PlanetLab Application Management Using Plush

Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat
University of California, San Diego
{*jalbrecht, ctuttle, snoeren, vahdat*}@*cs.ucsd.edu*

Abstract

Support for application deployment and monitoring in large-scale distributed systems such as PlanetLab remains in its early stages. While a number of solutions exist for specific subtasks of deployment and monitoring, these tools suffer from a lack of integration. Most tools were developed specifically to deploy and manage a particular service or application on a single platform and were not designed to be general enough to support different environments. In this paper, we consider three different classes of PlanetLab applications to distill a set of requirements for a general application-control infrastructure. We then discuss initial experiences and lessons learned during the development and PlanetLab deployment of Plush, a tool designed to manage applications running over large-scale distributed systems.

1 Introduction

Emerging distributed computation infrastructures such as PlanetLab [2, 18] and the Grid [8] hope to support applications that simultaneously run on hundreds or even thousands of heterogeneous physical machines distributed across the Internet. Today, however, running even simple jobs across such infrastructures is typically a cumbersome, manual, and error-prone process. A number of tasks must be completed before starting an application, including resource discovery, resource allocation, file distribution, and environment configuration. Finally, once the application starts up, its execution must be carefully monitored and controlled. Our experience indicates that researchers regularly expend a great deal of time and effort deploying and managing their applications, considerably complicating the end goal of conducting research experiments or maintaining an Internet service. While a number of tools exist to independently address some of the challenges, their overall utility is limited by their lack of integration.

There are two options for running individual applications across heterogeneous distributed environments like PlanetLab. Currently, a large number of PlanetLab applications address deployment and monitoring in an *ad hoc*, application-specific fashion. Such custom implementations can perfectly match each application's semantics and requirements while providing high performance in certain environments. The major challenge, of course, is retooling the infrastructure when application requirements or PlanetLab conditions change. A second, alternate approach develops a runtime environment that exports a common set of abstractions to service the requirements of a broad range of applications.

One benefit of the latter approach is masking the often significant complexity associated with executing, configuring, and managing large-scale distributed computations from end users who would otherwise have to relearn the black art of reliably deploying and maintaining networked systems across asynchronous and failure-prone distributed environments. A second benefit is separating the specification of a distributed computation—a high-level description of what resources a particular application requires or desires, how the application should react to failures, and its individual phases of computation—from the application logic that actually implements the computation. In this manner, we can avoid tools that "hardwire" knowledge of a particular distributed environment's configuration such as the characteristics of individual hosts or network links. These attributes inherently change over time, resulting in brittle tools.

While both approaches have merit in different scenarios and settings, in this article we explore the benefits of the second approach. We present Plush, a framework of components that, when taken together, provide a unified environment to support the distributed application design and deployment life cycle. Plush users describe distributed applications using an extensible XML specification language. The language allows users to customize various aspects of the deployment life cycle to fit the needs of an application. This functionality can be used, for example, to specify a specific resource discovery or allocation tool to use during application deployment.

Once an application is up and running, Plush monitors it for failures or application-level errors for the duration of its execution. Upon detecting a problem, Plush can perform a number of user-configurable actions, such as restarting the application, automatically reconfiguring it, or even searching for alternate resources. For applications requiring wide-area synchronization, Plush provides a number of efficient synchronization primitives. In particular, Plush provides two new barrier semantics, which relax traditional barrier semantics for increased performance and robustness in failure-prone environments.

The remainder of this article reports our experiences designing and building Plush. Section 2 describes three common classes of PlanetLab applications. Motivated by these classes, Section 3 enumerates requirements for a general-purpose application control infrastructure. Section 4 describes the architecture of Plush in more detail. In Section 5, we outline some of the important lessons we have learned during the development of Plush. We discuss related work in Section 6 before concluding in Section 7.

2 Usage scenarios

We begin by exploring three different classes of applications that often run on PlanetLab: a short-lived distributed application, a continuously running Internet service, and a Grid-style parallel application.

2.1 Short-lived applications

One of the most common uses of PlanetLab is to interactively execute short distributed computations. Applications range from the simple, where a novice wishes to gain experience with PlanetLab applications, to the complex, where experienced users wish to test new protocols. Short-lived applications like these typically run for a few days or less and are closely monitored by the user. For example, suppose a user wants to test a file-distribution protocol on 50 PlanetLab nodes scattered around the world. The user would have to gain access to PlanetLab, find 50 machines capable of running the application, install the software on those 50 machines, run the application, and collect any output files produced for analysis. We examine this process in more detail.

