH pos-

```
<?xml version="1.0" encoding="utf-8"? >
<plush>
  <project name="simple">
    <component name="Group1">
      <rspec>
        <num_hosts>10</num_hosts>
        <shirako>
            <num_hosts>10</num_hosts>
            <type>1</type>
            <memory>200</memory>
            <bandwidth>200</bandwidth>
            <cpu>50</cpu>
            <lease_length>600</lease_length>
            <server>http://shirako.cs.duke.edu:20000</serve r>
        </shirako>
      </rspec>
      <resources>
        <resource type="ssh" group="shirako "/>
      </resources>
    </component>
  </project>
</plush>
```

Figure 4.6: Plush component definition containing Shirako resources.

sible after the resources are created. When all requested resources are available, Plush sends a message to the user indicating that the resources are ready for use. After the requested lease length expires, Shirako contacts the Plush controller and asks that the resources be removed from the resource pool.

Plush is currently being used by Shirako users regularly at Duke University. While Shirako multiplexes resources on behalf of users, it does not provide any abstractions or functionality for using the resources once they have been created. On the other hand, Plush provides abstractions for managing applications on distributed resources, but provides no support for multiplexing resources. A “resource” is merely an abstraction in Plush to describe a machine that can host a distributed application. Resources can be added and removed from the application’s resource pool, but Plush relies on external mechanisms (like Shirako) for the creation and destruction of resources.

The integration of Shirako and Plush allow users to seamlessly leverage the functionality of both systems. Although Shirako does provide a web based interface

for creating and destroying resources, it does not provide an interface for using the new resources, so Shirako users benefit from the interactivity provided by the Plush command-line interface. Researchers at Duke are currently using Plush to orchestrate workflows of batch tasks and perform data staging for scientific applications including BLAST [6] on virtual machine clusters managed by Shirako [45].

4.4.2 Using Usher

The integration of Plush and Usher is very similar to the integration of Plush and Shirako. Like Shirako, Usher takes a request that describes the characteristics of the desired resources as input and creates clusters of virtual machines that satisfy the specified constraints. Figure 4.7 shows an example of a Plush component definition that includes an Usher resource request. Also like Shirako, Usher creates Xen virtual machines, and the attributes in the resource request correspond to attributes used by Xen to create customized resources. Specifically, Usher uses four attributes. The attribute "count" simply defines how many machines are needed, and "ram" specifies the amount of required memory. Note that the number of machines needed (count) should be greater than or equal to the number of resources in the component (num_hosts). The "cluster" attribute specifies an optional name for the virtual cluster. Usher supports one additional optional parameter, the "kernel_uri" attribute (not shown in Figure 4.7), that defines the location of the desired kernel image.

Unlike Shirako, Usher is not a lease based system. When a user requests a virtual machine, the machine is created immediately. Thus, rather than requiring a callback that adds new resources to the resource pool after some potential delay as in Shirako, Usher returns the information for accessing the newly created machines in response to the initial "createResource" call. This response includes the new virtual machine hostnames, usernames, and group identifiers. Similar to Shirako, Usher requires that the user register their SSH key with the Usher server before requesting resources so that the key can be automatically installed on the virtual machines, making SSH passphrase-less authentication possible.

```
<?xml version="1.0" encoding="utf-8"? >
<plush>
  <project name="simple">
    <component name="Group1">
      <rspec>
        <num_hosts>10</num_hosts>
        <usher>
            <count>10</count>
            <ram>256</ram>
            <cluster>plush</cluster>
        </usher>
      </rspec>
      <resources>
        <resource type="ssh" group="usher"/>
      </resources>
    </component>
  </project>
</plush>
```

Figure 4.7: Plush component definition containing Usher resources.

The integration of Plush and Usher is still in a preliminary stage, and has not yet seen extensive use. Usher has been in use for only a few months, and is still under development. Also, Usher itself provides a terminal interface that is similar to the Plush command-line interface, and hence Usher users have been less enthusiastic about using Plush than Shirako users who benefited from the functionality of the Plush command-line. Since Usher users already have a user-friendly interface for controlling resources, there is less motivation for them to try Plush. However, we are hopeful that as the popularity of Usher increases, we will be able to convince new Usher users who are managing complex applications to use Plush.

4.5 ModelNet Emulated Resources

Aside from PlanetLab resources and virtual machines, Plush also supports running applications on resources in emulated environments. In this section we discuss how Plush supports adding emulated resources from ModelNet [103] to the resource pool. Further, we describe how the Plush XML-RPC programmatic interface is used to perform job execution in a batch scheduler that arbitrates access to ModelNet resources.

Mission is a simple batch scheduler used to manage the execution of jobs that run on ModelNet in our research cluster. ModelNet is a network emulation environment that consists of one or more Linux edge nodes and a set of FreeBSD core machines running a specialized ModelNet kernel. The code running on the edge hosts routes packets through the core machines, where the packets are subjected to the delay, bandwidth, and loss specified in a target topology. A single physical machine hosts multiple "virtual" IP addresses that act as emulated resources on the Linux edge hosts. To setup the ModelNet computing environment with the target topology, two phases of execution are required: deploy and run. Before running any applications, the user must first *deploy* the desired topology on each physical machine, including the FreeBSD core. ModelNet topologies are defined in two files: the model file and the route file. The model file specifies the assignment of emulated hosts to physical machines, as well as the subnet and IP addresses for the emulated hosts. The route file defines the properties of the network links that connect the emulated hosts. The deploy process essentially instantiates the emulated hosts, and installs the topology on all machines. Then, after setting a few environment variables, the user is free to *run* applications on the emulated hosts using virtual IP addresses just as applications are run on physical machines using real IP addresses.

A single ModelNet experiment typically consumes almost all of the computing resources available on the physical machines involved. Thus, when running an experiment, it is essential to restrict access to the machines so that only one experiment is running at a time. Further, there are a limited number of FreeBSD core machines running the ModelNet kernel available, and access to these hosts must also be arbitrated. Mission, a simple batch scheduler, was developed locally to help accomplish this goal. ModelNet users submit their jobs to the Mission queue, and as the machines become available, Mission pulls jobs off the queue and runs them on behalf of the user. This ensures that no two jobs are run simultaneously, and also allows the resources to be shared more efficiently. Rather than partitioning the resources into smaller groups so that multiple users can reserve machines for exclusive access, all users can share all of

```
<?xml version="1.0" encoding="UTF-8"?>
<plush>
 <resource_manager type="ssh">
  <node hostname="sys80.ucsd.edu:1540" group="phys" flag="core"/>
  <node hostname="sys81.ucsd.edu:1540" group="phys"/>
  <node hostname="sys81.ucsd.edu:1541" vip="10.0.0.1" vn="1" group="emul"/>
  <node hostname="sys81.ucsd.edu:1542" vip="10.0.0.2" vn="2" group="emul"/>
  <node hostname="sys81.ucsd.edu:1543" vip="10.0.0.3" vn="3" group="emul"/>
  <node hostname="sys81.ucsd.edu:1544" vip="10.0.0.4" vn="4" group="emul"/>
 </resource_manager>
</plush>
```

Figure 4.8: Plush directory file for a ModelNet topology. sys80 is the FreeBSD core machine. sys81 is a Linux edge host that is running four emulated virtual hosts.

the resources, allowing for an increased number of total hosts in the emulated topology, and also maximizing the overall utility of the cluster.

4.5.1 Configuring ModelNet with Plush

A Mission job submission has two components: a Plush application specification and directory file. For ModelNet, the directory file contains information about both the physical and virtual (emulated) resources on which the ModelNet experiment will run. Typically the directory file is generated directly from a ModelNet model file. Unlike Plush directory files for other environments, the ModelNet directory file entries contain extra parameters that specify the mapping from physical hosts to virtual IP addresses. Figure 4.8 shows an example directory file for a ModelNet topology. In this figure, some of the resources include two extra parameters, “vip” and “vn”, which define the virtual IP address and virtual number (similar to a hostname) for the emulated resources. Also, notice that different group names are used to distinguish emulated hosts from physical hosts. The Plush controller parses this file at startup and populates the resource pool with both the emulated and physical resources. The matcher then uses the group information to ensure that the correct resources are used in each stage of the execution.

In addition to the directory file that is used to populate the Plush resource pool, users also submit an application specification describing the application they wish to run

on the emulated topology to the Mission server. This application specification contains two component blocks. The first component block describes the processes that run on the physical machines during the deployment phase (where the emulated topology is instantiated). The corresponding component that is associated with this component block specifies that the resources used during this phase belong to the "phys" group. The second component block defines the processes associated with the target application. The component for this component block specifies that resources belong to the "emul" group. When the controller starts the Plush clients on the emulated hosts, it specifies extra command-line arguments that are defined in the directory file by the "vip" and "vn" attributes. These arguments set the appropriate ModelNet environment variables, ensuring that all commands run on that client on behalf of the user inherit those settings. An example application specification that uses the resources defined in Figure 4.8 is shown in Figure 4.9.

When a user submits a Plush application specification and directory file to Mission, the Mission server parses the directory file to identify which resources are needed to host the application. When those resources become available for use, Mission starts a Plush controller on behalf of the user using the Plush XML-RPC interface. Mission passes Plush the directory file and application specification, and continues to interact with Plush throughout the execution of the application via XML-RPC. After Plush notifies Mission that the execution has ended, Mission kills the Plush process and reports back to the user with the results. Any terminal output that is generated is emailed to the user.

Plush jobs are currently being submitted to Mission on a daily basis at UCSD. These jobs include experimental content distribution protocols, distributed model checking systems, and other distributed applications of varying complexity. Mission users benefit from Plush's automated execution capabilities. Users simply submit their jobs to Mission and receive an email when their task is complete. They do not have to spend time configuring their environment or starting the execution. Individual host errors that occur during execution are aggregated into one message and returned to the user in an

```
<?xml version="1.0" encoding="utf-8"?>
<plush>
  <project name="deploy">
    <component name="PhysicalHosts">
      <rspec>
        <num_hosts>2</num_hosts>
      </rspec>
      <resources>
        <resource type="ssh" group="phys" />
      </resources>
    </component>
    <component name="VirtualHosts">
      <rspec>
        <num_hosts>4</num_hosts>
      </rspec>
      <resources>
        <resource type="ssh" group="emul" />
      </resources>
    </component>
    <application_block name="modelnet_app">
      <execution>
        <component_block name="phys_block">
          <component name="PhysicalHosts" />
          <process_block name="phys">
            <process name="deploy">
              <path>/projects/modelnet/bin/deployhost</path>
              <cmdline>
                <arg>example.model</arg>
                <arg>example.route</arg>
              </cmdline>
            </process>
          </process_block>
        </component_block>
        <component_block name="virt_block">
          <component name="VirtualHosts" />
          <predecessor name="phys_block"/>
          <process_block name="modelnet_virtual">
            <process name="test_hostname">
              <path>/bin/hostname</path>
            </process>
          </process_block>
        </component_block>
      </execution>
    </application_block>
  </project>

</plush>
```

Figure 4.9: ModelNet application specification. Each emulated resource runs the command "/bin/hostname."

email. Logfiles are collected in a public directory on a common file system and labeled with a job ID, so that users are free to inspect the output from individual hosts if desired. Another key benefit of using Mission is that it allows users to more easily transition from an emulation environment to live deployment. Once a Mission user has created an application specification for execution on ModelNet, the changes required to adapt the specification for execution on PlanetLab are trivial, and mostly involve removing

the component block responsible for deploying the emulated topology. In this context, Plush accomplishes its goal of helping users seamlessly transition from emulation to live deployment during application development.

4.6 Summary

In this chapter, we discussed how Plush interacts with several external resource management services—namely SWORD, Shirako, Usher, and Mission—to add and remove different types of resources to and from an application's resource pool. We also described how the Plush resource matcher uses these resources to create matchings that contain the best set of resources available for hosting the target application. Using the extensible resource abstractions provided by Plush for interacting with resources from a variety of different environments, users are able to run their applications on PlanetLab, ModelNet, or on clusters of Xen virtual machines without ever having to worry about the underlying details of the environment.

4.7 Acknowledgments

Chapter 4, in part, has been submitted for publication of the material as it appears in the Large Installation System Administration Conference, 2007, Albrecht, Jeannie; Braud, Ryan; Dao, Darren; Topilski, Nikolay; Tuttle, Christopher; Snoeren, Alex C.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it will appear in the ACM Transactions on Internet Technology, May 2008, Albrecht, Jeannie; Oppenheimer, David; Patterson, David; Vahdat, Amin. The dissertation author was one of the primary investigators and authors of this paper.

Chapter 5

Partial Barriers

This chapter discusses techniques for accomplishing wide-area distributed synchronization in Plush. Traditionally, synchronization barriers have been used to ensure that no cooperating process advances beyond a specified point until all processes have reached that point. In heterogeneous large-scale distributed computing environments, with unreliable network links and machines that may become overloaded and unresponsive, traditional barrier semantics are too strict to be effective for a broad range of distributed applications. In response to this limitation, we explore several relaxations and introduce a *partial barrier*, which is a synchronization primitive designed to enhance liveness in failure-prone computing environments. Partial barriers are robust to variable network conditions; rather than attempting to hide the asynchrony inherent to wide-area settings, they enable appropriate application-level responses. In this chapter, we describe how partial barriers have been integrated into Plush, and in Chapter 6 we evaluate the improved performance achieved using partial barriers for several wide-area distributed applications.

5.1 Background and Overview

One of the main goals of Plush is to support the deployment of a broad range of applications in a variety of distributed environments, including large-scale wide-area computing platforms, where significant variations in processor performance, network

connectivity, and node reliability are the norm. These computing environments lie in stark contrast to the tightly-coupled cluster and supercomputer environments traditionally employed by compute-intensive applications. What remains unchanged, however, is the need to synchronize various phases of computation across the participating resources. The realities of these failure-prone environments place additional demands on the synchronization mechanisms required in Plush; while existing techniques provide correct operation in volatile environments, the application's performance is often severely degraded due to failures or overloaded resources. In this chapter, we describe how new, relaxed synchronization semantics in Plush can provide significant performance improvements in distributed applications run across the wide-area.

Synchronizing parallel and distributed computations in general has long been the focus of significant research effort. At a high-level, the goal of synchronization in this context is to ensure that concurrent computation tasks—across independent threads, processors, nodes in a cluster, or resources spread across the Internet—are able to make independent forward progress while still maintaining some higher-level correctness semantic. Perhaps the simplest synchronization primitive is the *barrier* [50], which establishes a rendezvous point in the computation that all concurrent nodes, processors, or threads must reach before any are allowed to continue. Bulk synchronous parallel programs running on massively parallel processors (MPPs) or tightly-coupled clusters employ barriers to perform computation and communication in phases, transitioning from one consistent view of shared underlying data structures to another. Thus during the design of Plush it only seemed natural to support the barrier synchronization primitive to accomplish similar synchronization goals.

