Distributed Application Configuration, Management, and Visualization with Plush

6

JEANNIE ALBRECHT, Williams College
CHRISTOPHER TUTTLE, Google
RYAN BRAUD, Thousand Eyes
DARREN DAO, eHarmony
NIKOLAY TOPILSKI, Akamai Technologies
ALEX C. SNOEREN and AMIN VAHDAT, University of California, San Diego

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.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Design, Experimentation, Performance, Reliability

Additional Key Words and Phrases: Application management, PlanetLab

ACM Reference Format:
Albrecht, J., Tuttle, C., Braud, R., Dao, D., Topilski, N., Snoeren, A. C., and Vahdat, A. 2011. Distributed application configuration, management, and visualization with Plush. ACM Trans. Internet Technol. 11, 2, Article 6 (December 2011), 41 pages.
DOI = 10.1145/2049656.2049658 http://doi.acm.org/10.1145/2049656.2049658

1. INTRODUCTION

Many applications deployed on the Internet today run simultaneously on hundreds or thousands 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

This work was supported by the National Science Foundation under grants CNS-0834243 and CNS-0845349. C. Tuttle, R. Braud, D. Dao, and N. Topilski contributed to this project while attending graduate school at the University of California, San Diego.
Author's address: J. Albrecht, Williams College, Williamstown, MA; email: jeannie@cs.williams.edu.

DOI 10.1145/2049656.2049658 http://doi.acm.org/10.1145/2049656.2049658

are able to satisfy the ever-increasing demand of their users. Some of the most popular distributed applications, such as peer-to-peer services, Web search engines, and social networking sites use tens of thousands of computers (or more) to host their services and meet user demand [Markoff and Hansell 2006]. Additionally, other distributed applications, such as content distribution networks like CoDeeN [Pai et al. 2003] and Coral [Freedman et al. 2004], 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. 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 need mechanisms for detecting and recovering from the inevitable failures and problems endemic to distributed environments. To achieve availability and reliability, 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 reconfiguring 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 stand-alone applications in wide-area distributed environments take one of two management approaches. On wide-area live deployment platforms like PlanetLab [Bavier et al. 2004], 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 (such as the Globus Toolkit [Foster 2005]) 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 distributed emulation environments, the management approach largely depends on the platform. Some emulation environments such as ModelNet [Vahdat et al. 2002] provide toolkits and Web interfaces for manipulating applications, while others rely on researchers to write their own management scripts.

1.1. Contributions 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. To this end, we hypothesize that a unified set of abstractions (discussed in Section 2) 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 our work is to understand the abstractions and define the interfaces for specifying and managing distributed computations run in any execution environment. Through the design and implementation of an environment-independent toolkit for managing distributed applications, 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 [Ritchie and Thompson 1974] 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 data management.

As an evaluation of our hypothesis, we present Plush [Plush 2004; Albrecht et al. 2006b, 2007], 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 article we show how Plush manages applications that fall into three different classes: short-lived computations, long-lived services, and parallel grid applications. More specifically, we look at a case study from each of these classes and describe how Plush supports each one in an effort to provide evidence 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.

The remainder of this article is organized as follows. Section 2 identifies three classes of distributed applications and distills the five main abstractions required of any distributed application management infrastructure. Section 3 discusses how we design and implement the abstractions outlined in Section 2 in Plush. In Section 4, we highlight the details of the Plush resource matcher, which is responsible for providing

support for many different types of resources. Section 5 describes the design and implementation of a new synchronization primitive—a partial barrier—in Plush, while Section 6 evaluates Plush's ability to manage different types of applications. Lastly, Section 7 discusses related work, and Section 8 concludes.

2. ABSTRACTIONS 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 abstractions 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, the user first obtains access to machines capable of hosting the application. On PlanetLab, this involves creating a PlanetLab account or *slice* and uploading an SSH key pair that will be used for authentication on PlanetLab resources [Bavier et al. 2004]. When running a short-lived application, most users strive to find powerful machines with good connectivity at the time when the application is started. Resource discovery tools like SWORD [Albrecht et al. 2008] are commonly used to help find desirable machines. 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. An example of a short-lived computation could involve the evaluation of a new peer-to-peer file-distribution protocol.

2.1.2. Long-Lived Internet Services. 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 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. Since operators generally do not closely monitor long running services, failure detection and recovery must be automated. The environments in which services run also 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 [Pai et al. 2003] and Coral [Freedman et al. 2004].

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 on PlanetLab involves gathering data from specific sites and then processing this data using a compute-intensive algorithm to produce the desired

result.[1] Unlike interactive short-lived computations that often embrace 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 [Ripeanu et al. 2004]. 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, even if this means replacing slow machines in the middle of a computation. An example grid application is EMAN [Ludtke et al. 1999], which is a software package used for reconstructing 3-D models of particles using 2-D electron micrographs.

2.2. Required Application Management Abstractions

Though the low-level details for managing these different types of 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 distributed applications and build an application control infrastructure that supports all applications and execution environments. Providing a "one-size-fits-all solution" to application management runs the risk of introducing limitations that are not present in toolkits that are designed for specific platforms and applications. However, these toolkits also make it difficult to transition from one platform or application to another, which is especially problematic for researchers who want to make use of multiple deployment environments. We now extract the general requirements for a general distributed application management infrastructure. Together, these requirements identify the five key abstractions needed for defining and managing the flow of control for any distributed application.

2.2.1. Application Description. A distributed application controller must allow the user to customize the flow of control for each application. For example, the user must specify the resources (i.e., machines) required and any necessary credentials, the software needed to run the application (and instructions for how to install it), the processes that run on each resource, and the environment-specific execution parameters. 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. To manage complex multiphased 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 application's flow of control.

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

[1]In reality, PlanetLab is typically not used for running parallel grid applications. Dedicated computational grids such as NEESgrid [Pearlman et al. 2004] and Teragrid [Catlett 2002] are often used instead. We use PlanetLab in this discussion to provide a better comparison to the previous two examples.

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, Creation, and Acquisition. 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.

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 execution environment, resource acquisition can take a number of forms. In best-effort computing environments (e.g., PlanetLab), for example, 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 by batch schedulers, resource acquisition may involve 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., authentication keys) required for access. If failures occur 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.

2.2.3. 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, and CoBlitz [Park and Pai 2004], 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 specification. 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 *synchronization* abstraction to guarantee that various phases of computation start at approximately the same time on all resources. Providing synchronization guarantees without sacrificing performance in distributed computing environments is challenging, especially in failure-prone and volatile large-scale networks.

2.2.4. Application Control and Maintenance. 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 the 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.