To then test this new protocol on PlanetLab, the user must first gain access to PlanetLab resources. Authentication on PlanetLab is based on public-key cryptography, and access to PlanetLab machines is achieved through SSH login using RSA authentication. The user must register with PlanetLab Central (PLC) to obtain a user account and create an SSH key pair. Once the user uploads a public key to the PLC database, she can associate her account with a PlanetLab slice. A slice is a named set of distributed PlanetLab resources. It forms the basis for both resource allocation and isolation. The user binds the slice to a number of physical machines (e.g., using a Web interface), which causes the user's public key to be copied to the nodes and the user to be authorized for login to the machines.

The next step is to find a suitable set of machines. Resource discovery tools like SWORD [15] can be used to help streamline the process. For our example, the user may issue a SWORD query for 50 machines with fast processors, large amounts of free memory, and high pairwise bandwidth. Once a set of machines has been identified, the user must transfer any required software to the 50 machines using a file transfer protocol such as scp, Bullet [14], or CoBlitz [17]. The executable must then be started on all 50 machines at approximately the same time. This is accomplished by connecting to each machine separately via SSH and then executing the appropriate command. Typically, once the execution completes, a set of output files must then be copied to a central location for analysis.

2.2 Long-lived Internet services

Along with short computations, PlanetLab is also used to deploy services that run continuously. Current PlanetLab services include CoDeen [16], Coral [9], and OpenDHT [20], among others. Long-running services must be robust to a variety of failures that short-lived applications generally do not encounter. Hence, in addition to the tasks described above for short-lived applications, users who manage services must perform additional tasks to maintain the service over an extended period of time. These network services are expected to run for several months or longer and are often not monitored as closely as those with shorter lifetimes. Further, since these applications provide a service to others, availability is critical.

Suppose a user wants to deploy a new resource discovery service on PlanetLab. The service aims to run on as many PlanetLab machines as possible to provide accurate information to users. To deploy such a service, a user must go through the same process as described in the previous section for establishing authentication, adding nodes to a slice, finding a suitable set of resources, transferring software, and starting the executable. However, users running short-lived applications often pick powerful machines that have good connectivity, *i.e.*, low-latency and high-bandwidth connections. When running a service on a larger number of machines, a user is subjected to slow or lossy connections in addition to more desirable low-latency, high-bandwidth links. Further, some machines may have slower processors and less free memory. Thus, choosing nodes to host an Internet service often hinges on avoiding nodes that frequently perform poorly over relatively long time periods rather than choosing nodes that perform well at any given point in time [19].

Once the service is running, the user monitors her processes for failures. When running short-lived applications, users often treat a failure as an aberrant condition and discard the results of the run. For long-running services, failures are the rule rather than the exception and, therefore, must be addressed as such. Thus, if a failure does occur, the user attempts to restore the service as quickly as possible to maintain high availability.

2.3 Grid-style parallel applications

Though there are many different types of Grid-style applications, one of the most common usage scenarios is harnessing resources at one or more sites to execute a computationally intensive job. A typical Grid application often involves gathering data from specific sites, and then processing this data in a compute-intensive application to produce the desired result. Many Grid applications are 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. For a researcher running a Grid application, choosing the appropriate set of machines is crucial to achieving good throughput. Experience shows this can be quite difficult on PlanetLab.

For example, suppose a physicist wants to run EMAN [6] on PlanetLab. EMAN is a publicly-available software package used for reconstructing 3-D models of particles using 2-D electron micrographs. The program takes a 2-D micrograph image as input and then runs a "refinement" process on the image to create a 3-D model. The refinement process is

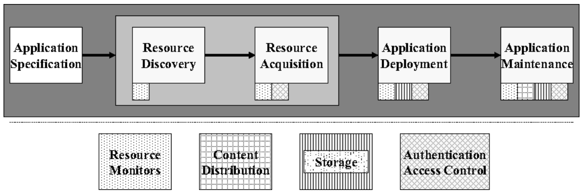

Figure 1: Basic requirements and flow of control for a distributed application control infrastructure.

run repeatedly until yielding a result with the desired quality. Each iteration of refinement consists of both computationally inexpensive sequential computations and computationally expensive parallel computations. For multiple iterations of refinement, the entire cycle is repeated.