In past work, barriers and other synchronization primitives have defined strict semantics that ensure safety—*i.e.*, that no node, thread, or processor falls out of lockstep with the others—at the expense of liveness. In particular, if employed naively, a parallel or distributed computation using barrier synchronization moves forward at the pace of the slowest participant and the entire computation must be aborted if any participant fails. In closely-coupled supercomputer or cluster environments, these problems

can be avoided relatively easily. Failure during computation is expected to be rare, and skillful programmers optimize the performance of their applications by leveraging knowledge about the relative speed of individual processors and nodes in their target environment. Further, dataflow can be carefully crafted based upon an understanding of transfer times and access latencies to prevent competing demand for the I/O bus.

In wide-area computing environments—where individual node speeds are unknown and variable, communication topologies are unpredictable, and failure is commonplace—applications must be robust to a range of operating conditions. It is nearly impossible to predict the performance of individual resources, and thus the performance tuning common in cluster and supercomputing environments is impractical. Further, individual node failures are almost inevitable, hence applications that run in these volatile environments are generally engineered to adapt to or recover from a variety of failures. Additionally, due to the unpredictable nature of these environments, it is often difficult to determine the optimal degree of concurrency *a priori*. As a result, one goal in the design of Plush is to provide robust and adaptive mechanisms for adjusting the degree of concurrency of an application during its execution, which is especially helpful in cases where parallel execution appears to be degrading performance due to self-interference.

In addition to providing mechanisms for adaptively determining the optimal concurrency level in distributed applications, Plush partial barriers address two other limitations of traditional barriers. Using traditional semantics, a resource arriving at a barrier blocks and waits for all other resources to arrive before continuing computation. Using partial barriers, a resource need not necessarily block waiting for all other resources to arrive—doing so would likely sacrifice efficiency or even liveness as the application waits for either slow or failed resources. Similarly, releasing a barrier does not necessarily imply that all resources should pass through the barrier simultaneously—*e.g.*, simultaneously releasing thousands of resources to download a software package effectively mounts a denial-of-service attack against the target repository. Instead, partial barriers allow distributed applications to manage the entry and release semantics of

their logical barriers in an application-specific manner, as described in the Plush application specification.

In summary, this chapter discusses the design and implementation of a new synchronization primitive in Plush that focuses on improving performance in distributed applications running in wide-area computing environments. In this context, we make the following contributions:

- We introduce a *partial barrier*, a synchronization primitive designed for heterogeneous failure-prone environments. By relaxing traditional semantics, partial barriers enhance liveness in the face of slow, failed, and self-interfering resources.
- Based on the observation that the arrival rate at a barrier will often form a heavy-tailed distribution, we design a heuristic to dynamically detect the *knee of the curve*—the point at which arrivals slow considerably—allowing applications to continue despite slow nodes. We also adapt the rate of release from a barrier to prevent performance degradation due to self-interfering processes.
- We integrate partial barriers into the core design of Plush, so that Plush and the applications being managed by Plush can achieve the benefits that partial barriers provide. In Chapter 6 we discuss how we added partial barriers to several existing distributed applications that are managed by Plush, resulting in significant performance improvements.

5.2 Motivation

In the remainder of this chapter, we refer to nodes or resources as *entering* a barrier when they reach a point in the computation that requires synchronization. When a barrier *releases* or *fires* (we use the terms inter-changeably), the blocked nodes are allowed to proceed on to the next phase of computation. According to strict barrier semantics, ensuring safety, *i.e.*, global synchronization, requires all nodes to reach a synchronization point before any node proceeds. In the face of wide variability in performance and prominent failures, however, strict enforcement may force the majority of

nodes to block while waiting for a handful of slow or failed nodes to catch up. Many wide-area applications already have the ability to reconfigure themselves to tolerate node failures. We can harness this functionality in Plush to avoid excessive waits at barriers: once slow nodes are identified, Plush can remove them from the computation, and possibly replace them with quicker nodes for the remainder of the execution.

One of the important questions, then, is determining when to release a barrier, even if all nodes have not arrived at the synchronization point. That is, it is important to dynamically determine the point where waiting for additional nodes to enter the barrier will cost the application more than the benefit brought by any additional arriving nodes in the future. This determination often depends on the semantics of individual applications. Even with full knowledge of application behavior, making the decision appropriately requires future knowledge. During the design of Plush we developed a technique to dynamically identify "knees" in the node arrival process, *i.e.*, points where the arrival process significantly slows. Plush uses these points to make informed application-specific decisions regarding when to release barriers.

A primary contribution of this work is the definition of relaxed barrier semantics in Plush that provide support for a variety of wide-area distributed computations. Although we will look at specific applications that use partial barriers in Chapter 6, to motivate our proposed extensions consider the following general application scenarios:

Application initialization. Many distributed applications require a startup phase to initialize their local state and to coordinate with remote participants before beginning their operation. Consider a distributed hash table that must initialize routing tables at a fraction of the participants before performing any lookups, or an overlay multicast tree that must have most participants join before beginning to transmit data. Typically, developers introduce an artificial delay judged to be sufficient to allow the system to stabilize. Alternatively, when using Plush to manage these applications, we can define partial barriers that cleanly separate the initialization phase from the remainder of the execution in our application specification. For example, if each node entered a barrier and informed the Plush controller upon completing the initialization phase, the

Plush controller would know exactly when initialization completes across all participants, and developers would be freed from introducing arbitrarily chosen delays into the interactive development/debugging cycle.

Phased computation and communication. Scientific applications and large-scale computations often operate in phases consisting of one or many sequential, local computations, followed by periods of global communication or parallel computation. For instance, an application might consist of a phase of local computation on a data source followed by a phase of global communication to distribute the necessary updates for the next phase of computation. These computations naturally map to the partial barrier abstraction in Plush: one phase of the computation defined in the application specification must complete before proceeding to the next phase, and each phase is separated by a barrier. Other applications operate on a work queue that distributes tasks to available machines based on the rate that they complete individual tasks. Here, a barrier may serve as a natural point for distributing or reallocating work.

Coordinated network operations. Many distributed applications measure network characteristics. However, uncoordinated measurements can self-interfere, leading to wasted effort and incorrect results. Such systems benefit from a mechanism that limits the number of nodes that simultaneously perform probes. Similarly, imagine an application that requires thousands of nodes to simultaneously download a file from the same web server. Serving thousands of simultaneous requests at once will cause the server to thrash and become overloaded. Again we want to limit the number of simultaneous downloads based on the speed and network capacity of the server. Partial barriers are capable of providing this needed functionality in both cases. In these applications, the Plush application specification defines two barriers to delimit a "critical section" of activity (*e.g.*, a network measurement or file transfer). The first barrier releases some maximum number of nodes into the critical section at a time and waits until these nodes reach the second barrier, thereby exiting the critical section, before releasing the next round of nodes. In this context, partial barriers provide the functionality of a counting semaphore that limits the number of simultaneous network operations.

To further clarify the goal of partial barriers in Plush, it is perhaps worthwhile to consider what we *are not* trying to accomplish. Partial barriers in Plush provide only a loose form of group membership [28, 42, 74]. In particular, partial barriers do not provide any ordering of messages relative to barriers as in virtual synchrony [14, 16], nor do partial barriers require that all participants come to consensus regarding a view change [61]. In effect, we strive to construct an abstraction that exposes the asynchrony and the failures prevalent in wide-area computing environments in a manner that allows Plush to make dynamic and adaptive application-specific decisions as to how to respond.

It is also important to realize that not all applications will benefit from relaxed synchronization semantics. The correctness of certain classes of applications cannot be guaranteed without strict synchronization. For example, some applications may require a specific number of hosts to complete a computation, and thus releasing a barrier early without waiting for all participants to arrive will yield incorrect results. Other applications may approximate a measurement (such as network delay) and continuing without all nodes reduces the accuracy of the result. However, many distributed applications, including several shown in Chapter 6, can afford to sacrifice global synchronization without negatively impacting the results. These applications either support dynamically degrading the computation, or are robust to failures and can tolerate mid-computation reconfigurations. Our results indicate that for applications that are willing and able to sacrifice safety, the semantics provided by partial barriers in Plush have the potential to improve performance significantly, especially in volatile wide-area environments.

5.3 Design and Implementation

Partial barriers are a set of semantic extensions to the traditional barrier synchronization abstraction. Our implementation has a simple interface for customizing barrier functionality in Plush. This section describes these extended semantics, details our API, and presents the implementation details.

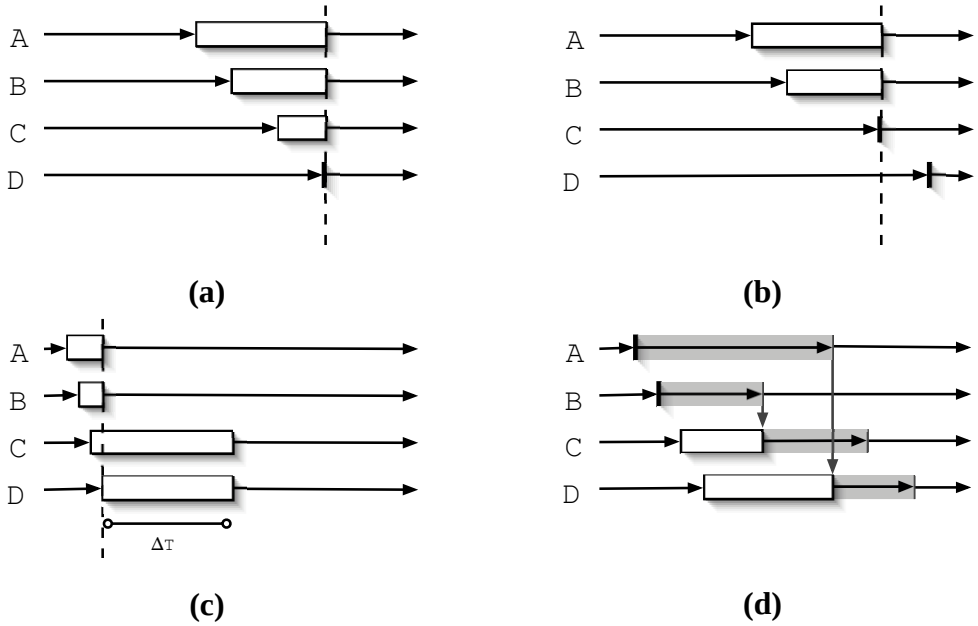

Figure 5.1: (a) Traditional semantics: All hosts enter the barrier (indicated by the white boxes) and are simultaneously released (indicated by dotted line). (b) Early entry: The barrier fires after 75% of the hosts arrive. (c) Throttled release: Hosts are released in pairs every ΔT seconds. (d) Counting semaphore: No more than two hosts are simultaneously allowed into a "critical section" (indicated by the grey regions). When one host exits the critical section, another host is allowed to enter.

5.3.1 Design

We define two new barrier semantics to provide better support for applications that require synchronization in failure-prone wide-area computing environments. The new semantics are described below.

Early entry – Traditional barriers require all nodes to enter a barrier before any node may pass through, as in Figure 5.1(a). A partial barrier with early entry (also called early release) is instantiated with a timeout, a minimum percentage of entering nodes, or both. Once the barrier has met either of the specified preconditions, nodes that have already entered the barrier are allowed to pass through without waiting for the remaining slow nodes to arrive (Figure 5.1(b)). Alternatively, an application may instead choose to receive callbacks from the Plush controller as nodes enter and manually release the barrier, enabling the evaluation of arbitrary predicates.

Throttled release – Typically, a barrier releases all nodes simultaneously when a barrier's precondition is met. A partial barrier with throttled release specifies a rate of release, such as two nodes every ΔT seconds as shown in Figure 5.1(c). A special variation of throttled release barriers allows applications to limit the number of nodes that simultaneously exist in a "critical section" of activity, creating an instance of a counting semaphore [30] (shown in Figure 5.1(d)), which may be used, for example, to throttle the number of nodes that simultaneously perform network measurements or software downloads. A critical distinction between traditional counting semaphores and partial barriers, however, is support for failures. For instance, if a sufficient number of slow or soon-to-fail nodes pass a counting semaphore, they will limit access to other participants, possibly forever. Thus, as with early entry barriers, throttled release barriers eventually time out slow or failed nodes, allowing the system as a whole to make forward progress despite individual failures.

One issue that our proposed semantics introduce that does not arise with strict barrier semantics is handling nodes performing late entry, *i.e.*, arriving at an already released barrier. Plush supports two options to address this case: i) pass-through semantics that allow the node to proceed with the next phase of the computation even though it arrived late; ii) catch-up semantics that issue an exception allowing Plush to reintegrate the node into the mainline computation in an application-specific manner. This may involve skipping ahead to the next barrier (subsequently omitting the intervening computation) in an effort to "catch up" to the other nodes. Alternatively, Plush may decide to completely remove the late arriving node from the remainder of the computation, or ask the resource matcher for a replacement.

5.3.2 Partial Barrier API

Figure 5.2 summarizes the Plush partial barrier API. Note that application developers who wish to take advantage of partial barriers in Plush need not code to the APIs shown in this section (although we do support this functionality). Rather, the API is shown here for clarity, and to help explain our implementation. In most

```
class PartialBarrier {
  PartialBarrier(string name, int max, int timeout,  int percent, int minWait);
  static void setManager(string Hostname);
  void enter(string label, string Hostname);
  void setEnterCallback(bool (*callbackFunc)(string label,  string Hostname,
        bool default), int timeout);
  map<string label, string Hostname> getHosts(void);
}
```

Figure 5.2: Partial barrier instantiation API.

applications, partial barriers are defined within the Plush application specification by simply specifying a few extra attributes in a typical barrier block.