3. DESIGN AND IMPLEMENTATION OF PLUSH

We now describe Plush, an extensible distributed application controller that implements the key abstractions of large-scale distributed application management discussed in Section 2. This section focuses on the application specification, application deployment, and application maintenance abstractions. Due to their complexity, we address the remaining two abstractions, resources and synchronization, separately in the following sections. 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 to Plush *clients*. The controller, typically run from the user's workstation,

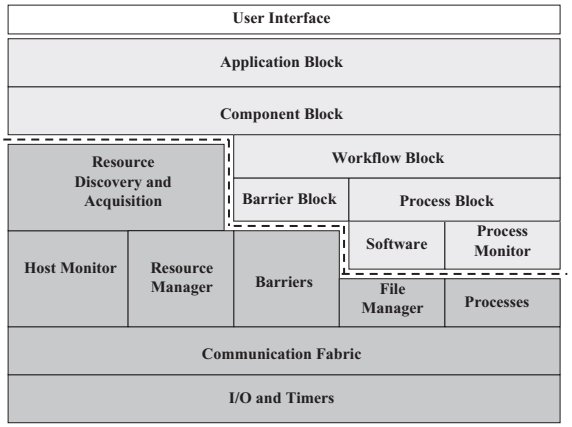


Fig. 1. The architecture of Plush. The *user interface* interacts with all boxes in the lower subsystems. Boxes below the user interface and above the dotted line are defined within the *application specification*. Boxes below the line represent the *core functional units*.

directs the flow of control throughout the life of the distributed application. The clients run on resources spread across the network and perform actions based on instructions received from the controller.

Figure 1 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 subsystems: 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. 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 section, we describe the design and implementation details of each of the Plush subsystems. (Note that the components within the subsystems are highlighted using italic font throughout the text in the remainder of this section.)

3.1. Application Specification

Developing a complete, yet accessible, application specification language was one of the principal challenges in this work. Our approach, which evolved over four years of experimentation and use, consists of combinations of five different "building block" abstractions for describing distributed applications. These blocks are represented by XML that is parsed by the Plush controller when the application is run. As a preview, Figure 2 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, runtime environment details, file and process I/O information, and the specific commands needed to start a process on a resource.

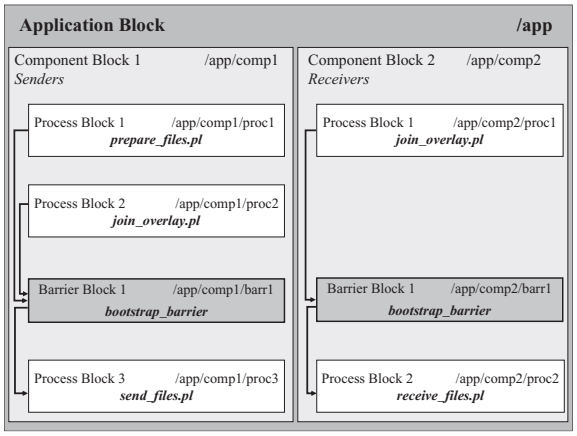


Fig. 2. Example file-distribution application comprised of application, component, process, and barrier blocks in Plush. Arrows indicate control-flow dependencies. (i.e., Block x → Block y implies that Block x must complete before Block y starts.)

(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. Workflow blocks are implemented using an internal barrier in Plush to keep track of task completion.
(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 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 2. 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 corresponding application specification for this file-distribution application contains one application block, which is used to define general characteristics about the application including the liveness properties and default failure recovery behavior. The

application block contains two component blocks that describe the groups of resources required to run the application. These component blocks run in parallel since there are no arrows indicating control-flow dependencies between them. Our application consists of component blocks that describe a set of senders and a set of receivers. Each component block defines the location and installation instructions for the required software and includes authentication information to access the resources. Specific resource requirements, such as processor speed or memory available, are also included in component blocks.

Within each component block, a combination of 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 also defined within the component blocks, and are used in applications where data files must be distributed and iteratively processed. The process blocks in our example describe the specific commands required to execute the application, including processes for file preparation, overlay membership, and file transfer details. 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. The barrier blocks in our example are used to separate the two phases of execution. Note that although each barrier block is uniquely defined within the component blocks, they 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.

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 1 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.

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. The Plush *resource matcher* then uses the resources in the resource pool to create a matching for the application. We discuss this process in detail in Section 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 1 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. Since traditional synchronization techniques are too strict for volatile, wide-area network conditions, Plush uses the relaxed synchronization semantics of partial barriers (described in detail in Section 5) for managing applications.

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. Once the resources are prepared with the necessary software, the application deployment phase completes by starting the execution. This is accomplished by starting processes on the resources. Plush *processes* are defined within process blocks in the application specification. A Plush process is a group of (identical) UNIX processes running across a distributed set of resources.

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 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.

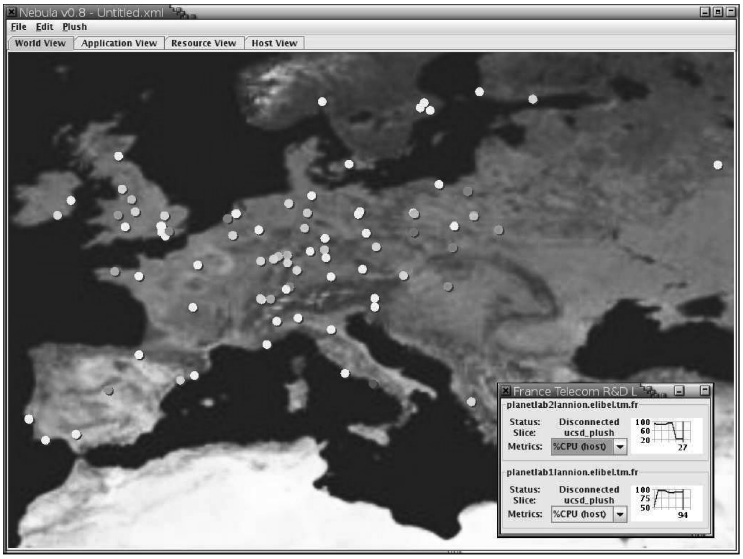
At the bottom of all of the other units is the Plush I/O and timer abstraction. As messages are received in the communication fabric, message handlers fire events. These events are sent to the I/O and timer layer and enter a queue. The event loop pulls events off the queue and calls the appropriate event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

3.3. Plush User Interfaces

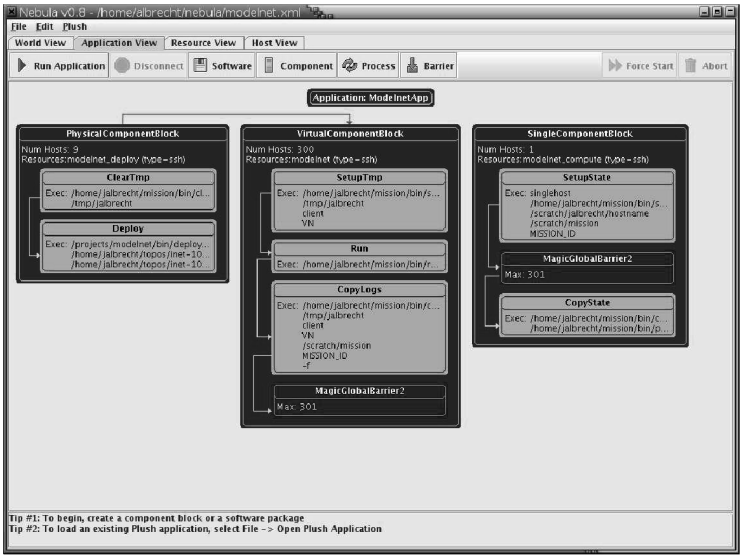
Plush aims to support a variety of applications being run by users with a wide range of expertise in building and managing distributed applications. Thus, Plush provides three interfaces which each provide users with techniques for interacting with their applications. In Figure 1, the user interface 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, creating a customizable application controller.