As in the other applications, the researcher running EMAN has to gain access to PlanetLab, find suitable resources, distribute the software and data files, install the software, and start the executable. Unlike the short-lived application or the long-running service described above, however, the performance of EMAN is greatly affected by the computational resources available on the machines hosting the parallel computations. Thus, with each iteration of the refinement process, the researcher running the application wants to use the set of machines that has the most available computational resources. Further, if a machine fails or suddenly becomes overloaded during the refinement process, the machine should be replaced by another with more available resources. In parallel applications such as EMAN, the rate of completion for individual tasks is often delayed by a few slow machines or processors. Detecting and recovering from these bottlenecks is both difficult and essential to achieve high performance in parallel applications.

3 Requirements

In the previous section we investigated the process of executing three different classes of distributed applications. Though the low level details for managing the applications were different, at a high level the requirements for each example were largely similar. Rather than reinvent the same infrastructure for each application separately, we set out to identify commonalities across all three classes of distributed applications, and build an application control infrastructure that supports all three types of applications and environments. Based on the discussion in the previous section, we now extract some general requirements for a distributed application control infrastructure. Together, these requirements define the typical flow of control for any distributed application, as shown in Figure 1.

Application Specification. The application specification identifies all aspects of the execution and environment needed by the application control infrastructure to successfully deploy, manage, and maintain the application. It describes the software required to run the application—including how to access and install it—and processes that will run on each machine. To support a variety of environments, the user specifies these details using an extensible description language that captures desired resource specifications, declares how they should be acquired, and includes any additional information needed to correctly instantiate and run the application. Any information required for authentication is also included in this description.

Resource Discovery and Acquisition. The first step to successfully running any distributed application is obtaining a suitable set of resources on which to run. Because resources in distributed environments are often heterogeneous, users naturally want to find the set of resources that will best satisfy the requirements of their applications. Even on PlanetLab, where the hardware is largely homogeneous, dynamic characteristics of a node such as available bandwidth, CPU load, *etc.*, vary greatly over time. The goal of resource discovery is to find the best *current* set of physical resources for the distributed application as specified by the user.

The role of the application control infrastructure is to parse the user's request for resources and send the request to an appropriate resource discovery mechanism. The resource discovery mechanism interacts directly with the resource acquisition system. Resource acquisition can be accomplished in a number of ways. For example, if resource reservations are required, the resource acquisition mechanism is responsible for submitting a resource request on the user's behalf and subsequently obtaining a usage lease. Currently, the machines in PlanetLab are in a "best effort" pool, which means that no advanced reservations are required for use. Therefore no further steps for acquisition are typically needed, but supplemental requests can be issued to systems like Bellagio [1] or Sirius [21] if desired.

Application Deployment. Once a set of resources have been located, the next step required in most scenarios is deployment. The process of application deployment involves preparing the physical resources with the correct software and data files, and then running the executable to start the application. This typically involves copying, unpacking, and installing the software on the hosts that were selected in the resource discovery and acquisition process. An application control infrastructure must handle a variety of different file transfer protocols for each environment, and must provide support for failures that occur during the transfer of software or the starting of the executable.

Application Maintenance. Perhaps the most difficult requirement of the application control infrastructure is monitoring an application after it has been started. Monitoring involves probing the hosts for failure due to network outages or hardware malfunctions, querying the application for indications of failure during execution, and providing hooks into application-specific code for observing the progress of an execution. The goal of application maintenance is to maintain application liveness, provide detailed error information, and achieve forward progress in the face of failures. A robust application control infrastructure must be able to adapt to "less-than-perfect" conditions and continue execution. For example, if a user wants to use 50 machines, but only 48 can be contacted, the application control infrastructure should adapt appropriately and continue with only 48 machines.