When an application uses partial barriers, each Plush client becomes a barrier participant. Each barrier participant involved in the application initializes a local barrier with a constructor that takes the following arguments: `name`, `max`, `timeout`, `percent`, and `minWait`. `name` is a globally unique identifier for the barrier. `max` specifies the maximum number of participants expected to arrive at the barrier. (While we do not require *a priori* knowledge of participant identity, it would be straightforward to add.) The `timeout` in milliseconds sets the maximum time that can pass from the point where the first node enters a barrier before the barrier is released. The `percent` field similarly captures a minimum percentage of the maximum number of nodes that must reach the barrier to activate early release. The `minWait` field is associated with the `percent` field and specifies a minimum amount of time to wait (even if the specified percentage of nodes have reached) before releasing the barrier with less than the maximum number of nodes. Without this field, the barrier will deterministically be released upon reaching the threshold percent of entering nodes even when all nodes are entering rapidly. However, the barrier is always released if `max` nodes arrive, regardless of `minWait`. The `timeout` field overrides the `percent` field; the barrier will fire if the timeout is reached, regardless of the percentage of nodes entering the barrier. Thus `timeout` must be greater than `minWait`—otherwise `minWait` is never used. The last three parameters to the constructor are optional; if left unspecified the barrier operates as a traditional synchronization barrier, ignoring partial barrier properties.

Coordination of barrier participants is controlled by a barrier manager, who is typically (and by default) defined to be the Plush controller. The identity of the barrier manager can be overridden in the application specification, and is set internally using the `setManager()` method. Plush clients call their local barrier's `enter()` method and pass in their `Hostname` and `label` when they reach the appropriate point in their execution. The `label` argument supports advanced functionality such as load balancing, which we will discuss in detail in Chapter 6. The clients' `enter()` method notifies the barrier manager that the particular node has reached the synchronization point. Our implementation supports blocking calls to `enter()` (as described here) or optionally a callback-based mechanism where the entering node is free to perform other functionality until the appropriate release callback is received, allowing the node to advance to the next phase of computation.

While the standard Plush partial barrier API provides simplistic support for the early release of a barrier, an application may maintain its own state to determine when a particular barrier should fire and to manage any side effects associated with barrier entry or release. For instance, an application may want the barrier manager to kill all processes running on a node that arrives late to a particular (already released) barrier. To support application-specific functionality, the `setEnterCallback()` method specifies a function to be called when any node enters a barrier. (In order to take advantage of this advanced functionality, applications must code to the API directly, rather than defining partial barriers within the application specification.) The callback takes the `label` and `Hostname` passed to the `enter()` method and a boolean variable that specifies whether the barrier manager would normally release the barrier upon this entry. The callback function returns a boolean value to specify whether the barrier should actually be released or not, potentially overriding the manager's decision. A second argument to `setEnterCallback()` called `timeout` specifies the maximum amount of time that may pass before successive invocations of the callback, preventing situations where the application waits an indefinite amount of time for the next node to arrive before deciding to release.

```
class ThrottleBarrier extends PartialBarrier {
  void setThrottleReleasePercent(int percent);
  void setThrottleReleaseCount(int count);
  void setThrottleReleaseTimeout(int timeout);
}

class SemaphoreBarrier extends PartialBarrier {
  void setSemaphoreCount(int count);
  void setSemaphoreTimeout(int timeout);
  void release(string label, string Hostname);
  void setReleaseCallback (int (*callbackFunc)(string label, string Hostname,
       int default), int timeout);
}
```

Figure 5.3: ThrottleBarrier and SemaphoreBarrier API.

Barrier participants may wish to learn the identity of all hosts that have passed through a barrier, similar to (but with relaxed semantics from) view advancement or GBCAST in virtual synchrony [15]. The `getHosts()` method returns a map of `Hostnames` and `labels` through a remote procedure call with the barrier manager. If many hosts are interested in membership information, it can optionally be propagated from the barrier manager to all nodes by default as part of the barrier release operation.

Figure 5.3 describes a subclass of `Barrier`, called `ThrottleBarrier`, with throttled release semantics. These semantics allow for a pre-determined subset of the maximum number of nodes to be released from the barrier at a specified rate. The methods `setThrottleReleasePercent()` and `setThrottleReleaseCount()` periodically release a percentage and number of nodes, respectively, once the barrier fires. `setThrottleReleaseTimeout()` specifies the periodicity of release.

In addition, Figure 5.3 details a variant of throttled release barriers, `SemaphoreBarrier`, which specifies a maximum number of nodes that may simultaneously enter a critical section. A `SemaphoreBarrier` extends the throttled release semantics further by placing a barrier at the beginning and end of a critical section of activity to ensure that only a specific number of nodes pass into the critical section simultaneously. One key difference for this type of barrier is that it does not require any

minimum number of nodes to enter the barrier before beginning to release nodes into the subsequent critical section. It simply mandates a maximum number of nodes that may simultaneously enter the critical section. The `setSemaphoreCount()` method sets this maximum number allowed in the critical section. Nodes call the barrier's `release()` method upon completing the activity in the subsequent critical section, allowing the barrier to release additional nodes. `setSemaphoreTimeout()` allows for timing out nodes that enter the critical section but do not complete within a maximum amount of time. In this case, they are assumed to have failed, enabling the release of additional nodes. The `setReleaseCallback()` enables application-specific release policies and timeout of slow or failed nodes in the critical section. The callback function in `setReleaseCallback()` returns the number of hosts to be released.

5.3.3 Implementation

Partial barrier participants (which are also Plush clients) implement the interface described above while a separate barrier manager (usually the Plush controller) coordinates communication across nodes. While it is not required that the barrier manager run on the Plush controller, this is the default behavior for applications being managed by Plush that do not explicitly specify another barrier manager. Our implementation of partial barriers consists of approximately $3,000$ lines of C++ code, and is included as part of the core Plush codebase. At a high-level, a barrier participant calling `enter()` transmits a `BARRIER_REACHED` message using TCP to the manager with the calling host's unique identifier (hostname), barrier name, and label. The manager updates its local state for the barrier, including the set of nodes that have thus far entered the barrier, and performs appropriate callbacks as necessary. The manager starts a timer to support various release semantics if this is the first node entering the barrier and subsequently records the inter-arrival times between nodes entering the barrier.

If a sufficient number of nodes enter the barrier or a specified amount of time passes, the manager transmits `FIRE` messages using TCP to all nodes that have entered the barrier. For throttled release barriers, the manager releases the speci-

fied number of nodes from the barrier in FIFO order. The manager also sets a timer as specified by `setThrottleReleaseTimeout()` to release additional nodes from the barrier when appropriate. For semaphore barriers, the manager releases the number of nodes specified by `setSemaphoreCount()` and, if specified by `setSemaphoreTimeout()`, also sets a timer to expire for each node entering the critical section. Each call to `enter()` transmits a `SEMAPHORE_REACHED` message to the manager. When there is room in the critical section, the manager transmits a `FIRE` message to the node and starts a timer. If the semaphore timer associated with the node expires before receiving the corresponding `SEMAPHORE_RELEASED` message, the manager assumes that node has either failed or is proceeding slowly, and an additional node is released into the critical section. Each `SEMAPHORE_RELEASED` message releases one new node into the critical section.

For all barriers, the manager must gracefully handle nodes arriving late, *i.e.*, after the barrier has fired. Plush employs two techniques to address this case. For pass-through semantics, the manager transmits a `LATE_FIRE` message to the calling node, releasing it from the barrier. In catch-up semantics, the manager issues an exception and transmits a `CATCH_UP` message to the node. Catch-up semantics allow applications to respond to the exception in an application-specific manner. Depending on the application's response to the exception, the Plush controller may attempt to reintegrate the node back into the computation at a later time. The type of barrier—pass-through or catch-up—is specified at barrier creation time (omitted from Figure 5.2 for clarity).

5.3.4 Fault Tolerance

Similar to our concerns over controller failures in Plush, one concern with our centralized barrier manager is tolerating manager faults. We improve overall system robustness with support for replicated managers—just as we did for the Plush controllers—with a few added features for maintaining consistency. Our algorithm is a variant of traditional primary/backup systems: each participant maintains an ordered list of barrier managers. Any message sent from a client to the logical barrier manager

is sent to all managers in the list. Because application-specific entry callbacks may be non-deterministic, a straightforward replicated state machine approach where each barrier manager simultaneously decides when to fire is insufficient. Instead, the primary manager forwards all `BARRIER_REACHED` messages to the backup managers. These messages act as implicit "keep alive" messages from the primary. If a backup manager receives `BARRIER_REACHED` messages from clients but not the primary for a sufficient period of time, the first backup determines the primary has failed and assumes the role of primary manager. The secondary backup takes over should the primary fail again, and so on. Note that our approach admits the case where multiple managers simultaneously act as the primary manager for a short period of time. Participants ignore duplicate `FIRE` messages for the same barrier, so progress is assured, and one manager eventually emerges as primary.

Although using the replicated manager scheme described above lowers the probability of losing `BARRIER_REACHED` messages, it does not provide any increased reliability with respect to messages sent from the manager(s) to the remote hosts. All messages are sent using reliable TCP connections. If a connection fails between the manager and a remote host, however, messages may be lost. For example, suppose the TCP connections between the manager and some subset of the remote hosts break just after the manager sends a `FIRE` message to all participants. Similarly, if a group of nodes fails after entering the barrier, but before receiving the `FIRE` message, the failure may go undetected until after the manager transmits the `FIRE` messages. In these cases, the manager will attempt to send `FIRE` messages to all participants, and detect the TCP failures after the connections time out. Such ambiguity is unavoidable in asynchronous systems; the manager simply informs the application of the failure(s) via a callback and lets the application decide the appropriate recovery action. As with any other failure in Plush, the application may choose to continue execution and ignore the failures, find new hosts to replace the failed ones, or abort the execution entirely.

5.3.5 Design Alternatives

To address potential scalability problems with our centralized barrier implementation, a tree of barrier managers could be built that aggregates `BARRIER_REACHED` messages from children before sending a single message up the tree [47, 114, 73]. This tree could be built randomly from the expected set of hosts, or it could be crafted to match the underlying communication topology, in effect forming an overlay aggregation tree [104, 113]. In these cases, the manager at the root of the tree would send FIRE messages to its children, which would in turn propagate the message down the tree to the leaves. One difficult question with this approach is determining when interior nodes should pass summary `BARRIER_REACHED` messages to their parent. Although a tree-based approach may provide better scalability by aggregating messages up the tree, the latency required to pass a message to all participants may increase since the number of hops required to reach all participants is greater than in the centralized approach.

A gossip-based algorithm could also be employed to manage barriers in a fully decentralized manner [14]. In this case, each node acts as a barrier manager and swaps barrier status with a set of remote peers. Given sufficient pair-wise exchanges, some node will eventually observe enough hosts having reached the barrier and it will fire the barrier locally. Subsequent pair-wise exchanges will propagate the fact that the barrier was fired to the remainder of the participants, until eventually all active participants are informed. Alternatively, the node that determines that a barrier should be released could broadcast the `FIRE` message to all participants. Fully decentralized solutions like this gossip-based approach have the benefit of being highly fault tolerant and scalable since the work is shared equally among participants and there is no single point of failure. However, since information is propagated in a somewhat *ad hoc* fashion, it takes more time to propagate information to all participants, and the total amount of network traffic is greater. There is also an increased risk of propagating stale information. In our experience, we have not yet observed significant reliability limitations with our centralized barrier implementation to warrant exploring a fully decentralized approach.

We expect that all single-manager algorithms will eventually run into scalability limitations based on a single node's ability to manage incoming and outgoing communication with many peers. However, based on our evaluation of scalability (see Section 5.5), the performance of centralized barriers is acceptable to at least 100 nodes. In fact, we find that our centralized barrier implementation out-performs an overlay tree with an out-degree of ten for 100 total participants with regards to the time it takes a single message to propagate to all participants.

5.4 Adaptive Release

Unfortunately, the extended barrier semantics in Plush partial barriers introduce additional parameters: the threshold for early release and the concurrency level in throttled release. Experience has shown it is often difficult to select values that are appropriate across heterogeneous and changing network conditions. Hence, we provide adaptive mechanisms in Plush to dynamically determine appropriate values during execution.

5.4.1 Early Release

There is a fundamental tradeoff in specifying an early release threshold. If the threshold is too large, the application will wait unnecessarily for a relatively modest number of additional nodes to enter the barrier; if too small, the application will lose the opportunity to have participation from other nodes had it just waited a bit longer. Thus, Plush uses the barrier's callback mechanism to determine release points in response to varying network conditions and node performance.

In our experience, the distribution of node arrivals at a barrier is often heavy-tailed: a relatively large portion of nodes arrive at the barrier quickly with a long tail of stragglers entering late. In these situations, many target distributed applications would wish to dynamically determine the "knee" of a particular arrival process and release the barrier upon reaching it. Unfortunately, while it can be straightforward to manually

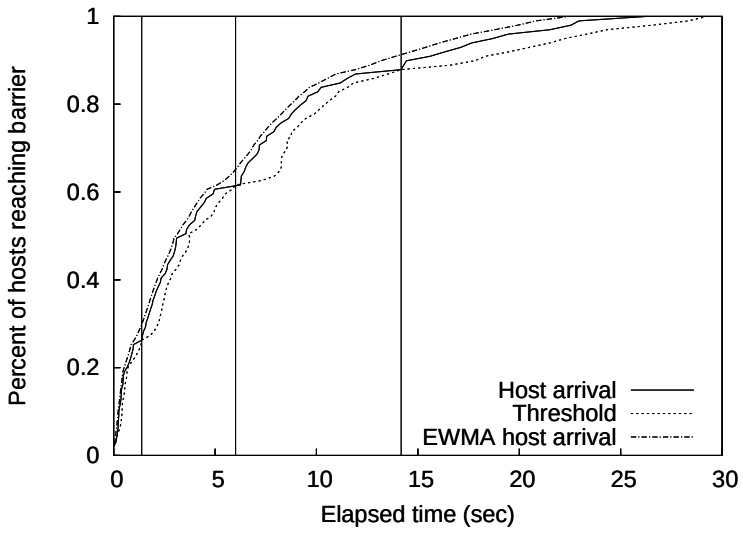

Figure 5.4: Dynamically determining the knee of arriving processes. Vertical bars indicate a knee detection.

determine the knee offline once all of the data for an arrival process is available, it is difficult to determine this point online.

The heuristic used in Plush, inspired by TCP retransmission timers and MONET [7], maintains an exponentially weighted moving average (EWMA) of the host arrival times (arr), and another EWMA of the deviation from this average for each measurement ($arrvar$). As each host arrives at the barrier, Plush records the arrival time of the host, as well as the deviation from the average. Then Plush recomputes the EWMA for both arr and $arrvar$, and use the values to compute a maximum wait threshold of $arr + 4 * arrvar$. This threshold indicates the maximum time Plush is willing to wait for the next host to arrive before firing the barrier. If the next host does not arrive at the barrier before the maximum wait threshold passes, Plush assumes that a knee has been reached. Figure 5.4 illustrates how these values interact for a simulated group of 100 hosts entering a barrier with randomly generated exponential inter-arrival times. Notice that a knee occurs each time the host arrival time intersects the threshold line.