3.3.1. 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 (GUI) for Plush called Nebula [Nebula 2007]. In particular, we designed Nebula (as shown in Figure 3) 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.

The main Nebula window contains four tabs that show different information about the user's application. We show two of these tabs in Figure 3. 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, as shown in Figure 3(a). As the application proceeds through the different phases of execution, the sites change color, allowing the user to visualize the progress of their application. For example, a red dot on a site indicates failure while a green dot indicates correct 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(a). Note that, like Plush, Nebula is application-independent



(a) World View.



(b) Application View.

Fig. 3. Different tabs in Nebula. Additional images available at Nebula [2007].

and does not support application-specific monitoring. We are currently exploring ways to extend Nebula to allow more customized, application-specific monitoring and visualization.

The second tab in the Nebula main window is the "Application View." The Application View tab, shown in Figure 3(b), 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. After creating or loading an application specification, the

Table I. 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 control`	Print the controller's state information
`run`	Execute application (after loading specification)
`shell <quoted string>`	Run "quoted string" as a shell command on resources

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.

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.

The fourth tab in Nebula is called the "Host View" tab. This tab contains a table that displays the hostname of all available PlanetLab resources. 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. Right-clicking on a host in the Host View tab opens a new tab that contains an SSH connection directly to the host.

3.3.2. Command-Line Interface. Motivated by the popularity of the UNIX shell interface, Plush further streamlines the develop-deploy-debug cycle for distributed application management through a simple command-line interface where users can deploy, run, and debug their distributed applications running on hundreds of resources. The Plush command-line combines the functionality of a distributed shell with 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 I shows a subset of the available commands.

3.3.3. Programmatic Interface. Most 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. (Some of the common callback functions are shown in the bottom of Figure 4 in class `PlushXmlRpcCallback`.)

Figure 4 shows the Plush XML-RPC API.[2] The functions shown in the `PlushXmlRpcServer` class are available to users who wish to access Plush programmatically

[2]Note that the XML-RPC API provides a control plane interface only to the controller to integrate with third-party applications. The controller does not use XML-RPC to communicate with the clients in the data plane. Instead, the data plane uses an internal protocol running over TCP.

```
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 sendApplicationExit();
  void sendHostStatus(string host);
  void sendBlockStatus(string block);
  void sendResourceMatching(HashMap matching);
}
```

Fig. 4. Plush XML-RPC API.

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 Section 4 we will examine some specific uses of this API within the context of different resource management frameworks.

3.4. 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.

3.4.1. Fault Tolerance. To achieve 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 the possible failures that can occur, we discuss how Plush handles three common failure classes, namely, process, resource, and controller failures.

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. This model supports multithreaded processes, but not processes that fork other processes. Plush then 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 II summarizes the supported process exit policies.

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*

Table II. 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

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.

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.

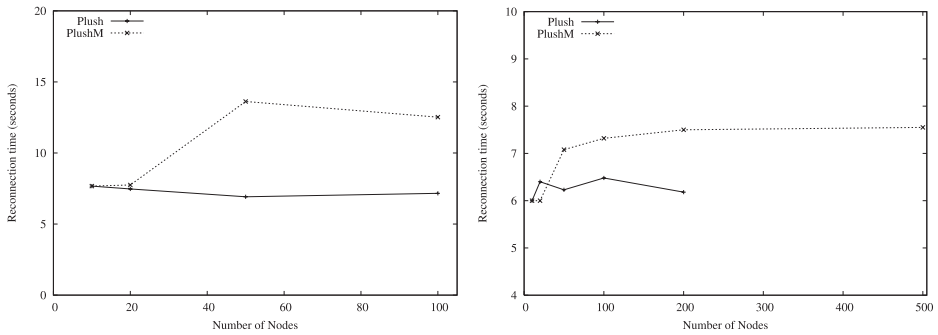
There are three possible actions in response to a resource failure: restart, rematch, and abort. By default, the controller tries all three actions in order. The first and easiest way to recover from a resource failure is to simply reconnect and restart the application on the failed resource. If the 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. 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.[3]

Controller failures. Because the controller is responsible for managing the flow of control across all clients, recovering from a failure at the controller is difficult. Plush uses 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 predetermined amount of time passes and the backup controllers do not receive any messages from the primary, it is assumed to have failed. The first backup becomes the primary, and execution continues.

This strategy has several potential drawbacks. First, it may cause extra messages to be sent over the network, which limits the scalability of Plush. Second, this approach may 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 may be difficult to reestablish consistency among all clients and resources. One solution to these problems is to separate the functionality of the controller from the maintenance of the underlying communication mesh. Thus controllers can fail, disconnect, and reconnect from the communication mesh without communicating with any nodes except for the "root" or "center" of the mesh.

3.4.2. Scalability. In addition to fault tolerance, an application controller designed for large-scale environments must scale to hundreds or even thousands of participants. Our experience in the design and implementation of Plush has shown that there is

[3]Some batch scheduling resource managers, such as LSF [Load Sharing Facility (LSF)] and SGE [Gentzsch 2001], provide similar mechanisms for coping with resource failures. However, Plush currently does not support interaction with these services.



(a) PlanetLab (b) ModelNet

Fig. 5. Average recovery time after failure of 25% of the overlay participants.

a trade-off between performance and scalability in this context. 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.

By default, all Plush clients connect directly to the controller forming a star topology. 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, we developed an extension to Plush called Plush-M that uses a random overlay tree (built using Mace [Killian et al. 2007]) for organization and communication [Topilski et al. 2008]. In an effort to reduce the number of hops between the clients and controller, Plush-M constructs "bushy" trees where the depth of the tree is small, and each node in the tree has many children. To make our tree more fault tolerant, the functionality of the controller is separated from the root of the overlay tree, which allows the overlay tree to be restructured quickly without changing the identity of the controller. In addition, by separating the controller from the root of the tree, it allows the controller to disconnect or go offline without impacting the execution of the application. While this improves scalability, it complicates failure recovery due to the potential overhead of tree reconfigurations. To quantify the overhead of reconfiguration, we ran experiments to measure the average recovery time after failure of Plush-M on ModelNet and PlanetLab. The results are shown in Figure 5.