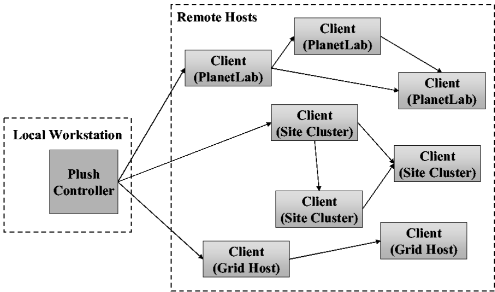

Figure 2: Plush application controller running on a local workstation connected to light-weight clients running on remote hosts in different environments.

```
<plush>
  <project name="demo_proj">
    <software name="demo_soft" type="tar">
      <package name="demo.tar" type="web">
        <path>http://plush.ucsd.edu/demo.tar</path>
      </package>
    </software>
    <configuration name="demo_conf">
      <cluster name="demo_group">
        <software name="demo_soft"/>
        <rspec>
          <num_hosts>50</num_hosts>
        </rspec>
        <execution>
          <process name="demo">
            <path>./demo.exe</path>
            <cmdline><arg>300</arg></cmdline>
          </process>
        </execution>
      </cluster>
      <resources>
        <resource type="plab" group="ucsd_3"/>
      </resources>
     </configuration>
    <application name="demo_app">
      <execution>
        <configuration name="demo_conf"/>
      </execution>
    </application>
  </project>
</plush>
```

Figure 3: Sample application specification that defines a process (`demo.exe`) that will run for 300 seconds on 50 remote PlanetLab hosts assigned to slice `ucsd_3`.

4 Plush

In this section, we describe Plush, an extensible application control infrastructure for large-scale distributed systems designed to meet the requirements from Section 3. A primary goal of Plush is to simplify the develop-deploy-debug cycle that researchers go through when developing large-scale distributed applications. Plush achieves this goal through a simple terminal interface where users can deploy, run, monitor, and debug their distributed applications running on hundreds of remote machines through basic terminal commands.

Unlike local clusters, wide-area environments like PlanetLab and the Grid typically do not use a common file system that is shared among all machines. This introduces challenges related to maintaining specific versions of code, setting up environment variables, or gathering output from all machines hosting the experiment. The result is that researchers who test applications in wide-area environments often end up spending more time in the cycle's deploy and debug phases than in development. Plush provides much of the needed functionality to ease the burden of deploying and debugging distributed applications, allowing users to spend more time developing.

4.1 Architecture

The main components of Plush are an application controller that typically runs on a user's workstation and a lightweight client process that runs on all nodes hosting the application. Though the remainder of this discussion focuses on PlanetLab, the set of managed clients is not limited to one platform. The same controller has the ability to manage clients across all supported platforms, including PlanetLab, the Grid, and any local clusters maintained at the users' site. Figure 2 depicts this architecture.

The application controller is the center of control for all Plush-managed applications[1]. Using authentication information provided by the user, Plush determines an available pool of resources at startup. For PlanetLab, the user specifies the slice name and Plush looks up all hosts assigned to the slice and automatically adds them to the user's resource pool. If other resources are desired, such as machines in a local cluster, these are specified separately by the user.

To run an application, the controller parses an application description (see Figure 3) and executes the specified actions. The XML description language is hierarchical. Projects are defined at the highest level, and they contain software, configuration, and application components. Software components describe where to locate the desired software packages, and how to transfer and unpack the files on the remote machines. Configurations contain "cluster" components that describe one or more clusters of machines. Clusters define exactly what physical resources are desired for each group of machines requested. Each cluster component also specifies what previously defined software components should be installed, in addition to specifying what exact commands should be run on each host during execution, and what resource pool to use for locating resources. The application components of the project specify the configurations that should be run. By using multiple configurations it is possible to define multiple applications at once and execute them sequentially at runtime.

After creating an available resource pool and parsing an abstract application description, the controller determines the resources that should host the application. Plush passes the required resource description to a user-selectable resource discovery package, which returns a set of physical machines that match the user's requirements. To bootstrap deployment, the Plush controller connects to the selected resources and copies the client to all remote hosts–thereby creating an underlying communication mesh–and then starts the client Plush processes. The application controller then installs the required software on all remote hosts. Several common file transfer protocols are supported by Plush, including scp,

[1]One or more backup controllers can be specified to handle controller failures. The details are omitted for clarity.