With the capability to detect multiple knees, it is important to provide a way for applications to indicate to Plush how to pick the right knee and avoid firing earlier

or later than desired. Aggressive applications may choose to fire the barrier when the first knee is detected. Conservative applications may wish to wait until some specified amount of time has passed, or a minimum percentage of hosts have entered the barrier before firing. To support both aggressive and conservative applications, Plush partial barriers allow the application to specify a minimum percentage of hosts, minimum waiting time, or both for each barrier. If an application specifies a minimum waiting time of five seconds, knees detected before five seconds are ignored. Similarly, if a minimum host percentage of 50% is specified, the Plush knee detector ignores knees detected before 50% of the total hosts have entered the barrier. If both values are specified, the knee detector uses the more conservative threshold so that both requirements (time and host percentage) are met before firing.

One variation in the Plush approach compared to other related approaches is the values for the weights in the moving averages. In the RFC for computing TCP retransmission timers [27], the weight in the EWMA of the *rtt* places a heavier weight (0.875) on previous delay measurements. This value works well for TCP since the average delay is expected to remain relatively constant over time. In distributed applications running across the wide-area, however, we expect the average arrival time of nodes to increase, and thus we decrease the weight used in Plush to be 0.70 for previous measurements of arr. This allows the arr value to more closely follow the actual data being recorded. When measuring the average deviation, which is computed by averaging $|sample - arr|$ (where $sample$ represents the latest arrival time recorded), Plush uses a weight of 0.75 for previous measurements, which is the same weight used in TCP for the variation in *rtt*.

5.4.2 Throttled Release

Plush also employs an adaptive method to dynamically adjust the amount of concurrency in the "critical section" of a semaphore barrier. In many applications, it is impractical to select a single value which performs well under all conditions. Similar in spirit to SEDA's thread-pool controller [109], our adaptive release algorithm selects

an appropriate concurrency level based upon recent release times. The algorithm starts with a low-level of concurrency and increases the degree of concurrency until response times worsen; it then backs off and repeats, oscillating about the optimum value.

Mathematically, the algorithm used in Plush compares the median of the distributions of recent and overall release times. For example, if there are 15 hosts in the critical section when the 45th host is released, the algorithm computes the median release time of the last 15 releases, and of all 45. If the latest median is more than 50% greater than the overall median, no additional hosts are released, thus reducing the level of concurrency to 14 hosts. If the latest median is more than 10% but less than 50% greater than the overall median, one host is released, maintaining a level of concurrency of 15. In all other cases, two hosts are released, increasing the concurrency level to 16. The thresholds and differences in size are selected to increase the degree of concurrency whenever possible, but keep the magnitude of each change small.

5.5 Partial Barriers in Plush

Partial barriers are part of the core design of Plush, enabling all applications being managed by Plush to experience the benefits of relaxed synchronization semantics. Plush users have the option of specifying traditional barriers or partial barriers using barrier blocks in their application specifications. When defining partial barriers, extra parameters are defined, including the timeout, minimum percentage of nodes required, release rate, and whether the adaptive release techniques described in Section 5.4 should be used. In order for applications to achieve the maximum benefits, however, our centralized implementation of partial barriers must be scalable enough to support many hosts spread across the wide-area. Further, Plush itself uses partial barriers internally to separate different stages in an application's flow of control. Thus it is important to evaluate how effective partial barriers are in the context of application management. In this section we evaluate the scalability and performance of partial barriers in Plush.

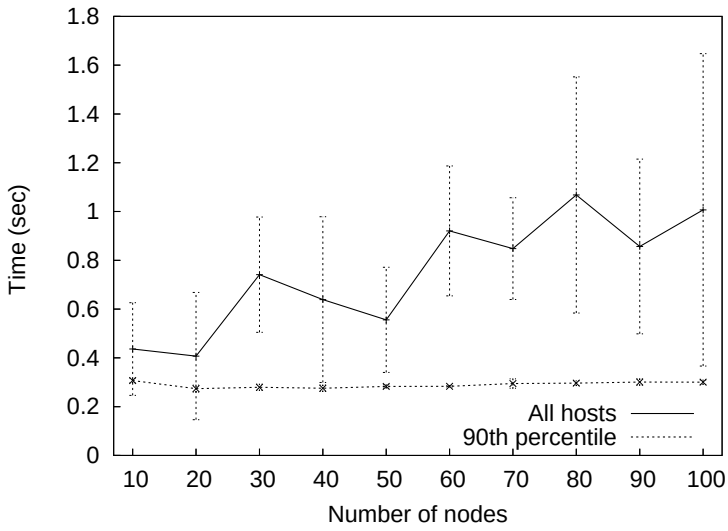

Figure 5.5: Scalability of centralized Plush barrier implementation. "All hosts" line shows the average time across five runs for barrier manager to receive BARRIER_REACHED messages from all hosts. "90th percentile" line shows the average time across five runs for barrier manager to receive BARRIER_REACHED messages from 90% of all hosts. Error bars indicate standard deviation.

5.5.1 Scalability

To estimate baseline barrier scalability in Plush, we measure the time it takes to move between two barriers for an increasing number of PlanetLab hosts. In this experiment, the Plush controller (and barrier manager) waits for all hosts to reach the first barrier. All hosts are released, and then immediately enter the second barrier. We measure the time between when the barrier manager at the Plush controller sends the first FIRE message for the first barrier and receives the last BARRIER_REACHED message for the second barrier. No partial barrier semantics are used for these measurements, since we are trying to evaluate our baseline performance. Figure 5.5 shows the average completion time for varying numbers of nodes across a total of five runs for each data point. Error bars in the graph show the standard deviation.

Notice that even for 100 nodes, the average time for the barrier manager to receive the last BARRIER_REACHED message for the second barrier is approximately one second. The large standard deviation values indicate that there is much variability

in our results. This variability is due to the presence of straggler nodes that delay the firing for several seconds or more. The 90th percentile, on the other hand, has little variation and is relatively constant as the number of participants increases. This augurs well for the potential of partial barrier semantics to improve performance in the wide-area. Overall, we were satisfied with the performance achieved using our centralized implementation for 100 nodes. Unfortunately we were unable to obtain a larger working set of PlanetLab hosts at the time this experiment was performed. However, based on our experience with using tree topologies for scalable communication in Plush (as discussed in Chapter 3), we are confident that using a tree for barrier communication will allow us to scale significantly further without sacrificing performance.

5.5.2 Admission Control

In Chapter 3 we discussed the design of the Plush architecture, consisting of the controller that directs of the flow of the execution, and the distributed clients that run on the resources involved in the application and execute commands. Thus, Plush itself is a phased distributed application that has the potential to benefit from the use of partial barriers in wide-area environments. In particular, partial barriers are especially helpful during Plush's application deployment phase to separate the tasks of node configuration and process startup. We found that in the heterogeneous and volatile PlanetLab environment, the time to configure a set of nodes with the requisite software can vary widely or fail entirely at individual hosts. In this case, we found it beneficial to timeout the internal "software configuration" barrier in Plush and either proceed with the available nodes or recruit additional nodes.

Similar to our discussion of coordinated network operations in Section 5.2, in this section we consider the benefits of using a semaphore barrier to perform admission control for parallel software installations in Plush. As part of application deployment, Plush configures a set of resources with the software required to execute a particular application. This process often involves installing the same software packages located on a central server separately on each resource. Simultaneously downloading the same

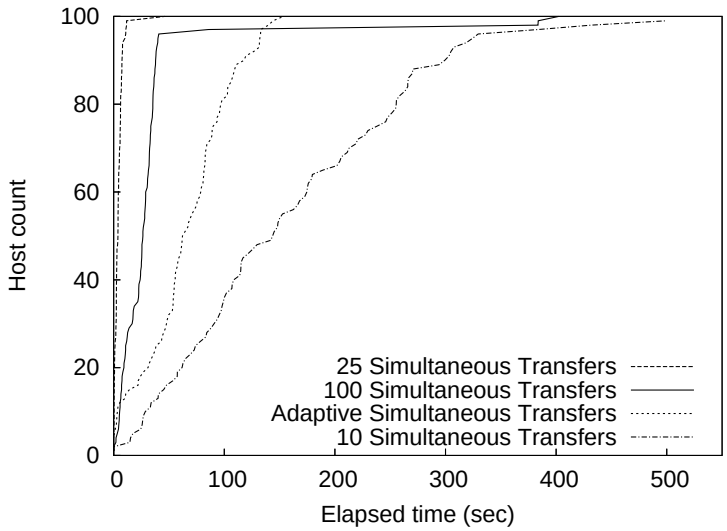

Figure 5.6: Software transfer from a high-speed server to PlanetLab hosts using a SemaphoreBarrier to limit the number of simultaneous file transfers.

software packages across hundreds of nodes can lead to thrashing at the server hosting the packages. The overall goal in using partial barriers is to ensure sufficient parallelism such that the server is saturated (without thrashing) while balancing the average time to complete the download across all participants.

For our results, we measure the time it takes Plush to install the same 10-MB file on 100 responsive and randomly chosen PlanetLab hosts while varying the number of simultaneous downloads using a semaphore barrier. Figure 5.6 shows the results of this experiment. The data indicates that limiting parallelism can improve the overall completion rate. Releasing too few hosts does not fully consume server resources, while releasing too many taxes available resources, increasing the time to completion. This is evident in the graph since 25 simultaneous downloads finishes more quickly than both ten and 100 simultaneous transfers.

While statically defining the number of hosts allowed to perform simultaneous downloads works well for our simple file transfer experiment, varying network conditions means that statically picking any single value is unlikely to perform well under all conditions. Some applications may benefit from a more dynamic throttled release tech-

nique that attempts to find the optimal number of hosts that maximizes throughput from the server without causing saturation. The "Adaptive Simultaneous Transfers" line in Figure 5.6 shows the performance of the Plush adaptive release technique as described in Section 5.4.2. In this example, the initial concurrency level is 15, and the level varies according to the duration of each transfer. In this experiment the adaptive algorithm line reaches 100% before the lines representing a fixed concurrency level of ten or 100, but the algorithm was too conservative to match the optimal static level of 25 given the network conditions at the time.

5.6 Summary

To summarize, in this chapter we showed that Plush partial barriers represent a useful relaxation of the traditional barrier synchronization primitive that targets wide-area, volatile deployment environments. In the next chapter we will show how partial barriers are easily integrated in existing applications, and we believe that our simple API bodes well for their general utility. Although we focused on the use of partial barriers in Plush in this chapter, we are hopeful that the semantics provided by partial barriers in general can be used to bring to the wide area other sophisticated parallel algorithms initially developed for tightly coupled environments. In our work thus far, we find in many cases it may be as easy as directly replacing existing synchronization primitives with their relaxed partial barrier equivalents.

5.7 Acknowledgments

Chapter 5, in part, is a reprint of the material as it appears in the USENIX Annual Technical Conference, 2006, Albrecht, Jeannie; Tuttle, Christopher; Snoeren, Alex C.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Application Case Studies

The preceding chapters explored the design and implementation of the Plush architecture, including a detailed discussion on resource management and partial barriers. In this chapter, we revisit some of our initial design goals as described in Chapter 2 and take a closer look at how Plush supports different classes of applications. In particular, we look at an example short-lived computation, long-lived service, and two parallel grid applications, and discuss how Plush manages each type of application. Additionally, we evaluate various aspects of the Plush design, including partial barriers and failure recovery, in the context of specific applications running on PlanetLab.

6.1 Short-lived Computations

In Chapter 2 we describe a short-lived computation as one that is closely monitored by the user and runs for a few days or less. In this section, we examine how Plush manages a specific short-lived computation—namely Bullet [56]—on PlanetLab. Like the example file-distribution application in Chapter 2, Bullet aims to run on PlanetLab machines with fast processors and low CPU load. In this section we show how Plush uses SWORD to satisfy these resource constraints. We also quantify the benefits of using partial barriers during application initialization within the context of Bullet.

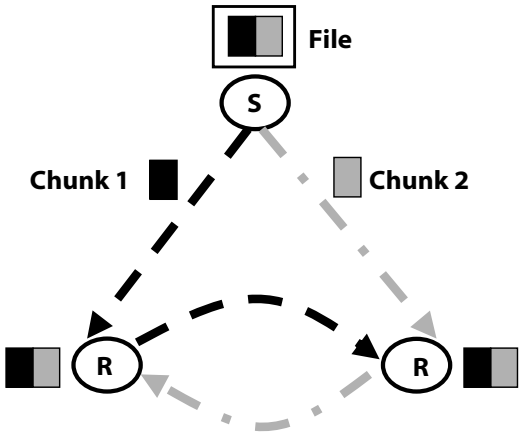

Figure 6.1: Bullet execution with one sender (S) sending to two receivers (R).

6.1.1 Managing Bullet on PlanetLab

Bullet is an overlay-based file-distribution infrastructure. In Bullet, a source transmits a file to multiple receivers spread across the Internet. Rather than waiting for the sender to send each byte (or "chunk") of the file to each receiver separately, however, Bullet leverages the parallel bandwidth available among receivers by allowing receivers to also exchange data. This decreases the total download time across all hosts and increases the overall throughput of the application. However, the receivers only benefit from this technique if they are able to obtain data from other receivers that they do not already possess. Redundant data is useless and wastes valuable network resources. To alleviate the amount of redundant data transmitted, receivers share information with one another regarding the location of specific chunks using a protocol called RanSub [55]. RanSub sends uniformly random subsets of global information throughout the overlay network. Using RanSub, the receivers learn where to obtain missing chunks of the file without waiting for the sender to send them directly. Figure 6.1 illustrates a Bullet execution with one sender and two receivers. Notice how the sender splits the file into two chunks and sends each chunk to a different receiver. As the sender transmits the chunks to each receiver, the receivers also begin exchanging data. The end result is that each receiver obtains the entire file quicker, maximizing the throughput of the application.

As part of the application initialization bootstrapping process in Bullet, all receivers join the overlay by initially contacting the source before settling on their final position in the overlay network topology. The published quantitative evaluation of Bullet presents a number of experiments across PlanetLab. However, to make performance results experimentally meaningful when measuring behavior across a large number of PlanetLab receivers, the authors hard-coded a 30-second delay at the sender from the time that it starts to the time that it begins data transmission. This delay allowed the receivers to join the overlay and figure out their position before starting transmission. While typically sufficient for the particular targeted configuration, the timeout was often too long, unnecessarily extending turnaround time for experimentation and interactive debugging. Depending on overall system load and the number of participants, the timeout was also sometimes too short, meaning that some participants did not complete the join process before the sender began transmitting. While this latter case was not a problem for the correct behavior of the application, it made interpreting experimental results difficult.