Figure 5(a) shows the results from running our failure recovery experiment on PlanetLab. In this experiment we failed one quarter of the participants and measured the reconnection time. For Plush-M, the average reconnection time for overlays with more than 12 participants took approximately 12 seconds. When less than 12 nodes were involved, Plush-M created a tree with one level and all nodes directly linking to the root, and the average time was about 8 seconds. Plush running over its default star topology performed consistently as the number of nodes increased, with an average reconnect time of about 7 seconds. In Figure 5(b), we performed the same failure recovery experiment on ModelNet. Our results show that the reconnection latency difference between Plush-M and Plush is less than 2 seconds in most cases and can again be explained by the overhead associated with the tree reconfiguration

during node reconnections. Since network links were emulated and variable wide-area network conditions did not affect our results, the overhead of tree reconfiguration on ModelNet is much less significant than the overhead of tree configuration on PlanetLab. In general, we were satisfied with these results, and conclude that the overhead associated with tree reconfiguration is not significant, especially in underutilized networked environments.

3.5. Implementation Details

Plush is a publicly available software package [Plush 2004]. 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.

Nebula consists of approximately 25,000 lines of Java code. Nebula communicates with Plush using the XML-RPC interface described in Section 3.3.3. 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.

4. PLUSH RESOURCE MATCHER

In this section, we take a closer look at the resource abstraction in Plush. Recall that a resource in Plush is a (virtual or physical) machine that can host an application on behalf of the user. Section 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 employ instead an extensible resource discovery and acquisition unit (as shown in Figure 1) 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.

After constructing the resource pool, the algorithm used by the resource matcher is simple. Suppose the user wants to run an application on N resources. The matcher

```
<?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>
```

Fig. 6. Plush resource directory file.

then picks the first N resources from the resource pool and attempts to deploy the application. Depending on the target deployment environment, the resources may or may not exist in advance, and the resources may be ordered in some way. In this section, we examine the details pertaining to how the Plush resource matcher interacts with different types of resources provided by external services to construct a valid matching and run applications.

4.1. Plush Resource Pools

Before discussing how a matching is created, we first describe how resource pools are constructed in Plush. 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. Put simply, a resource pool is a grouping of resources that are available to the user and are reachable 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 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.

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. A sample directory.xml file is shown in Figure 6.

In addition to the resources defined in a directory file, resources are also added and removed by external services during an application's execution. This is accomplished using the Plush XML-RPC interface described in Section 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

```
<?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>
```

Fig. 7. Plush software and component definition.

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), 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 7 shows the Plush software and component definition that is part of the application specification. 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 file name the package is saved as on the resources.

The component definition begins below the software specification in Figure 7. 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. 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 previously 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 the appropriate number of node objects (as determined by "num_hosts") and inserts them into the matching. The Plush controller then begins to configure these chosen 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,[4] removes it from the matching, and inserts another randomly chosen resource from the resource pool. This process is repeated for each 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 some environments, 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. If a resource succeeds, the preference value is increased. 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 are more reliable, and thus are typically able to achieve better results.

4.3. PlanetLab Resource Selection

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. SWORD [Albrecht et al. 2008] is a publicly available service that is designed to perform resource discovery for PlanetLab. 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 allows users to rank the importance of various attributes. SWORD returns a list of resources organized by group that have the lowest overall penalty.

Although setting preferred hosts 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,

[4]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.

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. A full component definition that includes a SWORD query is shown in Figure 14 in Section 6.

The XML that appears between the "sword" tags in Figure 14 is a complete and unmodified SWORD query. 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.

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 [Irwin et al. 2006] and Usher [McNett et al. 2007]. Shirako 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 is a virtual-machine scheduling system for cluster environments. It allows users to create their own virtual machines or clusters. 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 to 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. As the Plush controller parses the application specification, it stores the resource description, and when the "plushCreateResources" command is issued (either via the command-line 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.

Similar to the way we included the SWORD query in the application specification in the preceding section, if a Plush user wants to obtain Shirako or Usher 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 and Usher, the attributes define the resources that will be created. Figure 8 shows an example of a Plush component definition that is augmented with a Shirako resource request. Shirako currently creates Xen [Barham et al.

```
<?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>
```

Fig. 8. Plush component definition containing Shirako resources.

2003] 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. 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.

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 [Vahdat et al. 2002] to the resource pool. Further, we describe how the Plush XML-RPC programmatic interface is used to perform job execution in a batch scheduler.

Mission is a simple batch scheduler that uses Plush to manage the execution of ModelNet jobs in our research cluster. 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

```
<?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>
```

Fig. 9. 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 queue and runs them on behalf of the user. This ensures that no two jobs are run simultaneously, and it also allows the resources to be shared more efficiently.

A Mission job submission has two components: a Plush application specification and a 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 configuration 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 9 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 ModelNet 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.

5. PARTIAL BARRIERS

This section discusses the wide-area distributed synchronization abstraction in Plush. Traditionally, synchronization barriers [Jordan 1978] 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

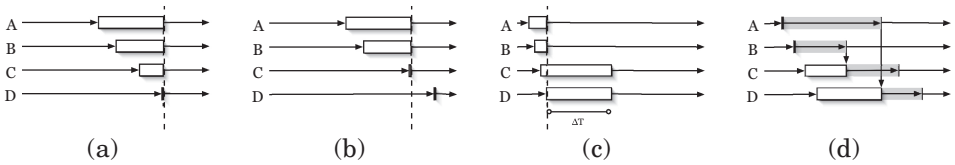


Fig. 10. (a) Traditional semantics: All hosts enter the barrier (indicated by the white boxes) and are simultaneously released (indicated by dotted line). (b) Early release: The barrier fires after 75% of the hosts arrive. (c) Throttled release: Hosts are released in pairs every ΔT seconds. (d) Counting semaphore: No more than two hosts are simultaneously allowed into a "critical section" (indicated by grey bars). When one host exits the critical section, another host enters.

liveness in failure-prone computing environments [Albrecht et al. 2006a]. 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 section, we describe how partial barriers have been integrated into Plush, and in Section 6 we evaluate the improved performance achieved using partial barriers.

5.1. Partial Barriers Semantics

Partial barriers are a set of semantic extensions to the traditional barrier synchronization abstraction. Specifically, we define two new barrier release rules that provide better support for applications that require synchronization in failure-prone wide-area computing environments. The new semantics are described as follows.

Early release. Traditional barriers require all nodes to enter a barrier before any node may pass through, as in Figure 10(a). A partial barrier with 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 10(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 10(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 [Dijkstra 1968] (shown in Figure 10(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 release 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, for example, 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

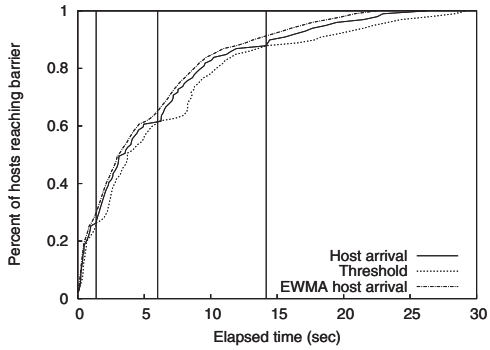


Fig. 11. Dynamically determining the knee of arriving processes. Vertical bars indicate a knee detection.

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.

5.2. Adaptive Release

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 for changing network conditions. Hence, we provide adaptive mechanisms in Plush to dynamically determine appropriate values during execution.