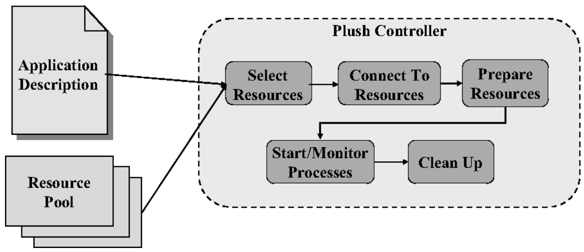

Figure 4: Steps taken by Plush controller.

wget, and rsync. When the controller is ready to start the application, it instructs the clients to execute the appropriate command on each remote host.

Once the application is running, the clients communicate with the application controller to notify it of status updates and potential failures. If a failure is detected, the controller attempts to recover from it according to the actions enumerated in the user's application specification. Since many failures are application-specific, Plush exports optional callbacks to the application itself to determine the appropriate reaction.

Plush provides a set of monitoring tools to help the user better understand status on the remote hosts. For example, Plush provides a shell interface that allows users to issue a command on all hosts simultaneously. Users also have the option of redirecting the `stdout` from the remote hosts, so that all output from all hosts is streamed back to the controller's terminal. Further, Plush monitors the liveness of nodes automatically and notifies the user when a node fails. When the application completes (or upon a user command), Plush stops all associated processes, transfers output data back to the controller's local disk, performs user-specified cleanup actions, kills the client processes, and disconnects the hosts from the communication mesh. The entire process is shown in Figure 4.

4.2 Barriers

Plush separates an application's lifecycle phases with synchronization barriers. Traditionally, synchronization barriers separate different phases of computation across multiple processes. Barriers were first introduced as a parallel-programming construct to synchronize individual processors interconnected by a high-speed network [13]. In the context of a Plush-managed application, barriers separate phases of execution across a set of remote machines. Barriers require all hosts involved in a distributed application to reach a specific point of execution before continuing to the next phase. For example, barriers separate the file transfer phase from the execution phase to ensure that a suitable set of resources is found and prepared with the appropriate software before attempting to start the application. In this same manner, barriers can be used to loosely synchronize the beginning of an execution across all remote hosts. Barriers can also separate different phases of staged executions, as are often present in Grid-style applications like EMAN.

Traditional barriers are not well suited for volatile, wide-area network conditions; the semantics are simply too strict. In order to achieve better resilience in the presence of failures, Plush extends traditional barrier semantics with two new relaxations. The first relaxation primitive, *early entry*, allows hosts that reach a barrier to be released before all hosts have entered. This prevents progress from stalling due to a small subset of delayed hosts. The second primitive, *throttled release*, allows the user to control the rate of release from a barrier. These relaxations are discussed in detail in Section 5.

5 Challenges

While Plush addresses some of the requirements described in Section 3, outstanding issues remain. This section describes a few of the challenges we have faced and the lessons we have learned thus far in our development of Plush.

Challenge: Create a language capable of succinctly describing application requirements. The language must be easy to understand but also expressive enough to support complex scenarios. Additionally, users will need to define each phase of an application's lifecycle within the language. Based on user experiences, we know that it is important to establish a balance between functionality and usability in the design of the application specification language. If the language gets too complicated, novice users may become intimidated by the complexity and give up. However, without support for advanced features, experienced users will be unable to express all of their requirements.

When designing the XML syntax for the application specification language of Plush, we decided to require only a small set of easily defined attributes, while also supporting a variety of specialized features. We believe that this establishes a balance between functionality and ease of use. We separated the various components of a distributed application and described them using an extensible schema that allows users to make the application specifications as complicated or basic as desired. In the simplest case, the user only needs to define the required software (if any), the number of machines desired, and the command to run on the remote machines.

In the future, we plan to provide a graphical user interface (GUI) for Plush. Part of this GUI will allow users to create application descriptions without writing XML. In the early stages of the development of Plush, we designed and built a simple GUI that allowed users to run and monitor applications, create application descriptions, and visualize various statistics about the status of the remote hosts. However we found that as we continued to develop Plush, the rapid rate of change of the code and design of the internal data structures made it difficult to maintain a functional GUI. Small changes in a data structure typically required significant changes to the layout of the GUI. The development of the GUI has since been postponed and will be resumed again in the upcoming months.