To address the limitations associated with using hard-coded initialization timeouts, we use the partial barrier API in Plush to integrate barriers directly into the Bullet source code. Since Bullet is also written in C++, this integration is straightforward, and only requires adding two lines of code to Bullet. Once the join process completes on an individual host (which means that they have successfully joined the overlay and are ready to receive data), the host simply enters a barrier. The Bullet source also registers for a callback from the Plush barrier manager to be notified of a barrier release, at which point the source begins transmitting data. This approach eliminates the problem associated with arbitrarily choosing an application initialization timeout value, since the barrier manager knows exactly when all hosts have joined the overlay. However, without partial semantics for our barriers, Bullet still suffers from poor performance due to stragglers. We observe that when choosing a substantial number of time-shared PlanetLab hosts to perform the same amount of work, the completion time varies widely in general, often following a heavy-tailed distribution. Thus most Bullet participants are

stuck waiting for the few stragglers that fall in the “tail of the curve” to finish joining the overlay before starting any file transfers.

A more desirable behavior in the Bullet initialization phase is to dynamically detect knees in the heavy-tailed join process. When the barrier manager running on the Plush controller determines that the knee of the join process has been reached, participants already in the barrier are released; one side effect is that the Bullet source host begins data transmission. Further, by calling the `getHosts()` method in the partial barrier API, the sender records the identities of participants that should be considered in interpreting the experimental results later. Note that for this particular application, any late arriving participants who enter the barrier after transmission has started are ignored by other participants and the barrier manager instructs them to exit immediately.

Figure 6.2 shows the application specification for Bullet on PlanetLab. Notice that the top of the file defines the software package, which in this case is “bullet.tar.” The component definition describes the desired resources, which include 130 PlanetLab hosts assigned to the ucsd_bullet slice. The component also includes a SWORD query that requests resources with fast processors (based on the SWORD attribute “cpuspeed,” which is measured in gigahertz) and low load (based on the attribute “fiveminload”). After the component definition, the component block specification defines the actual execution using the “run” process block. One interesting feature of this particular application specification is the redirection of terminal output on the PlanetLab hosts to a specific file. This redirection is accomplished by creating a “log_manager” within the process block. After the process block, the XML specifies the “bullet_barrier” barrier block that separates application initialization from the data transfer phase. Since we are using the Plush API directly within the modified Bullet source code, nothing is defined after the barrier block in the application specification. However, recall that the Bullet source host registers a callback with the Plush barrier manager. A release of the startup barrier signals the end of application initialization, and thus begins the transfer of data.

```
<?xml version="1.0" encoding="utf-8"?>
<plush>
  <project name="Bullet">
    <software name="bullet" type="tar">
      <package name="bullet.tar" type="web">
        <path>http://strength.ucsd.edu/~albrecht/bullet.tar</path>
        <dest>bullet.tar</dest>
      </package>
    </software>
    <component name="bullet_hosts">
      <rspec>
        <num_hosts>130</num_hosts>
        <sword>
          <request>
            <group>
              <name>bullet_hosts</name>
              <num_machines>all</num_machines>
              <fiveminload>0, 0, 2, 5, 1</fiveminload>
              <cpuspeed>1, 2, 5, 5, 1</cpuspeed>
            </group>
          </request>
        </sword>
      </rspec>
      <software name="bullet" />
      <resources>
        <resource type="planetlab" group="ucsd_bullet" />
      </resources>
    </component>
    <application_block name="Bullet">
      <execution>
        <component_block name="run_bullet">
          <component name="bullet_hosts" />
          <process_block name="run">
            <process name="appmacedon">
              <path>./appmacedon</path>
              <cwd>bullet</cwd>
              <log_manager type="default" use_api="true">
                <fd fd="1" type="file" path_prefix="bulletlog-" path_postfix=".txt" />
              </log_manager>
            </process>
          </process_block>
          <barrier_block name="bullet_barrier">
            <predecessor name="run" />
            <barrier name="ready to stream" type="1" max="130" knee_det="true"/>
          </barrier_block>
        </component_block>
      </execution>
    </application_block>
  </project>

</plush>
```

Figure 6.2: Bullet application specification.

6.1.2 Detecting Knees in Bullet

In this section we quantify the benefits of using partial barriers with knee detection during the application initialization phase of Bullet. Figure 6.3 plots the cumulative distribution of receivers that enter the startup barrier on the y-axis as a function

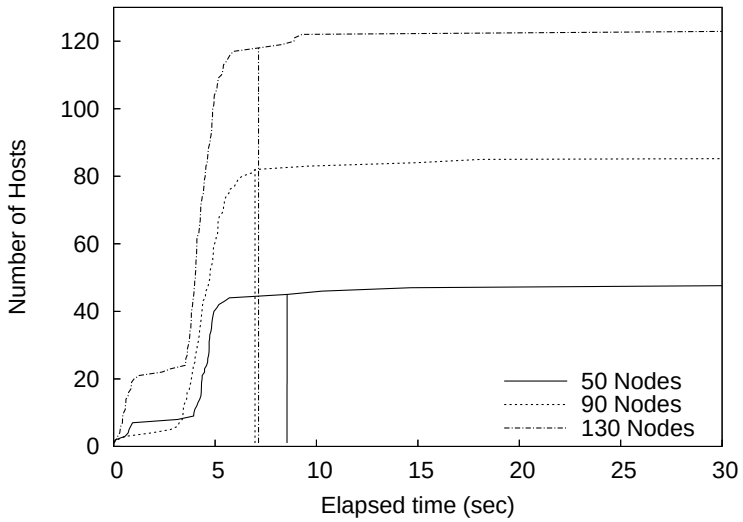

Figure 6.3: A startup barrier that regulates participants joining a large-scale overlay network in Bullet. Vertical bars indicate when the Plush barrier manager detects a knee and releases the barrier.

of time progressing on the x-axis. Each curve corresponds to an experiment with 50, 90, or 130 PlanetLab receivers in the initial target set. The goal is to run with as many receivers as possible from the given initial set without waiting an undue amount of time for a small number of stragglers to complete startup. Interestingly, it is insufficient to filter for any static set of known "slow" resources on PlanetLab as performance tends to vary on fairly small time scales and is influenced by multiple factors (such as CPU load, memory, and changing network conditions). Thus, manually choosing an appropriate static set may be sufficient for one particular batch of runs but not likely the next.

Vertical lines in Figure 6.3 indicate where the barrier manager detects a knee and releases the barrier. Although we ran the experiment multiple times, for clarity we plot the results from a single run. While differences in time of day or initial host characteristics affect the quantitative behavior, the general shape of the curve is maintained in each run. However, in all of our experiments, we are satisfied with our ability to dynamically determine the knee of the arrival process. The experiments are typically able to proceed with 85-90% of the initial set participating, and wait no more than eight seconds to begin transmission.

6.2 Long-lived Services

In addition to short-lived computations, Chapter 2 describes the typical application characteristics of long-running Internet services. To summarize, a long-running service is not closely monitored by the operator and typically runs for months or years. Many long-running services aim to run on as many resources as possible and are exposed to many different types of failures due to network and host variability and volatility. In this section, we consider how Plush manages a distributed version of SWORD running across all available PlanetLab hosts. Further, we evaluate the ability of the Plush controller to automatically detect and recover from failures in SWORD.

6.2.1 Managing SWORD on PlanetLab

SWORD, which is discussed in detail in Chapter 4, is an example of a long-running PlanetLab service. Specifically, SWORD is a resource discovery service that helps PlanetLab users find the best set of resources available to host their applications. Recall that the original version of SWORD was fully distributed and stored data in a DHT. In this distributed architecture, data from the DHT is used to respond to queries for groups of resources that have specific characteristics. Distributed SWORD aims to run on as many PlanetLab hosts as possible. This distributed design spreads the load of the system across many hosts allowing for increased scalability (since there is less work for each individual host when the total number of hosts is greater), and also allows SWORD to accurately respond to queries using information from a larger number of PlanetLab resources. The following paragraphs describe how Plush manages distributed SWORD.

The XML application specification for SWORD is shown in Figure 6.4. As in Bullet, the top half of the specification defines the SWORD software package and the component required for the application. Notice that SWORD uses one component consisting of hosts assigned to the ucsd_sword PlanetLab slice. An interesting feature of this component definition is the “num_hosts” tag. Since SWORD is a service that wants

```
<?xml version="1.0" encoding="utf-8"?>
<plush>
  <project name="sword">
    <software name="sword_software" type="tar">
      <package name="sword.tar" type="web">
        <path>http://plush.ucsd.edu/sword.tar</path>
        <dest>sword.tar</dest>
      </package>
    </software>
    <component name="sword_participants">
      <rspec>
        <num_hosts min="10" max="800"/>
      </rspec>
      <resources>
        <resource type="planetlab" group="ucsd_sword"/>
      </resources>
      <software name="sword_software"/>
    </component>
    <application_block name="sword_app_block" service="1" reconnect_interval="300">
      <execution>
        <component_block name="participants">
          <component name="sword_participants"/>
          <process_block name="sword">
            <process name="sword_run">
              <path>dd/planetlab/run-sword</path>
            </process>
          </process_block>
        </component_block>
      </execution>
    </application_block>
  </project>

</plush>
```

Figure 6.4: SWORD application specification.

to run on as many nodes as possible, we specify a range of acceptable values rather than a single number. Hence, as long as a minimum of ten hosts are available, Plush continues managing SWORD. Since the max value is set to 800, Plush does not look for more than 800 resources to host SWORD. Currently, PlanetLab contains less than 800 hosts, which means that Plush attempts to run SWORD on all available PlanetLab resources. The lower half of the application specification defines the application block, component block, and process block that describe the SWORD execution.

The application block specification for SWORD is similar to the application block specification of Bullet, save a few important differences. When defining the application block object for SWORD, we include special "service" and "reconnect_interval" attributes. The service attribute tells the Plush controller that SWORD is a long-running service and requires different default behaviors for initialization and failure recovery. For example, during application initialization, the controller does not wait for all par-

ticipants to install the software before starting all hosts simultaneously. Instead, the controller instructs individual clients to start the application as soon as they finish installing the software, since there is no reason to synchronize the execution across all hosts. Further, if a process fails when the service attribute has been specified, the controller attempts to restart SWORD on that host without aborting the entire application. The reconnect_interval attribute specifies the period of time the controller waits before rerunning the resource discovery and acquisition unit. For long-running services, hosts often fail and recover during execution. The reconnect_interval attribute value tells the Plush controller how often to check for new hosts that have come alive since the last run of the resource discovery unit. The controller also unsets any "failed host" tags in the matching Node objects at this time. Rerunning the resource discovery and acquisition unit is the controller's way of "refreshing" the list of available hosts. The controller continues to search for new hosts until reaching the maximum num_hosts value, which is 800 in our case.

6.2.2 Evaluating Fault Tolerance in SWORD

To demonstrate Plush's ability to automatically recover from host failures for long-running services, we run SWORD on PlanetLab with 100 randomly chosen hosts, as shown in Figure 6.5. The host set includes machines behind DSL links as well as hosts from other continents. When Plush starts the application, the controller starts the Plush client on 100 randomly chosen PlanetLab machines, and each machine begins downloading the SWORD software package (38-MB). It takes approximately 1000 seconds for all hosts to successfully download, install, and start SWORD. At time $t = 1250s$, we kill the SWORD process on 20 randomly chosen hosts to simulate host failure. (Normally, Plush would automatically try to restart the SWORD process on these hosts. However, we disable this feature for this experiment to simulate host failures and force a rematching.) After killing the SWORD processes, the Plush controller detects that the processes and hosts have failed, and the controller begins to find replacements for the failed machines. The replacement hosts join the Plush overlay and start downloading

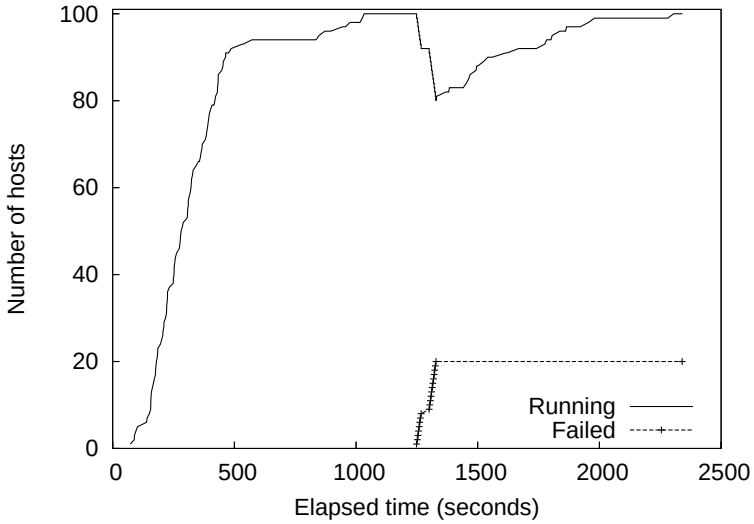

Figure 6.5: SWORD running on 100 randomly chosen PlanetLab hosts. At $t = 1250$ seconds, we fail 20 hosts. The Plush controller finds new hosts, who start the Plush client process and begin downloading and installing the SWORD software. Service is fully restored at approximately $t = 2200$ seconds.

the SWORD software. As before, Plush chooses the replacements randomly, and low bandwidth/high latency links have a great impact on the time it takes to fully recover from the host failure. At $t = 2200s$, the service is restored on 100 machines.

Using Plush to manage long-running services like SWORD alleviates operators of the burden of manually probing for failures and configuring/reconfiguring hosts. Further, Plush interfaces directly with the PlanetLab Central (PLC) API, which means that users can automatically add hosts to their slice and renew their slice using Plush. This feature is beneficial since services typically want to run on as many PlanetLab hosts as possible, including any new hosts that come online after initially starting the service. By periodically contacting PLC and retrieving the master list of PlanetLab hosts, the Plush controller maintains an up-to-date list of all PlanetLab resources, and is able to notify the service operator if new resources are available. In addition, Plush simplifies the task of debugging problems by providing a single point of control for all connected PlanetLab hosts. Thus, if a user wants to view the memory consumption of their service across all connected hosts, a single Plush command retrieves this information, making it easier to monitor a service running on hundreds of resources around the world.