5.2.1. Early Release. There is a fundamental trade-off 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 it is too small, the application will lose the opportunity to have participation from other nodes had it just waited a bit longer. Thus, Plush dynamically determines 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 determine the knee offline once all of the data for an arrival process is available, it is difficult to determine this point online.[5]

The heuristic used in Plush, which is inspired by TCP retransmission timers and MONET [Andersen et al. 2005], 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 uses 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 11 illustrates how these values interact for a simulated group of 100 hosts entering a barrier with randomly generated

[5]Additional findings on algorithms for offline and online knee detection can be found in Satopää et al. [2011].

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 are met before firing.

5.2.2. Throttled Release. Adaptive methods are also used in Plush 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. 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. This technique increases the degree of concurrency whenever possible, but keeps the magnitude of each change small.

5.3. Partial Barriers in Plush

Partial barriers are part of the core design of Plush, enabling all applications 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 5.2 should be used. Plush itself uses partial barriers internally to separate different stages in an application's flow of control. In this section we evaluate the performance of partial barriers 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 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.

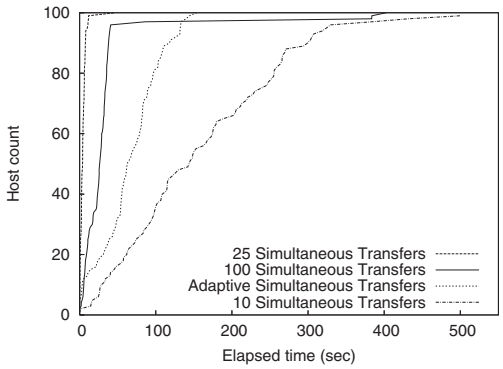


Fig. 12. Software transfer using a partial barrier to limit the number of simultaneous file transfers.

For our results, we measure the time it takes Plush to install the same 10MB file on 100 responsive and randomly chosen PlanetLab hosts while varying the number of simultaneous downloads using a semaphore barrier. Figure 12 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 10 and 100 simultaneous transfers. The "Adaptive Simultaneous Transfers" line in Figure 12 shows the performance of the Plush adaptive release technique as described in Section 5.2.2. 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 10 or 100, but the algorithm was too conservative to match the optimal static level of 25.

6. APPLICATION CASE STUDIES

The preceding sections explored the design and implementation of the Plush architecture. In this section, we revisit our initial design goals and take a closer look at how Plush supports different classes of applications.

6.1. Short-Lived Computation: Managing Bullet on PlanetLab

In Section 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 [Kostić et al. 2003], on PlanetLab. To maximize performance, 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.

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. Figure 13 illustrates a Bullet execution with one sender and two receivers.

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

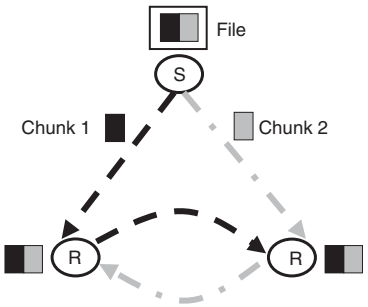


Fig. 13. Bullet execution with one sender (S) sending to two receivers (R).

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. A more desirable behavior in the Bullet initialization phase is to dynamically detect knees in the heavy-tailed join process. When 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.

Figure 14 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 using a "log_manager." After the process block, the XML specifies the "bullet_barrier" barrier block that separates application initialization from the data transfer phase. Since we modified the Bullet source code to interface directly with the Plush partial barrier API, nothing is defined after the barrier block in the application specification. However, a release of the startup barrier at the Bullet source host signals the end of application initialization, and thus begins the transfer of data.

6.1.1. 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 15 plots the cumulative distribution of receivers that enter the startup barrier on the y-axis as a function 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).

```
<?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>
```

Fig. 14. Bullet application specification.

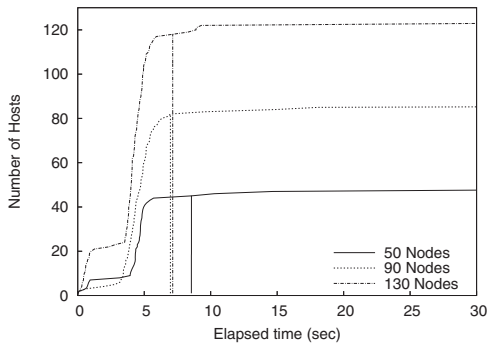


Fig. 15. A barrier regulating participants joining an overlay network in Bullet. Vertical bars indicate detected knees.

```
<?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>
```

Fig. 16. SWORD application specification.

Thus, manually choosing an appropriate static set may be sufficient for one particular batch of runs but not likely for the next. Vertical lines in Figure 15 indicate where the barrier manager detects a knee and releases the barrier. Notice that in all cases, the experiments proceed with 85–90% of the initial set participating, and wait no more than eight seconds to begin transmission.

6.2. Long-Lived Service: SWORD on PlanetLab

In Section 2 we described a long-running service as an application that 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 different types of failures due to network and host variability and volatility. We now 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.

SWORD is an example of a long-running PlanetLab service for resource discovery. SWORD stores data in a distributed hash table (DHT) and uses data from the DHT to respond to queries for groups of resources with specific characteristics. Distributed SWORD aims to run on as many PlanetLab hosts as possible, thus spreading the load of the system across many hosts allowing for increased scalability and also allowing SWORD to accurately respond to queries using information from a larger number of PlanetLab resources.

The XML application specification for SWORD is shown in Figure 16. 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 to run on as many nodes as possible, we specify a range of acceptable values rather than a

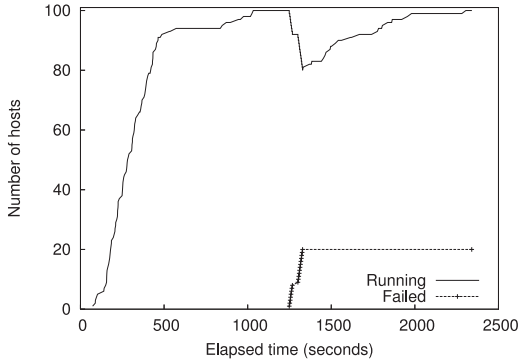


Fig. 17. Plush recovering from SWORD failures on PlanetLab. Service restored at $t = 2200$ seconds.

single number. Hence, as long as a minimum of 10 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. 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 except for 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 participants 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. 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. 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.

6.2.1. 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 17. 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 (38MB). It takes approximately 1000 seconds for all hosts to successfully download, install, and start SWORD. At time $t = 1250$s, 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 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 $t = 2200$s, the service is restored on 100 machines.

Using Plush to manage long-running services like SWORD relieves 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 Application: EMAN on PlanetLab

In this section we consider how Plush manages a typical grid application. Recall again from Section 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 one widely used grid application in this section, namely EMAN, and show how Plush manages the execution. Additionally, we show how partial barriers significantly improve the performance achieved across the wide-area.