Challenge: Build a generic infrastructure that meets the demands of a variety of distributed applications and is as powerful as tools designed specifically for a single application. Our goal during the development of Plush was to build a general infrastructure for application management that controls all aspects of the distributed application lifecycle without sacrificing important features available in specialized tools. We quickly realized that the best way to do this was to use the existing tools directly, rather than trying to reinvent them. Hence, Plush is a customizable framework that provides the ability to incorporate existing tools. Users can modify their application description to plug in the specialized tools they need to execute and manage their applications. It is this "pluggable" aspect of Plush that allows users to run their applications in a variety of environments.

One challenge in designing an infrastructure that supports the ability to plug in arbitrary existing tools is implementing the glue code necessary to integrate each tool into Plush. Unfortunately, there are no official standards or common APIs to which developers adhere on PlanetLab. Hence, the integration of each tool must be addressed separately.

Challenge: Design an application control infrastructure that scales to hundreds or even thousands of heterogeneous machines. Currently, PlanetLab consists of over 600 machines at approximately 300 different locations around the world. Many Grid environments contain thousands of machines distributed worldwide. In order for an application control infrastructure to support distributed applications in these environments, it must scale to potentially thousands of machines while maintaining acceptable levels of performance. The initial design of Plush uses a star topology (as opposed to, say, a mesh) for communication, so every host running the application connects directly to the controller[2]. The limiting factor in the star design (and one that also circumscribes Plush's scalability) is the number of simultaneous connections the controller can support.

During the development of Plush, we experimented with several different designs that exhibited varying degrees of success with respect to scalability and performance. First, we used a fixed-size pool of threads and looped through remote connections. The problem with this approach is that the progress of the entire application was limited by a few slow hosts. Although we could scale to several hundred machines, the performance was unacceptable. To avoid the potential bottleneck created by slow hosts, we increased the number of threads in use so that each connection used two separate threads. The performance of this technique was much improved over the fixed-size thread pool. However this approach suffered from a variety of new problems, and ultimately could not scale beyond approximately 200 connections before some machines ran out of threads. Finally, we moved to the current event-driven design that uses a single thread and an event loop for execution. The performance of this approach is good, and the number of connections can scale to approximately 800. The limiting factor currently is the processor overhead required for XML serialization. We are working on ways to trim the XML and reduce this overhead, and hopefully increase the scalability further.

Another problem that arises in heterogeneous environments is inconsistencies in execution environments. This is particularly problematic when executables are dynamically linked to system libraries. We found that statically linking the client executable that runs on remote hosts helped solve this problem in most cases. When statically linking executables, however, it is important to avoid architecture-specific system calls, such as some cryptographic random number generators.

Challenge: Achieve forward progress in potentially volatile environments. Wide-area environments tend to be erratic, with failures both common and expected. Numerous errors can and often do occur that make it difficult to achieve forward progress. These errors include hardware failures, software configuration errors, network outages and congestion, and application failures. Sometimes even simple tasks, such as connecting to 100 PlanetLab machines and running the command "hostname," can prove troublesome to users. Further, machines can fail at any point during execution.

After experimenting with the use of barriers in Plush on PlanetLab, we found that traditional barrier semantics are too strict to be effective for many applications in volatile environments. We have found that the distribution of completion times for common actions (including file transfers and application execution) across a large set of PlanetLab hosts often exhibits a heavy tail. The majority of hosts finish the task in a reasonable time period, while a few hosts typically take orders of magnitude more time to complete the same task. Thus, requiring all hosts to reach a barrier before being simultaneously released often resulted in unacceptable performance.

Section 4 presented two relaxations (early entry and throttled release) for large-scale distributed computations where maintaining a large set of working machines is often difficult due to the failures and intermittent connectivity inherent to the environment. We discovered that by using early entry (which releases the barrier before all hosts have entered), progress is not delayed because of a few slow hosts. For example, if 98 out of 100 requested machines have installed the required software and are ready to execute, an early-entry barrier may release the 98 hosts after some timeout period without waiting an unbounded amount of time for the two remaining delayed hosts. For increased adaptability, Plush gives users the option of using a "knee-detecting" algorithm to dynamically determine the knee of the completion time cumulative distribution function, which triggers the release of the barrier early without waiting for the remaining hosts. This saves users from having to specify a static timeout period. Further, the user can register an application specific callback with Plush to specify the exact course of action desired to deal with the slow machines.