6.3 Parallel Grid Applications

In this section we consider how Plush manages typical grid applications. Recall again from Chapter 2 that grid applications tend to be computationally intensive and easily parallelizable. Grid applications also tend to operate in phases that are easily separated by barriers. Thus, many grid applications have the potential to achieve higher performance by using partial barriers to reassign unfinished tasks on slow hosts to hosts that have already completed their assigned work. We consider two grid applications in this section, namely EMAN and MapReduce, and show how Plush manages their execution. Additionally, we show how partial barriers significantly improve the performance achieved across the wide-area.

6.3.1 Managing EMAN on PlanetLab

To illustrate how Plush manages applications with workflows, we consider running EMAN (described in Chapter 2) on PlanetLab. The computationally intense portion of EMAN's execution is the refinement stage, which is run repeatedly on 2-D electron micrograph images until achieving the desired level of detail in a 3-D model of the electron. Refinement is often run in parallel on multiple machines to improve performance. The EMAN refinement stage is a common example of a workflow in a scientific parallel application. In this section we describe how Plush runs a single round of the parallel refinement computation.

Figure 6.6 shows the application specification for EMAN. Note that we did not change the EMAN source code at all to run these experiments. Instead we wrote a simple 50-line wrapper perl script (called "eman.pl") that runs the publicly available EMAN software package [32]. As in the preceding examples, the application specification contains two main sections of interest. The top section defines the required software and components. The software required for EMAN is contained in a tarball called "eman.tar." The resources for EMAN, as specified in the component definition, are 98 PlanetLab hosts from the ucsd_plush slice. The lower section of the applica-

```
<?xml version="1.0" encoding="utf-8"?>
<plush>
  <project name="eman_proj">
    <software name="EmanSoftware" type="tar">
      <package name="eman.tar" type="web">
        <path>http://plush.ucsd.edu/eman.tar</path>
        <dest>eman.tar</dest>
      </package>
    </software>
    <component name="EmanGroup1">
      <rspec>
        <num_hosts>98</num_hosts>
      </rspec>
      <software name="EmanSoftware" />
      <resources>
        <resource type="planetlab" group="ucsd_plush" />
      <resources>
    </component>
    <application_block name="eman_app_block">
      <execution>
        <component_block name="eman_comp_block">
          <component name="EmanGroup1" />
          <workflow_block name="eman_workflow_block" id="eman_wf" num_tasks="98">
            <process_block name="eman_proc_block">
              <process name="eman">
                <path>./eman.pl</path>
                <cmdline>
                  <substitution name="sub" id="eman_wf" type="workflow" flag="--i" />
                </cmdline>
              </process>
            </process_block>
          </workflow_block>
        </component_block>
      </execution>
    </application_block>
  </project>

</plush>
```

Figure 6.6: EMAN application specification. Plush uses this specification to configure the resources, which are 98 PlanetLab hosts from the ucsd_plush slice. Each host runs "eman.pl --i n", where n identifies each unique task, as specified by the workflow block.

tion specification consists of the component, process, and workflow blocks that define the EMAN refinement execution. One interesting characteristic of this application is the workflow block within the component block. The workflow block indicates that 98 tasks are shared among the 98 workers requested in the "EmanGroup1" component. The workflow block also has a process block containing the "eman.pl" process.

The substitution information in the process definition within the process block is used in conjunction with the EMAN perl script to split the workflow among the resources. Notice how the workflow block has an "id" attribute that is identical to the "id" attribute in the process substitution. In this case, "eman.pl" uses a command-line argu-

ment to specify the unique id of the task, which is then used to determine what fraction of the data files should be processed by each host. The workflow block substitutes the current task id (an integer between 1 and 98) for the command-line argument defined by the "--i" flag. For example, the first resource runs "./eman.pl --i 1," the second runs "./eman.pl --i 2," and so on. This technique divides and distributes the work evenly among the 98 PlanetLab workers.

Plush workflow blocks are unique because they actually contain a "hidden" internal barrier. As workflow tasks are completed, the internal barrier is entered with a label that specifies the unique id of the completed task. Using the partial barrier knee detector, the barrier manager determines when a knee is reached in the rate of completion of these tasks, indicating that a subset of resources are not operating as quickly as the rest. When a knee is detected, the tasks assigned to the slow resources (due to slow or busy processors) are redistributed to faster resources that have already completed their tasks. By using the knee detector to detect stragglers in this way, the knee detector also detects resources with low bandwidth capacities based on their slow download times and reallocates their work to machines with higher bandwidth. Our experiment requires approximately 240-MB of data to be transferred to each participating PlanetLab host, and machines with low bandwidth links have a significant impact on the overall completion time if their work is not reallocated to faster machines.

6.3.2 Work Reallocation in EMAN

We now evaluate an alternative use of partial barriers in Plush: to not only assist with the synchronization of tasks across physical hosts, but also to assist with work reallocation and load balancing for hosts spread across the wide-area. Further, we determine whether we can dynamically detect knees in the completion rate of individual hosts, and subsequently reallocate unfinished work to hosts that have already completed their assigned tasks.

To quantify the effectiveness of partial barriers in EMAN, we measure the time it takes to complete all 98 tasks with and without partial semantics. Without par-

tial semantics, the 98 tasks are allocated to 98 PlanetLab resources, and we measure the time it takes for all 98 resources to complete their single task. With partial semantics, we allow the Plush controller (and barrier manager) to detect a knee in the task completion curve, and then Plush reallocates unfinished tasks to faster resources. In this experiment we run EMAN on 98 responsive PlanetLab machines. The workflow consists of a 98-way image classification run in parallel across all resources. We measure the time it takes for each participant to download a 40-MB software archive containing the EMAN executables and a wrapper script, unpack the archive, download a unique 200-MB image file, and run the image classification process. At the end of the computation, each resource generates 77 output files stored on the local disk, which are later merged into 77 "master" files once all tasks complete across all resources.

Figure 6.7 shows the results of running EMAN on PlanetLab with and without partial barrier semantics. The Plush knee detector detects two knees in this experiment at $t = 300s$ and $t = 801s$. The first knee at $t = 300s$ indicates that around 21 hosts have good connectivity to the data repository, while the rest have longer transfer times. However this first knee is ignored by the Plush controller due to a minimum threshold of 60% at the partial barrier, which prevents task reconfiguration at this point. The second knee is detected at $t = 801s$ after 78 hosts have completed their work. Since more than 60% of the hosts have entered the barrier at the second knee, the Plush controller redistributes the 20 unfinished tasks. These tasks complete by 900 seconds, as shown by the dotted line in Figure 6.7. The experiment on the original set of hosts continues past $t = 2700s$, as indicated by the solid line in the graph, resulting in an overall speedup factor of more than three using partial semantics.

6.3.3 Managing MapReduce on PlanetLab

MapReduce [29] is a toolkit for application-specific parallel processing of large volumes of data. The model involves partitioning the input data into smaller *splits* of data, and spreading them across a cluster of worker nodes. Each worker node applies a *map* function to the splits of data that they receive, producing intermediate key/value

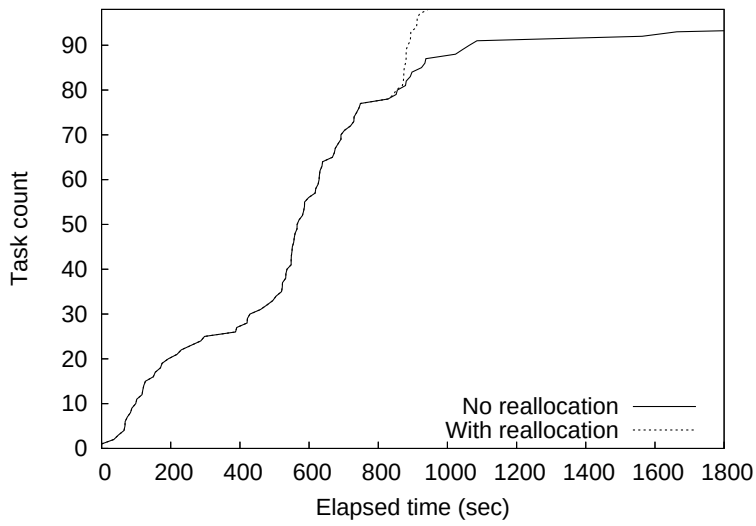

Figure 6.7: EMAN. Knee detected at 801 seconds. Total runtime (without knee detection) is over 2700 seconds.

pairs that are periodically written to specific locations on disk. The MapReduce master node tracks these disk locations, and eventually notifies another set of worker nodes that intermediate data is ready for processing. This second set of workers aggregate the data and pass it to the *reduce* function. This function processes the data to produce a final output file.

Our implementation of MapReduce leverages partial barriers to manage phases of the computation and to orchestrate the flow of data among nodes across the wide-area. Note that we do not code to the partial barrier API directly as in Bullet, but instead define workflow barriers with partial semantics in our Plush application specification. In our design of MapReduce, we have m map tasks and corresponding input files, n total nodes hosting the computation, and r reduce tasks. The Plush controller distributes the m split input files to a set of available nodes, and spawns the map process on each node. When the map tasks finish, intermediate files are written back to a central repository, and then redistributed to r hosts, who eventually execute the r reduce tasks. There are a number of natural barriers in this application corresponding to the completion of: i) the initial distribution of m split files to appropriate nodes; ii) executing m

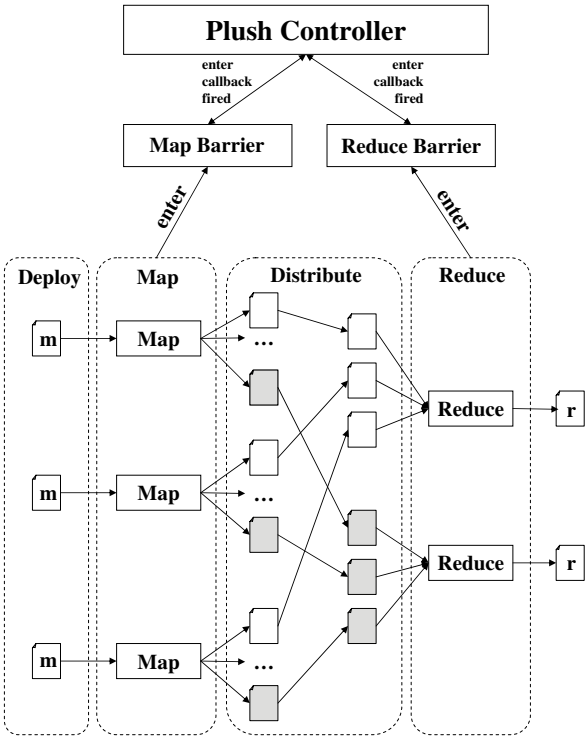

Figure 6.8: MapReduce execution. As each map task completes, "enter" is called. Once all m tasks enter the Map barrier, the barrier is released, causing the r reduce tasks to be distributed and begin execution. When all r reduce tasks have entered the Reduce barrier, MapReduce is complete. In both barriers, "callback" informs the Plush controller of task completion rates for possible task rebalancing.

map functions; iii) the redistribution of the intermediate files to appropriate nodes, and iv) executing r reduce functions.

As with the original MapReduce work, the load balancing aspects corresponding to barriers (ii) and (iv) (from the previous paragraph) are of particular interest. These barriers are shown in Figure 6.8. Recall that although there are m map tasks, the same physical host may execute multiple map tasks. Hence the goal is not necessarily to wait for all n hosts to reach the barrier, but for all m or r logical tasks to complete. Thus, we extended the Plush barrier entry semantics described in Section 5.3 to support synchronizing barriers at the level of a set of logical, uniquely named tasks or processes, rather than a set of physical hosts. To support this extension, we use a workflow barrier

in Plush that internally calls the `enter()` method of the Plush barrier API (see Figure 5.2) upon completing a particular map or reduce function. In addition to the physical hostname, we send a label corresponding to a globally unique name for the particular map or reduce task. Thus, rather than waiting for n hosts to arrive, the barrier instead waits for m or r unique labels to enter the barrier before firing.

Our experiences running EMAN on PlanetLab without partial semantics revealed that while most nodes complete their assigned tasks quickly, the overall completion time is often dominated by the performance of a small number of slow nodes. The original MapReduce work also noted that one of the common problems experienced during execution is the presence of straggler nodes that take an unusually long time to complete a map or reduce task. Although the authors mentioned an application-specific solution to this problem, by using partial barriers in our implementation, we are able to provide a more general solution that achieves the same results. In particular, we use the arrival rate of map/reduce tasks at their respective barriers to respawn a subset of the tasks that were proceeding slowly.

By using the partial barrier knee detector described in Section 5.4, the Plush controller dynamically determines the transition point between rapid arrivals and the long tail of stragglers. However, rather than releasing the barrier at this point, the Plush controller instead performs load rebalancing functionality by spawning additional copies of outstanding tasks on nodes disjoint from the ones hosting the slower tasks (potentially first distributing the necessary input/intermediate files). This technique is similar to the previous EMAN example, except that we are now considering multiple tasks per resource, rather than only one task per resource, allowing for finer granularity reconfiguration. Note that as in the original implementation of MapReduce, the barrier is not concerned with what copies of the computation complete first; the goal is for all m or r tasks to complete as quickly as possible. Thus is it possible for a single task to actually be completed twice due to reconfiguration.

6.3.4 Task Reallocation in MapReduce

To evaluate the benefits achieved from rebalancing tasks across hosts after detecting a knee, we use Plush to manage our MapReduce implementation with $m = 480$ map tasks and $r = 30$ reduce tasks running across $n = 30$ PlanetLab hosts. As in EMAN, we use workflow blocks with hidden internal barriers to handle the functionality shown by the map and reduce barriers in Figure 6.8. During each of the map and reduce rounds, the Plush controller evenly partitions the tasks over 30 PlanetLab resources and starts the tasks asynchronously. For this experiment, each map task simply reads 2000 random words from a local file and counts the number of instances of certain words. This count is written to an intermediate output file based on the hash of the words. The task is CPU-bound and requires approximately seven seconds to complete on an unloaded PlanetLab-class machine. The reduce tasks summarize these intermediate files with the same hash values. The application specification for MapReduce is shown in Figure 6.9.

In our MapReduce implementation, each map and reduce task performs an approximately equal amount of work as in the original MapReduce work, though it would be useful to generalize to variable-length computation. When complete, a map or reduce task enters the associated barrier with a unique identifier for the completed task. The barrier manager running on the Plush controller monitors the arrival rate and dynamically determines the knee, which is the where the completion rate begins to slow. We empirically determined that this slowing results from a handful of nodes that proceed substantially slower than the rest of the system. (Note that this phenomenon is not restricted to our wide-area environment; Dean and Ghemawat observed the same behavior for their runs on tightly coupled and more homogeneous clusters [29].) Thus, upon detecting the knee the Plush controller respawns additional copies of the slow tasks, ideally on nodes with the smallest number of outstanding tasks. Experience has shown that in most cases, by the time the knee is detected there are a number of hosts that have completed their initial allocation of work.