To illustrate how Plush manages applications with workflows, we consider running EMAN [Ludtke et al. 1999] 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 18 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. 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 application 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 argument to specify the unique id of the task, which is then used to determine what fraction of

```
<?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>
```

Fig. 18. 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.

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 partial 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 240MB 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.

6.3.1. 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

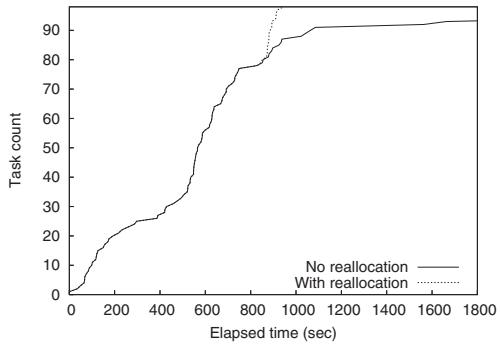


Fig. 19. EMAN. Knee detected at 801 seconds. Total runtime (without knee detection) is over 2700 seconds.

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 partial 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 40MB software archive containing the EMAN executables and a wrapper script, unpack the archive, download a unique 200MB 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 19 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 $t = 300$s and $t = 801$s. The first knee at $t = 300$s 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 $t = 801$s 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 19. The experiment on the original set of hosts continues past $t = 2700$s as indicated by the solid line in the graph, resulting in an overall speedup factor of more than three using partial semantics.

7. RELATED WORK

The functionality required by an application controller as discussed in this article 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. In addition, we also discuss related work that addresses workflow management, resource discovery, creation, acquisition, and synchronization since these are key components in distributed application management.

7.1. Remote Execution Tools

With respect to remote job execution, there are several tools available that provide a subset of the features that Plush supports, including cfengine [Burgess 1995], gexec [Chun], and vxargs [Mao]. The difference between Plush and these tools is that Plush provides more than just remote job execution. Plush also supports mechanisms for failure recovery and automatic reconfiguration due to changing conditions. In general, the pluggable aspect of Plush allows for the use of existing tools for actions like resource discovery and allocation, which provides more advanced functionality than most remote job execution tools.

From the user's point-of-view, the Plush command-line is similar to distributed shell systems such as GridShell [Walker et al. 2004] and GCEShell [Nacar et al. 2004]. These tools provide a user-friendly language abstraction layer that support script processing. Both tools are designed to work in Grid environments. Plush provides a similar functionality as GridShell and GCEShell, but unlike these tools, Plush works in a variety of environments.

7.2. Application Management Systems

In addition to remote job execution tools and distributed shells, projects like the PlanetLab Application Manager (appmanager) [Huebsch] and SmartFrog [Goldsack et al. 2003] focus specifically on managing distributed applications. appmanager is a tool for maintaining long running services and does not provide support for short-lived executions. SmartFrog [Goldsack et al. 2003] is a framework for describing, deploying, and controlling distributed applications. It consists of a collection of daemons that manage distributed applications and a description language to describe the applications. Unlike Plush, SmartFrog is a not a turnkey solution, but rather a framework for building configurable systems. Applications must adhere to a specific API to take advantage of SmartFrog's features.

The Grid community has several application management projects with goals similar to Plush, including Condor [Bricker et al. 1991] and GrADS/vGrADS [Berman et al. 2005]. Condor is a workload management system for compute-intensive jobs that is designed to deploy and manage distributed executions. Where Plush is designed to deploy and manage naturally distributed tasks with resources spread across several sites, Condor is optimized for leveraging underutilized cycles in desktop machines within an organization where each job is parallelizable and compute-bound. GrADS/vGrADS provides a set of programming tools and an execution environment for easing program development in computational grids. GrADS focuses specifically on applications where resource requirements change during execution. The task deployment process in GrADS is similar to Plush. Once the application starts execution, GrADS maintains resource requirements for compute-intensive scientific applications through a stop/migrate/restart cycle. Plush, on the other hand, supports a far broader range of recovery actions.

Lastly, the Globus Toolkit [Foster 2005] 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.

7.3. Workflow Management

Within the realm of workflow management, there are tools that provide more advanced functionality than Plush. For example, GridFlow [Coa et al. 2003], Kepler [Ludäscher et al. 2005], and the other tools described in Yu and Buyya [2005] are designed for advanced workflow management in Grid environments. The main difference between these tools and Plush is that they focus solely on workflow management schemes. Thus they are not well suited for managing applications that do not contain workflows, such as long-running services.

Workflow management systems like BOINC [Anderson 2004] are similar to Plush in that they aim to simplify tasks associated with the configuration, deployment, data distribution, and monitoring of public-resource distributed computing projects such as SETI@home. Unlike Plush, BOINC focuses on a much narrower type of application, that is, BOINC is designed for embarrassingly parallel computations that wish to make use of compute cycles donated from PCs worldwide. Plush supports a richer set of application control semantics where users can interact in real-time with their resources. Plush also supports multiple phases of execution and synchronization that is not easily achieved using platforms like BOINC.

7.4. Resource Discovery, Creation, and Acquisition

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, for example, resource discovery is accomplished using the Monitoring and Discovery Service [Zhang and Schopf 2004; Globus Toolkit Monitoring and Discovery System: MDS4]. The vGrADS project also has a tool for performing resource discovery and acquisition called vgFAB [Kee et al. 2005]. In Condor, applications and resource providers use a resource specification language called ClassAds [Litzkow et al. 1988], and Condor's matchmaker matches resource advertisements with requests using mechanisms called GangMatching [Raman et al. 2003] and SetMatching [Liu et al. 2002]. In addition, many grid environments rely on batch schedulers such as Sun Grid Engine (SGE) [Gentzsch 2001], Portable Batch System (PBS) [Portable Batch Scheduler], Maui [Maui], and Load Sharing Facility (LSF) [Load Sharing Facility (LSF)] for resource scheduling.

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 [Albrecht et al. 2008; Oppenheimer et al. 2005] and CoMon [Park and Pai 2006] were developed to help users find resources that best meet the needs of their application given the current operating conditions. CoMon is a PlanetLab service that measures resource usage across all PlanetLab nodes and provides basic support for simple queries via a Web interface.

The increasing popularity of virtual machine technologies has led to the development of several projects that explore using virtual machines to host network applications. In addition to Shirako [Irwin et al. 2006] and Usher [McNett et al. 2007], which are described in Section 4, other similar projects related to virtual machine creation and management include Sandpiper [Wood et al. 2007], In-VIGO [Adabala et al. 2005], VMPlants [Krsul et al. 2004], The Collective [Chandra et al. 2005], Virtual Workspaces [Keahey et al. 2004], and Virtuoso [Shoykhet et al. 2004].

7.5. 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 [Jordan 1978]. Given the importance of fast primitives for coordinating bulk synchronous SIMD applications, most massively parallel processors (MPPs) have hardware support for barriers [Leiserson et al. 1996; Scott 1996]. 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 [Keleher et al. 1994; Bershad et al. 1993]. In addition to shared memory, another popular programming model for loosely synchronized parallel machines is message-passing. Popular message passing libraries such as PVM [Geist and Sunderam 1992] and MPI [Message Passing Interface Forum 1994] contain implementations of barriers as a fundamental synchronization service.