The second relaxation, throttled release, is analogous to a counting semaphore. With throttled release, nodes are released from the barrier at a controlled rate, rather than releasing all nodes simultaneously. This may be used, for example, to limit the number of nodes that simultaneously perform network measurements or software downloads. During the barrier configuration phase, users can specify an exact rate of release from the barrier, or they can choose to let Plush dynamically determine the optimal release rate. In the latter case, Plush adapts to changing conditions in at attempt to find the release rate that provides the most throughput for the

[2] In future versions of Plush, we plan to remove the star and build a more scalable topology, perhaps based on a tree.

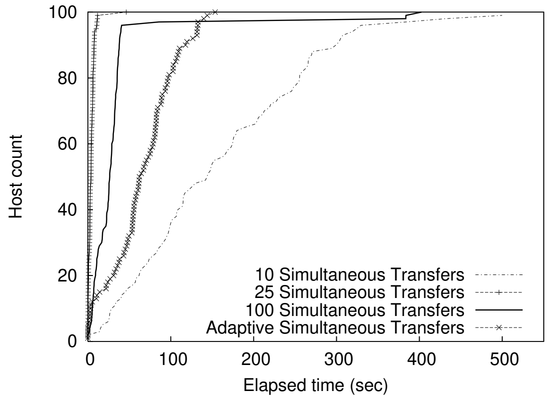

Figure 5: Software transfer (10 MB file) completion time from a local host to 100 PlanetLab hosts using throttled release to limit the number of simultaneous file transfers. 25 simultaneous transfers completes quicker than 100 simultaneous transfers. The "adaptive" curve shows the results of letting Plush dynamically determine the release rate based on the current network conditions.

application. Figure 5 shows the benefits achieved using this semantic for simultaneous file transfers.

Even though the file transfer example shown in Figure 5 can achieve better performance with a statically defined release rate, it is important for Plush to have the capability to adapt to changing conditions (shown in the "adaptive" curve in the graph). It is difficult to predict what the network conditions will be like in the future, and while using 25 simultaneous transfers completed the fastest in the experiment shown, it is very likely that the same setting will not perform best at all times. In most cases the user does not know what the optimal release rate should be before running the application. In addition, the use of throttled release barriers is not limited to just file transfers across the wide area. It may be even more difficult to accurately predict a release rate for other uses of throttled release barriers.

6 Related work

Several tools are available for easing remote job execution, including cfengine [5], gexec [10], and vxargs [22] among others. In general, these products provide a subset of the functionality that Plush provides. In addition to remote job execution, Plush reacts to a variety of failure conditions, and provides methods for automatic reconfiguration in response to changing conditions. Further, existing tools for resource discovery and allocation can be plugged into Plush, providing more advanced functionality than most remote job execution tools provide.

Aside from general-purpose remote job execution tools, there are a few projects that focus on managing distributed applications. The PlanetLab Application Manager [12] provides many of the same features as Plush does for long-running services on PlanetLab, but does not support short-lived applications as easily. It is designed to help maintain applications that provide a service and require high availability. It does not provide a way to interactively execute commands on remote machines, and scripts must be manually created for each managed application.

SmartFrog [11] also manages distributed applications. It is a framework for describing, deploying, and controlling distributed applications. It consists of a description language and a collection of daemons that manage distributed applications. Unlike Plush, SmartFrog is not a tool that can be used to interactively control distributed applications. It is a framework for building configurable systems and has components with similar functionality to those in Plush, but does not provide a packaged product or solution for managing applications.

In the Grid community, there are several projects that have similar goals as Plush. Condor [4] is a workload management system for compute-intensive jobs. Plush is similar to Condor in that both deploy and manage distributed executions. Condor is optimized for leveraging underutilized cycles in desktop machines in an organization, where each job or application is generally compute-bound and highly parallelizable. On the other hand, Plush is designed to deploy and manage naturally distributed tasks, which may include requirements for the concurrent scheduling of resources over several sites. Condor provides its own batch scheduler, and can schedule resources with a much greater efficiency than can Plush. Plush supports a wider range of scheduling policies, however, pushing those decisions to external resource allocators. Since Plush does not focus on a single class of distributed applications, it supports a wider range of reactions to failure than Condor.