Figure 6.10 shows the performance of one MapReduce run both with and without task respawn upon detecting the knee. Figure 6.10 plots the cumulative number of

```
<?xml version="1.0" encoding="UTF-8"?>
<plush>
  <project name="map-reduce">
    <software name="map-reduce" type="tar">
        <package name="map-reduce.tar" type="web">
            <path>http://plush.ucsd.edu/map−reduce.tar</path>
            <dest>map−reduce.tar</dest>
        </package>
    </software>
    <component name="Component">
        <software name="map-reduce"/>
        <rspec>
          <num_hosts>30</num_hosts>
        </rspec>
        <resources>
          <resource type="planetlab" group="ucsd_plush"/>
        </resources>
    </component>
    <application_block name="map-reduce">
      <execution>
        <component_block name="map-reduce">
          <component name="Component"/>
          <workflow_block name="map" id="map_wf" num_tasks="480" num_workers="30">
            <process_block name="map-process-block">
              <process name="Map">
                <path>./map.pl</path>
                <cmdline>
                  <arg>−−R</arg>
                  <arg>30</arg>
                  <substitution name="map" id="map_wf" type="workflow" flag="--i"/>
                </cmdline>
              </process>
            </process_block>
          </workflow_block>
          <workflow_block name="reduce" id="reduce_wf" num_tasks="30" num_workers="30">
            <process_block name="reduce-process-block">
              <process name="Reduce">
                <path>./reduce.pl</path>
                <cmdline>
                  <arg>−−M</arg>
                  <arg>480</arg>
                  <substitution name="reduce" id="reduce_wf" type="workflow" flag="--i"/>
                </cmdline>
              </process>
            </process_block>
          </workflow_block>
        </component_block>
      </execution>
    </application_block>
  </project>

</plush>
```

Figure 6.9: MapReduce application specification. Each PlanetLab host will run "map.pl" and "reduce.pl," as specified by the workflow blocks.

completed tasks on the y-axis as a function of time progressing on the x-axis. We see that the load balancing enabled by barrier synchronization on abstract tasks is critical to overall system performance. With task respawn using knee detection, the barrier manager detects the knee at approximately $t = 68s$ after approximately 53% of the tasks

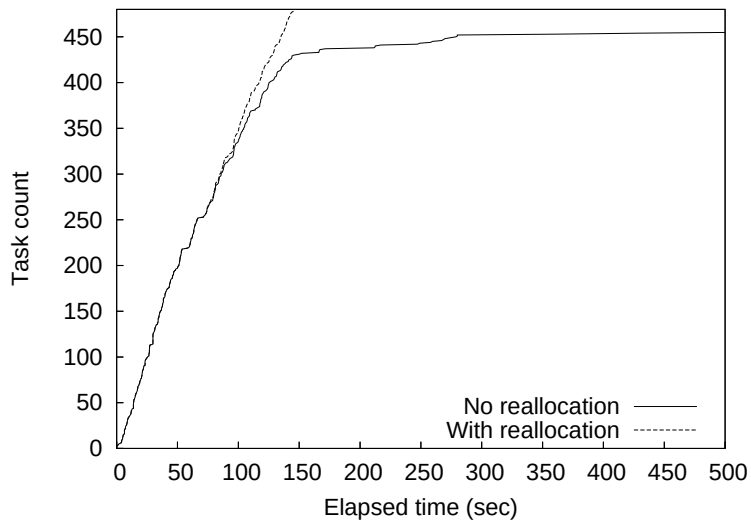

Figure 6.10: MapReduce: $m = 480, r = 30, n = 30$ with uniform prepartitioning of the data. Knee detection occurs at 68 seconds and callbacks enable rebalancing.

have completed. After detecting the knee, the Plush controller repartitions the remaining 47% of the tasks across available wide-area nodes. This point is where the curves significantly diverge in the graph. Without dynamic rebalancing the completion rate transitions to a long-tail lasting more than 2500 seconds (though the graph only shows the first 500 seconds), while the completion rate largely maintains its initial slope when rebalancing is enabled. Overall, our barrier-based rebalancing results in a factor of sixteen speedup in completion time compared to proceeding with the initial mapping of tasks to hosts. Multiple additional runs showed similar results.

Note that this load balancing approach differs from the alternate approach of trying to predict the set of nodes likely to deliver the highest level of performance *a priori*. Unfortunately, predicting the performance of tasks with complex resource requirements on a shared computing infrastructure with dynamically varying CPU, network, and I/O characteristics is challenging in the best case and potentially intractable. We advocate a simpler approach that does not attempt to predict performance characteristics in advance. Rather, we simply choose nodes likely to perform well and empirically observe the utility of our decisions. Of course, this approach may only be appropriate for

a particular class of distributed applications and comes at the cost of performing more work in absolute terms because certain computations are repeated. For the case depicted in Figure 6.10, approximately 30% of the work is repeated if we assume that the work on both the fast and slow nodes are run to completion (a pessimistic assumption as it is typically easy to kill tasks running on slow nodes once the fast instances complete).

6.4 Summary

In this chapter we discussed Plush's ability to manage different types of applications that were run on PlanetLab, including short-lived computations, long-lived services, and parallel grid applications. In particular, we started by describing how Plush managed Bullet, and showed that partial barriers with automatic knee detection allowed us to remove the arbitrary timeout value previously used for application initialization. Next, we evaluated how Plush automatically recovered from failures in SWORD, and showed that Plush successfully detected the failures, found replacement hosts, and fully restored the service in less than 20 minutes. To better understand how parallel grid applications are managed by Plush and benefit from the use of partial barriers, we provided two example applications: EMAN and MapReduce. In both cases, the use of partial barriers to detect slow participants and redistribute work resulted in a significant speedup with respect to total task completion time. Based on our experiences with using Plush to manage all of these applications, we believe that Plush is well-suited for a range of distributed applications.

6.5 Acknowledgments

Chapter 6, in part, is a reprint of the material as it appears in the USENIX Annual Technical Conference, 2006, Albrecht, Jeannie; Tuttle, Christopher; Snoeren, Alex C.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in part, has been submitted for publication of the material as it appears in the Large Installation System Administration Conference, 2007, Albrecht, Jeannie; Braud, Ryan; Dao, Darren; Topilski, Nikolay; Tuttle, Christopher; Snoeren, Alex C.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Conclusions and Future Work

In conclusion, Plush is an extensible application control infrastructure designed to meet the demands of a variety of distributed applications. Plush provides abstractions for resource discovery, creation, acquisition, software installation, process execution, and failure management in distributed environments. When an error is detected, Plush has the ability to perform several application-specific actions, including restarting the computation, finding a new set of resources, or attempting to adapt the application to continue execution and maintain liveness. In addition, Plush provides relaxed synchronization primitives in the form of partial barriers that help applications achieve good throughput even in unpredictable wide-area conditions where traditional synchronization primitives are too strict to be effective. The mechanisms provided by Plush help researchers cope with the limitations inherent to large-scale networked systems, allowing them to focus on the design and performance of their application rather than managing the deployment during the application development life cycle.

To evaluate the effectiveness of the abstractions provided by Plush, we used Plush to manage several different distributed applications—namely, Bullet, SWORD, EMAN, and MapReduce—run across the wide-area. In addition, we showed that the performance of these applications improved due to Plush's failure recovery mechanisms and relaxed synchronization semantics. Further, we showed how Plush manages resources from a variety of deployment environments by using a common interface to

interact with external resource management services, including SWORD, Mission, Shirako, and Usher. By integrating Plush with these external services, Plush supports execution on PlanetLab hosts, Xen virtual machines, and ModelNet emulated resources.

Plush is in daily use by researchers worldwide, and user feedback has been largely positive. Most users find Plush to be an "extremely useful tool"[1] that provides a user-friendly interface to a powerful and adaptable application control infrastructure. Other users claim that Plush is "flexible enough to work across many administrative domains (something that typical scripts do not do)." Further, unlike many related tools, Plush does not require applications to adhere to a specific API, making it easy to run distributed applications in a variety of environments. Our users tell us that Plush is "fairly easy to get installed and setup on a new machine. The structure of the application specification largely makes sense and is easy to modify and adapt."

Although Plush has been in development for over three years now, some features still need improvement. One important area for enhancements is error reporting. Debugging applications is inherently difficult in distributed environments. Plush tries to make it easier for researchers to locate and diagnose errors, but accomplishing this is a difficult task. For example, one user says that "when things go wrong with the experiment, it's often difficult to figure out what happened. The debug output occasionally does not include enough information to find the source of the problem." We are currently investigating ways to allow application-specific error reporting in Plush, and ultimately simplify the task of debugging distributed applications in volatile environments.

7.1 Lessons Learned

One of the main goals of our work with the design and implementation of Plush was to address the requirements described in Chapter 2. Accomplishing this goal was not an easy task, and this section describes a few of the challenges we faced and the lessons learned during the development of Plush.

[1] The user feedback presented in this section was obtained through email and conversations with various Plush users at UCSD, Duke University, and EPFL.

7.1.1 Application Specification Design

One of the first challenges that we addressed in the design of Plush was creating an application specification capable of succinctly describing application requirements. In order for application developers to accept and use Plush, we needed to design a specification that was easy to understand but also expressive enough to support complex scenarios. Additionally, users needed the ability to define each phase of an application's life cycle within the specification. Based on early user experiences, we discovered that it is important to establish a balance between functionality and usability in the design of the application specification. If the specification is too complicated, the complexity discourages and intimidates novice users, who often get frustrated and give up. However, without support for advanced features, experienced users are unable to express all of their requirements.

When designing the XML syntax for the application specification of Plush, we decided to require only a small set of easily defined attributes, while also optionally supporting a variety of specialized features. We believe that this establishes a balance between functionality and ease of use. We separated the various components of a distributed application and described them using an extensible schema that allows users to make the application specifications as complicated or basic as desired. In the simplest case, the user only needs to define the required software (if any), the number of resources desired, and the command to run on the resources. We also supported creating the specifications graphically with the GUI so users were not forced to understand the intricate details of the XML syntax to define even complicated applications.

7.1.2 Satisfying Different Application Demands

Another problem that we overcame in the design of Plush was building a generic infrastructure that satisfied the demands of a variety of distributed applications, and yet was just as powerful as tools designed specifically for a single application. We needed to control all aspects of the distributed application life cycle without sacrificing important features available in specialized tools. We quickly realized that the best

way to do this was to use the existing tools directly, rather than trying to reinvent them. Hence, Plush is a customizable framework that provides the ability to incorporate external tools and services. Users can modify their application description to plug in the specialized tools that they need to manage their applications or the resources on which their applications will run. It is this "pluggable" aspect of Plush that allows users to run their applications in a variety of environments and interact with different resource management frameworks.

One challenge in designing an infrastructure that supports the ability to plug in arbitrary existing tools was implementing the glue code necessary to integrate each tool into Plush. This task is somewhat simplified in grid environments due to the APIs that are inherited from the Globus Toolkit [34]. Since most tools written for grid environments adhere to the same standards, writing support for new tools is easier. Unfortunately, there are no official standards or common APIs for developers to use in most other distributed environments. Hence, the integration of each tool must be addressed separately.

7.1.3 Achieving Scalability

A third challenge that we faced was scalability. We needed to design an application management infrastructure that scaled to hundreds or even thousands of heterogeneous machines. Currently, PlanetLab consists of over 700 machines at approximately 345 different locations around the world. Other computing environments contain thousands of machines distributed worldwide. In order to support distributed applications in these environments, Plush must scale to potentially thousands of machines while maintaining acceptable levels of performance. The initial design of Plush used a star topology, so that every host running the application connected directly to the controller. The limiting factor in the star design was the number of simultaneous connections that the controller could support. To address this limitation, we added support for the tree topology in addition to the star. Using trees allowed Plush to scale much further without sacrificing performance to a great extent, as discussed in Chapter 3.

Other factors aside from the communication topology in Plush also contributed to its scalability. In particular, we explored the tradeoffs between threads and events in several different architectural designs, and experienced varying degrees of success with respect to scalability and performance. First, we used a fixed-size pool of threads and looped through client connections. The problem with this approach was that the progress of the entire application was limited by a few slow hosts. Although we could scale to several hundred machines, the performance was unacceptable. To avoid the potential bottleneck created by slow hosts, we increased the number of threads in use so that each client connection used two separate threads. The performance of this technique was much improved over the fixed-size thread pool. However this approach suffered from a variety of new problems, and ultimately could not scale beyond approximately 200 connections before some machines ran out of threads. Finally, we moved to the current event-driven design that uses a single thread and an event loop for execution. We are pleased with the performance of this approach thus far, and the number of client connections can scale to approximately 800 per machine, which is more than sufficient for even large tree topologies.

7.2 Future Work

Moving forward, one important problem that we have recently started to address in Plush is debugging and monitoring running applications. We plan to extend Plush to allow users to specify application-specific metrics and error detection techniques, so that potential problems are easily identified before a failure occurs, further increasing application reliability. Related to monitoring is visualizing distributed applications. On wide-area platforms, understanding exactly what each host is doing at a given point in time is no easy task. Real-time distributed application visualization is an interesting problem that we have only begun to investigate during the development of Nebula. We still have many ideas that we plan to incorporate into Nebula that will further enhance the user's ability to visualize applications. We believe that visualization is also a

key component to lowering the entry barrier for distributed systems research. The functionality provided by Nebula should make it easier for novice researchers—including undergraduates—to focus on the design and analysis of their applications rather than spending the majority of their time managing their testing environment. A key benefit of shared distributed computing platforms like PlanetLab is resource accessibility and availability, and we hope to take advantage of this benefit by using Plush in undergraduate classroom settings in the future.

Aside from monitoring and visualization, we hope to further increase the extensibility of Plush by providing support for emerging execution environments. While most of the work in this thesis focused on PlanetLab, Plush also supports (as discussed in Chapter 4) virtual machine resources. We believe it is important to extend Plush's current abstractions even further and provide advanced support for virtualization. One way to achieve this goal is to provide a tighter integration with virtual machine management systems, including Shirako and Usher. When dealing with virtual machine management systems, Plush can be used in one of two ways. In one approach, Plush manages all aspects of the application's execution, and the resource management system simply provides Plush with the resources needed to host the application. In the second approach, the resource management system runs the application using the remote execution functionality provided by Plush. Thus, the resource management framework creates the virtual machine resources, and then uses the Plush XML-RPC programmatic interface to run the user's applications. Note that the end result is the same in both cases—put simply, Plush runs an application on virtual machine resources. However from a design standpoint, there are significant differences relating to the delegation of control. Moving forward, we would like to explore the tradeoffs associated with both approaches, and ensure that Plush continues to support both usage scenarios,

Virtual machine environments also introduce new scalability challenges for Plush. Virtualization allows tens to hundreds of virtual machines to run on a single physical computer, resulting in an increased number of machines on which to run distributed applications. Hence, a single cluster composed of 500 physical computers each

running 100 virtual machines yields 5,000 total machines. Scaling Plush to manage applications on 5,000 machines is a significant research challenge that we have not yet explored.