The loose synchronization model used by Plush for running applications in failure-prone environments is related in spirit to a variety of efforts in relaxed consistency models for updates in distributed systems, including Epsilon Serializability [Pu and Leff 1991], the CAP principle [Fox and Brewer 1999], Bayou [Terry et al. 1995], TACT [Yu and Vahdat 2000], and Delta Consistency [Torres-Rojas et al. 1999]. All of these projects recognize the need for relaxed semantics to cope with wide-area inconsistencies and volatility.

8. 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, and EMAN, 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. We have also incorporated the use of Plush in the classroom in upper-level undergraduate courses to help lower the barrier of entry to experimentation with distributed systems [Albrecht 2009]. Most users find Plush to be an extremely useful tool[6] 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

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

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 since 2004, 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.

8.1. From Plush to Gush

In 2008, Plush was incorporated into the GENI (Global Environment for Network Innovations) initiative [GENI 2008]. Rather than trying to keep Plush up to date with GENI APIs and non-GENI APIs, a new version of Plush was created for GENI called Gush. The architectural design of Gush is largely the same as Plush, but most of the mechanisms for interacting with resource discovery and acquisition services have been replaced with GENI-specific services. The resources supported in Gush include PlanetLab hosts [PlanetLab GENI 2008], ProtoGENI virtual machines [ProtoGENI 2008], and ORCA virtual machines [ORCA-BEN 2008]. The syntax of the configuration files have changed as well and additional functionality has been added to support the elaborate resource specifications used in GENI [Albrecht and Huang 2010]. More information about Gush is available on the Gush Web site [Gush 2008].

REFERENCES

Adabala, S., Chadha, V., Chawla, P., Figueiredo, R., Fortes, J., Krsul, I., Matsunaga, A., Tsugawa, M., Zhang, J., Zhao, M., Zhu, L., and Zhu, X. 2005. From virtualized resources to virtual computing grids: The In-VIGO system. *Future Gen. Comput. Syst. 21,* 6.

Albrecht, J. 2009. Bringing big systems to small schools: Distributed systems for undergraduates. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE)*.

Albrecht, J., Braud, R., Dao, D., Topilski, N., Tuttle, C., Snoeren, A. C., and Vahdat, A. 2007. Remote control: Distributed application configuration, management, and visualization with Plush. In *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*.

Albrecht, J. and Huang, D. Y. 2010. Managing distributed applications using Gush. In *Proceedings of the ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, Testbed Practices Session (TridentCom)*.

Albrecht, J., Oppenheimer, D., Patterson, D., and Vahdat, A. 2008. Design and implementation tradeoffs for wide-area resource discovery. *ACM Trans. Internet Technol. 8,* 4.

Albrecht, J., Tuttle, C., Snoeren, A. C., and Vahdat, A. 2006a. Loose synchronization for large-scale networked systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*.

Albrecht, J., Tuttle, C., Snoeren, A. C., and Vahdat, A. 2006b. PlanetLab application management using Plush. *ACM Operat. Syst. Rev. 40,* 1.

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

Anderson, D. P. 2004. BOINC: A System for public-resource computing and storage. In *Proceedings of the IEEE/ACM International Workshop on Grid Computing*.

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

BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., MUIR, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. 2004. Operating systems support for planetary-scale network services. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

BERMAN, F., CASANOVA, H., CHIEN, A., COOPER, K., DAIL, H., DASGUPTA, A., DENG, W., DONGARRA, J., JOHNSSON, L., KENNEDY, K., KOELBEL, C., LIU, B., LIU, X., MANDAL, A., MARIN, G., MAZINA, M., MELLOR-CRUMMEY, J., MENDES, C., OLUGBILE, A., PATEL, M., REED, D., SHI, Z., SIEVERT, O., XIA, H., AND YARKHAN, A. 2005. New grid scheduling and rescheduling methods in the GrADS project. *Inter. J. Parall. Program. 33,* 2–3.

BERSHAD, B., ZEKAUSKAS, M., AND SAWDON, W. 1993. The midway distributed shared memory system. In *Proceedings of the IEEE Computer Conference (COMPCON)*.

BRICKER, A., LITZKOW, M., AND LIVNY, M. 1991. Condor technical summary. Tech. rep. 1069, Computer Science Department, University of Wisconsin–Madison.

BURGESS, M. 1995. Cfengine: A site configuration engine. *USENIX Comput. Syst. 8,* 3.

CATLETT, C. 2002. 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)*.

CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., AND LAM, M. S. 2005. The collective: A cache-based system management architecture. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

CHUN, B. gexec. http://www.theether.org/gexec/.

COA, J., JARVIS, S., SAINI, S., AND NUDD, G. 2003. GridFlow: Workflow managament for grid computing. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*.

DIJKSTRA, E. 1968. The Structure of the "THE"-multiprogramming system. *Comm. ACM 11,* 5.

FOSTER, I. 2005. A globus toolkit primer. http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf.

FOX, A. AND BREWER, E. 1999. Harvest, yield, and scalable tolerant systems. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HotOS)*.

FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIÈRES, D. 2004. Democratizing content publication with Coral. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

GEIST, G. A. AND SUNDERAM, V. S. 1992. Network-based concurrent computing on the PVM system. *Concurrency: Pract. Exper. 4,* 4.

GENI 2008. http://www.geni.net.

GENTZSCH, W. 2001. Sun grid engine: Towards creating a compute power grid. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*.

GLOBUS TOOLKIT MONITORING AND DISCOVERY SYSTEM: MDS4. http://www-unix.mcs.anl.gov/~schopf/Talks/mds4SC_nov2004.ppt.

GOLDSACK, P., GUIJARRO, J., LAIN, A., MECHENEAU, G., MURRAY, P., AND TOFT, P. 2003. SmartFrog: Configuration and automatic ignition of distributed applications. In *Proceedings of the HP Openview University Association Conference (HP OVUA)*.

GUSH 2008. http://gush.cs.williams.edu/.

HUEBSCH, R. PlanetLab application manager. http://appmanager.berkeley.intel-research.net.

IRWIN, D., CHASE, J., GRIT, L., YUMEREFENDI, A., BECKER, D., AND YOCUM, K. G. 2006. Sharing networked resources with brokered leases. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*.

JORDAN, H. F. 1978. A special purpose architecture for finite element analysis. In *Proceedings of the International Conference on Parallel Processing (ICPP)*.

KEAHEY, K., DOERING, K., AND FOSTER, I. 2004. From sandbox to playground: Dynamic virtual environments in the grid. In *Proceedings of the International Workshop in Grid Computing (Grid)*.

KEE, Y.-S., LOGOTHETIS, D., HUANG, R., CASANOVA, H., AND CHIEN, A. 2005. Efficient resource description and high quality selection for virtual grids. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*.

KELEHER, P., DWARKADAS, S., COX, A. L., AND ZWAENEPOEL, W. 1994. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter USENIX Conference (USENIX)*.