GrADS/vGrADS [3] is another Grid project that provides a set of programming tools and an execution environment for easing program development in computational grids. In particular, GrADS focuses 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. There is less support for a broader range of failure recovery actions than in Plush. vGrADS is an extension of the GrADS project that adds an abstraction layer for "virtual grids," and provides added support for Grid economies.

Perhaps the most widely used software package for grid development, the Globus Toolkit [7] is a framework for building Grid systems and applications. Several components of Globus are similar to Plush. The Globus Resource Specification Language (RSL) provides an abstract language for describing resources. It is very similar in design to our application description language. The Globus Resource Allocation Manager (GRAM) also provides much of the same functionality as Plush does. It processes requests for resources, allocates the resources, and manages active jobs in Grid environments. The main difference between Plush and the tools in Globus is that Plush provides a user interface where users can directly interact with their applications. Since Globus is a framework, each application must use the APIs to create the desired functionality. In the future, we plan to integrate Plush with some of the Globus tools, such as GRAM and RSL. In this scenario Plush will act as a front-end user interface for the tools available in Globus.

7 Status and conclusion

Plush is designed to meet the needs of a wide range of PlanetLab users. By explicitly considering three different classes of distributed applications that often run in large-scale heterogeneous environments such as PlanetLab, we attempted to extract a general set of requirements for application management. While these requirements are admittedly challenging, Plush represents the culmination of two years of development aimed at addressing these challenges in a streamlined and powerful manner.

Plush is far from a panacea; experience has repeatedly shown that completeness and ease of use are often at odds. In these instances, Plush attempts to err on the side of usability, instead leveraging the ability to interface with external, third-party tools where appropriate. Plush eases the burden of deploying and maintaining distributed applications by focusing on providing the following key functionality:

- An extensible specification language for describing a variety of distributed applications
- Interfaces for defining application-specific tools for various stages of the distributed-application life cycle
- Automated application monitoring and reconfiguration
- Relaxed synchronization semantics for failure-prone wide-area environments

Plush is currently in daily use. Source, binaries, and more information can be found at `http://sysnet.ucsd.edu/plush/`.

References

[1] A. AuYoung, B. N. Chun, A. C. Snoeren, and A. Vahdat. Resource allocation in federated distributed computing infrastructures. In *OASIS*, 2004.

[2] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating Systems Support for Planetary-Scale Network Services. In *NSDI*, 2004.

[3] F. Berman, H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, B. Liu, X. Liu, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, C. Mendes, A. Olugbile, M. Patel, D. Reed, Z. Shi, O. Sievert, H. Xia, and A. YarKhan. New grid scheduling and rescheduling methods in the GrADS project. *IJPP*, 33(2-3), 2005.

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

[5] M. Burgess. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3), 1995.

[6] EMAN. `http://ncmi.bcm.tmc.edu/EMAN/`.

[7] I. Foster. A globus toolkit primer, 2005.

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

[9] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with coral. In *NSDI*, 2004.

[10] gexec. `http://www.theether.org/gexec/`.

[11] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. SmartFrog: Configuration and automatic ignition of distributed applications. In *HP OVUA*, 2003.

[12] R. Huebsch. PlanetLab application manager. `http://appmanager.berkeley.intel-research.net`.

[13] H. F. Jordan. A Special Purpose Architecture for Finite Element Analysis. In *ICPP*, 1978.

[14] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.

[15] D. Oppenheimer, J. Albrecht, D. A. Patterson, and A. Vahdat. Design and implementation tradeoffs for wide-area resource discovery. In *HPDC*, 2005.

[16] V. S. Pai, L. Wang, K. Park, R. Pang, and L. Peterson. The dark side of the web: An open proxy's view. In *HotNets-II*, 2003.

[17] K. Park and V. S. Pai. Deploying large file transfer on an http content distribution network. In *WORLDS*, 2004.

[18] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *HotNets*, 2002.

[19] S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the embarrassing slowness of opendht on planetlab. In *WORLDS*, 2005.

[20] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, , and H. Yu. OpenDHT: A public DHT service and its uses. In *SIGCOMM*, 2005.

[21] P. S. C. Service. `https://snowball.cs.uga.edu/~dkl/pslogin.php`.

[22] vxargs. `http://dharma.cis.upenn.edu/planetlab/vxargs/`.