In addition to advanced support for virtualization, we also envision Plush being a core component for experiment management and support in the GENI (Global Environment for Network Innovations) project [39]. GENI is supported by the National Science Foundation, and many view it as a blueprint for the Future Internet. The goal of GENI is to increase the quality and quantity of experimental research in computer networks and distributed systems. For GENI to achieve this, it must be accessible to the broadest set of researchers. One of the biggest benefits of shared platforms like PlanetLab and GENI is that they enable researchers at small schools with limited resources to perform large-scale research in distributed systems. However many of these researchers are inexperienced with the complexities of deploying and monitoring applications in volatile environments, and struggle to make progress. We believe that Plush provides the functionality needed to help platforms like GENI and PlanetLab be more accessible by simplifying distributed application management, and thus we hope that Plush will be instrumental in making the GENI vision a success.

7.3 Acknowledgments

Chapter 7, in part, is a reprint of the material as it appears in the ACM Operating Systems Review, January 2006, Albrecht, Jeannie; Tuttle, Christopher; Snoeren, Alex C.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Bibliography

[1] S. Adabala, V. Chadha, P. Chawla, R. Figueiredo, J. Fortes, I. Krsul, A. Matsunaga, M. Tsugawa, J. Zhang, M. Zhao, L. Zhu, and X. Zhu. From Virtualized Resources to Virtual Computing Grids: The In-VIGO system. *Future Generation Computing Systems (FGCS)*, 21(6), 2005.

[2] J. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, and A. Vahdat. Remote Control: Distributed Application Configuration, Management, and Visualization with Plush. In *Proceedings of the Large Installation System Administration Conference (LISA)*, 2007. In Submission.

[3] J. Albrecht, D. Oppenheimer, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. *ACM Transactions on Internet Technology (TOIT)*, 8(2), 2008.

[4] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Loose Synchronization for Large-Scale Networked Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2006.

[5] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. PlanetLab Application Management Using Plush. *ACM Operating Systems Review (OSR)*, 40(1), 2006.

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215, 1990.

[7] D. G. Andersen, H. Balakrishnan, and F. Kaashoek. Improving Web Availability for Clients with MONET. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[8] Appistry. `http://www.appistry.com/`.

[9] G. Audin. Reality Check On Five-Nines. *Business Communications Review*, May 19, 2002.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2003.

[11] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating Systems Support for Planetary-Scale Network Services. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[12] F. Berman, H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, B. Liu, X. Liu, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, C. Mendes, A. Olugbile, M. Patel, D. Reed, Z. Shi, O. Sievert, H. Xia, and A. YarKhan. New Grid Scheduling and Rescheduling Methods in the GrADS Project. *International Journal of Parallel Programming (IJPP)*, 33(2-3), 2005.

[13] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE Computer Conference (COMPCON)*, 1993.

[14] K. Birman. Replication and Fault-Tolerance in the ISIS System. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1985.

[15] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the Association for Computing Machinery (CACM)*, 36(12), 1993.

[16] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1987.

[17] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, 1995.

[18] A. Bricker, M. Litzkow, and M. Livny. Condor Technical Summary. Technical Report 1069, University of Wisconsin–Madison, CS Department, 1991.

[19] M. Burgess. Cfengine: A Site Configuration Engine. *USENIX Computing Systems*, 8(3), 1995.

[20] C. Catlett. The Philosophy of TeraGrid: Building an Open, Extensible, Distributed TeraScale Facility. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2002.

[21] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The Collective: A Cache-Based System Management Architecture. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[22] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *Proceedings of the IEEE Symposium on High Performance Distributed Compuuting (HPDC)*, 2003.

[23] B. Chun. gexec. `http://www.theether.org/gexec/`.

[24] B. Chun. pssh. `http://www.theether.org/pssh/`.

[25] B. Chun and T. Spalink. Slice Creation and Management. Technical Report PDN–03–013, PlanetLab Consortium, 2003.

[26] J. Coa, S. Jarvis, S. Saini, and G. Nudd. GridFlow: Workflow Managament for Grid Computing. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2003.

[27] Computing TCP's Retransmission Timers (RFC). `http://www.faqs.org/rfcs/rfc2988.html`.

[28] F. Cristian. Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems. *Distributed Computing (DC)*, 4(4), 1991.

[29] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters . In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[30] E. Dijkstra. The Structure of the "THE"-Multiprogramming System. *Communications of the Association for Computing Machinery (CACM)*, 11(5), 1968.

[31] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the ACM SIGMETRICS Conference (SIGMETRICS)*, 1996.

[32] EMAN, 2005. `http://ncmi.bcm.tmc.edu/EMAN/`.

[33] A. Emigh. Online Identity Theft: Phishing Technology, Chokepoints and Countermeasures. *Internet Theft Technology Council Report*, 2005.

[34] I. Foster. A Globus Toolkit Primer, 2005. `http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf`.

[35] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Global Grid Forum, 2002.

[36] A. Fox and E. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HotOS)*, 1999.

[37] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing Content Publication with Coral. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[38] G. A. Geist and V. S. Sunderam. Network-based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience (C: P&E)*, 4(4), 1992.

[39] GENI. `http://www.geni.net`.

[40] W. Gentzsch. Sun Grid Engine: Towards Creating A Compute Power Grid. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2001.

[41] Globus Toolkit Monitoring and Discovery System: MDS4. `http://www-unix.mcs.anl.gov/~schopf/Talks/mds4SC_nov2004.ppt`.

[42] R. Golding. A Weak-Consistency Architecture for Distributed Information Services. *Computing Systems*, 5(4), 1992.

[43] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. SmartFrog: Configuration and Automatic Ignition of Distributed Applications. In *HP Openview University Association Conference (HP OVUA)*, 2003.

[44] Google. `http://www.google.com`.

[45] L. Grit, D. Irwin, V. Marupadi, P. Shivam, A. Yumerefendi, J. Chase, and J. Albrecht. Harnessing Virtual Machine Resource Control for Job Management. In *Proceedings of the First Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, 2007.

[46] R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1989.

[47] R. Gupta and C. R. Hill. A Scalable Implementation of Barrier Synchronization Using an Adaptive Combining Tree. *International Journal of Parallel Programming (IJPP)*, 18(3), 1990.

[48] R. Huebsch. PlanetLab Application Manager. `http://appmanager.berkeley.intel-research.net`.

[49] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing Networked Resources with Brokered Leases. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2006.

[50] H. F. Jordan. A Special Purpose Architecture for Finite Element Analysis. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, 1978.

[51] F. Kaashoek, B. Liskov, D. Andersen, M. Dahlin, C. Ellis, S. Gribble, A. Joseph, H. Levy, A. Myers, J. Mogul, I. Stoica, and A. Vahdat. Report of the NSF Workshop on Research Challenges in Distributed Computer Systems. *GENI Design Document 05-06*, 2005.

[52] K. Keahey, K. Doering, and I. Foster. From Sandbox to Playground: Dynamic Virtual Environments in the Grid. In *Proceedings of the International Workshop in Grid Computing (Grid)*, 2004.

[53] Y.-S. Kee, D. Logothetis, R. Huang, H. Casanova, and A. Chien. Efficient Resource Description and High Quality Selection for Virtual Grids. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2005.

[54] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter USENIX Conference (USENIX)*, 1994.

[55] D. Kostić, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using Random Subsets to Build Scalable Network Services. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[56] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2003.

[57] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing. In *Proceedings of the Supercomputing Conference (SC)*, 2004.

[58] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the Association for Computing Machinery (CACM)*, 21(7), 1978.

[59] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong-Chan, S.-W. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing (JPDC)*, 33(2), 1996.

[60] J. Li, J. Stribling, R. Morris, M. F. Kaashoek, and T. M. Gil. A Performance vs. Cost Framework for Evaluating DHT Design Tradeoffs Under Churn. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2005.

[61] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1991.

[62] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 1988.

[63] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and Evaluation of a Resource Selection Framework. In *Proceedings of the IEEE Symposium on High Performance Distributed Compuuting (HPDC)*, 2002.

[64] Load Sharing Facility (LSF). `http://www.platform.com/Products/Platform.LSF.Family/`.

[65] B. Ludscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice and Experience, Special Issue on Scientific Workflows (CC: P&E)*, 18(10), 2005.

[66] S. Ludtke, P. Baldwin, and W. Chiu. EMAN: Semiautomated Software for High-Resolution Single-Particle Reconstructions. *Journal of Structural Biology*, 122, 1999.

[67] Y. Mao. vxargs. `http://dharma.cis.upenn.edu/planetlab/vxargs/`.

[68] J. Markoff and S. Hansell. Hiding in Plain Sight, Google Seeks More Power. *New York Times*, June 14, 2006.

[69] M. Massie, B. Chun, and D. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7), 2004.

[70] Maui. `http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php`.

[71] M. McNett. Usher. `http://usher.ucsd.edu/`.

[72] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee, Knoxville, 1994.

[73] S. Moh, C. Yu, B. Lee, H. Y. Youn, D. Han, and D. Lee. Four-Ary Tree-Based Barrier Synchronization for 2D Meshes without Nonmember Involvement. *IEEE Transactions on Computers (TC)*, 50(8), 2001.

[74] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the Association for Computing Machinery (CACM)*, 39(4), 1996.

[75] M. A. Nacar, M. Pierce, and G. C. Fox. Developing a Secure Grid Computing Environment Shell Engine: Containers and Services. *Neural, Parallel, and Scientific Computations (NPSC)*, 12, 2004.

[76] The Network Simulator. `http://www.isi.edu/nsnam/ns/`.

[77] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. In *Proceedings of the IEEE Symposium on High Performance Distributed Compuuting (HPDC)*, 2005.

[78] Opsware. `http://www.opsware.com/`.

[79] V. S. Pai, L. Wang, K. Park, R. Pang, and L. Peterson. The Dark Side of the Web: An Open Proxy's View. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2003.

[80] K. Park and V. S. Pai. Deploying Large File Transfer on an HTTP Content Distribution Network. In *Proceedings of the ACM/USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, 2004.

[81] K. Park and V. S. Pai. CoMon: A Mostly-Scalable Monitoring System for PlanetLab. *ACM Operating Systems Review (OSR)*, 40(1), 2006.

[82] L. Pearlman, C. Kesselman, S. Gullapalli, J. B.F. Spencer, J. Futrelle, K. Ricker, I. Foster, P. Hubbard, and C. Severance. Distributed Hybrid Earthquake Engineering Experiments: Experiences with a Ground-Shaking Grid Application. In *Proceedings of the IEEE Symposium on High Performance Distributed Compuuting (HPDC)*, 2004.

[83] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2002.

[84] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences Building PlanetLab. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[85] L. Peterson, J. Hartman, S. Muir, T. Roscoe, and M. Bowman. Evolving the Slice Abstraction. Technical Report PDN–04–017, PlanetLab Consortium, 2004.

[86] Plush. `http://plush.ucsd.edu`.

[87] Portable Batch Scheduler (PBS). `http://www.altair.com/software/pbspro.htm`.

[88] Ptolemy II. `http://ptolemy.eecs.berkeley.edu/ptolemyII/`.

[89] C. Pu and A. Leff. Epsilon-Serializability. Technical Report CUCS-054-90, Columbia University, 1991.

[90] R. Raman, M. Livny, and M. Solomon. Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. In *Proceedings of the IEEE Symposium on High Performance Distributed Compuuting (HPDC)*, 2003.

[91] S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the Embarrassing Slowness of OpenDHT on PlanetLab. In *Proceedings of the ACM/USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, 2005.

[92] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, 2005.

[93] R. Ricci, J. Duerig, P. Sanaga, D. Gebhardt, M. Hibler, K. Atkinson, J. Zhang, S. Kasera, and J. Lepreau. The Flexlab Approach to Realistic Evaluation of Networked Systems. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[94] M. Ripeanu, M. Bowman, J. S. Chase, I. Foster, and M. Milenkovic. Globus and PlanetLab Resource Management Solutions Compared. In *Proceedings of the IEEE Symposium on High Performance Distributed Compuuting (HPDC)*, 2004.

[95] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Communications of the Association for Computing Machinery (CACM)*, 17(7), 1974.

[96] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[97] A. Shoykhet, J. Lange, and P. Dinda. Virtuoso: A System for Virtual Machine Marketplaces. Technical Report NWU-CS-04-39, Northwestern University Department of Computer Science, 2004.

[98] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time, 2002. Proceedings of the USENIX Security Symposium (Security).

[99] J. Stribling. All-pairs pings for PlanetLab. `http://www.pdos.lcs.mit.edu/~strib/pl_app/`.

[100] SWORD. `http://sword.ucsd.edu`.

[101] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1995.

[102] F. Torres-Rojas, M. Ahamad, and M. Raynal. Timed Consistency for Shared Distributed Objects. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 1999.

[103] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[104] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems (TOCS)*, 21(2), 2003.

[105] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience (CC: P&E)*, 13(8-9), 2001.

[106] G. von Laszewski and M. Hategan. Java CoG Kit Karajan/GridAnt Workflow Guide. Technical report, Argonne National Laboratory, 2005.

[107] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[108] E. Walker, T. Minyard, and J. Boisseau. GridShell: A Login Shell for Orchestrating and Coordinating Applications in a Grid Enabled Environment. In *Proceedings of the International Conference on Computing, Communications and Control Technologies (CCCT)*, 2004.

[109] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2001.

[110] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[111] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[112] World Internet Usage Statistics News and Population Stats. `http://www.internetworldstats.com/stats.htm`.

[113] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, 2004.

[114] J.-S. Yang and C.-T. King. Designing Tree-Based Barrier Synchronization on 2D Mesh Networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 9(6), 1998.

[115] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2000.

[116] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing (JGC)*, 3(3-4), 2005.

[117] X. Zhang and J. Schopf. Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2. In *Proceedings of the International Workshop on Middleware Performance (MP)*, 2004.