KILLIAN, C., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. 2007. Mace: Language support for building distributed systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

KOSTIĆ, D., RODRIGUEZ, A., ALBRECHT, J., AND VAHDAT, A. 2003. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*.

KRSUL, I., GANGULY, A., ZHANG, J., FORTES, J. A. B., AND FIGUEIREDO, R. J. 2004. VMPlants: Providing and managing virtual machine execution environments for grid computing. In *Proceedings of the Supercomputing Conference (SC)*.

LEISERSON, C. E., ABUHAMDEH, Z. S., DOUGLAS, D. C., FEYNMAN, C. R., GANMUKHI, M. N., HILL, J. V., HILLIS, W. D., KUSZMAUL, B. C., PIERRE, M. A. S., WELLS, D. S., WONG-CHAN, M. C., YANG, S.-W., AND ZAK, R. 1996. The network architecture of the connection machine CM-5. *J. Parall. Distrib. Comput. 33,* 2.

LITZKOW, M., LIVNY, M., AND MUTKA, M. 1988. Condor—A hunter of idle workstations. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*.

LIU, C., YANG, L., FOSTER, I., AND ANGULO, D. 2002. Design and evaluation of a resource selection framework. In *Proceedings of the IEEE Symposium on High Performance Distributed Compuuting (HPDC)*.

LOAD SHARING FACILITY (LSF). http://www.platform.com/Products/Platform.LSF.Family/.

LUDTKE, S., BALDWIN, P., AND CHIU, W. 1999. EMAN: Semiautomated software for high-resolution single-particle reconstructions. *J. Struct. Biol. 122*.

LUDÄSCHER, B., ALTINTAS, I., BERKLEY, C., HIGGINS, D., JAEGER-FRANK, E., JONES, M., LEE, E. A., TAO, J., AND ZHAO, Y. 2005. Scientific workflow management and the Kepler system. *Concurrency Computat. Pract. Exper.* (Special Issue on Scientific Workflows) *18,* 10.

MAO, Y. vxargs. http://dharma.cis.upenn.edu/planetlab/vxargs/.

MARKOFF, J. AND HANSELL, S. 2006. Hiding in plain sight, Google seeks more power. *New York Times*.

MAUI. Maui. http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php.

MCNETT, M., GUPTA, D., VAHDAT, A., AND VOELKER, G. M. 2007. Usher: An extensible framework for managing clusters of virtual machines. In *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*.

MESSAGE PASSING INTERFACE FORUM. 1994. MPI: A message-passing interface standard. Tech. rep. UT-CS-94-230, University of Tennessee, Knoxville.

NACAR, M. A., PIERCE, M., AND FOX, G. C. 2004. Developing a secure grid computing environment shell engine: Containers and services. *Neural Parall. Scientific Computat. 12*.

NEBULA 2007. http://plush.cs.williams.edu/nebula.

OPPENHEIMER, D., ALBRECHT, J., PATTERSON, D., AND VAHDAT, A. 2005. Design and implementation tradeoffs for wide-area resource discovery. In *Proceedings of the IEEE Symposium on High Performance Distributed Compuuting (HPDC)*.

ORCA-BEN 2008. https://ben.renci.org/.

PAI, V. S., WANG, L., PARK, K., PANG, R., AND PETERSON, L. 2003. The dark side of the Web: An open proxy's view. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*.

PARK, K. AND PAI, V. S. 2004. Deploying large file transfer on an HTTP content distribution network. In *Proceedings of the ACM/USENIX Workshop on Real, Large Distributed Systems (WORLDS)*.

PARK, K. AND PAI, V. S. 2006. CoMon: A mostly-scalable monitoring system for PlanetLab. *ACM Operat. Syst. Rev. 40,* 1.

PEARLMAN, L., KESSELMAN, C., GULLAPALLI, S., B.F. SPENCER, J., FUTRELLE, J., RICKER, K., FOSTER, I., HUBBARD, P., AND SEVERANCE, C. 2004. Distributed hybrid earthquake engineering experiments: Experiences with a ground-shaking grid application. In *Proceedings of the IEEE Symposium on High Performance Distributed Compuuting (HPDC)*.

PLANETLAB GENI 2008. http://groups.geni.net/geni/wiki/PlanetLab.

PLUSH 2004. Plush. http://plush.cs.williams.edu.

PORTABLE BATCH SCHEDULER. (PBS). http://www.altair.com/software/pbspro.htm.

PROTOGENI 2008. http://www.protogeni.net/.

PU, C. AND LEFF, A. 1991. Epsilon-serializability. Tech. rep. CUCS-054-90, Columbia University.

RAMAN, R., LIVNY, M., AND SOLOMON, M. 2003. Policy driven heterogeneous resource co-allocation with gangmatching. In *Proceedings of the IEEE Symposium on High Performance Distributed Compuuting (HPDC)*.

RIPEANU, M., BOWMAN, M., CHASE, J. S., FOSTER, I., AND MILENKOVIC, M. 2004. Globus and PlanetLab resource management solutions compared. In *Proceedings of the IEEE Symposium on High Performance Distributed Compuuting (HPDC)*.

RITCHIE, D. M. AND THOMPSON, K. 1974. The UNIX Time-sharing system. *Comm. ACM 17,* 7.

SATOPÄÄ, V., ALBRECHT, J., IRWIN, D., AND RAGHAVAN, B. 2011. Finding a "kneedle" in a haystack: Detecting knee points in system behavior. In *Proceedings of the IEEE Workshop on Simplifying Complex Networks for Practitioners (Simplex)*.

SCOTT, S. L. 1996. Synchronization and communication in the T3E multiprocessor. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

SHOYKHET, A., LANGE, J., AND DINDA, P. 2004. Virtuoso: A system for virtual machine marketplaces. Tech. rep. NWU-CS-04-39, Department of Computer Science, Northwestern University.

TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*.

TOPILSKI, N., ALBRECHT, J., AND VAHDAT, A. 2008. Improving scalability and fault tolerance in an application management infrastructure. In *Proceedings of the USENIX Workshop on Large-Scale Computing (LASCO)*.

TORRES-ROJAS, F., AHAMAD, M., AND RAYNAL, M. 1999. Timed consistency for shared distributed objects. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*.

VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIĆ, D., CHASE, J., AND BECKER, D. 2002. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*.

WALKER, E., MINYARD, T., AND BOISSEAU, J. 2004. 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)*.

WOOD, T., SHENOY, P., VENKATARAMANI, A., AND YOUSIF, M. 2007. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

YU, H. AND VAHDAT, A. 2000. 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)*.

YU, J. AND BUYYA, R. 2005. A taxonomy of workflow management systems for grid computing. *J. Grid Computing 3,* 3–4.

ZHANG, X. AND SCHOPF, J. 2004. Performance analysis of the Globus toolkit monitoring and discovery service, MDS2. In *Proceedings of the International Workshop on Middleware Performance (MP)*.

Received January 2009; revised July 2010; accepted May 2011