Remote Control: Distributed Application Configuration, Management, and Visualization with Plush

Jeannie Albrecht – Williams College
Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat – University of California, San Diego

ABSTRACT

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, maintenance, and visualization 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. In this paper, 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 and execution environments. 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. 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.

Introduction

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 re-configuring them if operating conditions change). It is not surprising, then, that a number of tools have been developed to address various aspects of the process in distributed environments, but no solution yet flexibly automates the application deployment and management process across all environments.

Presently, most researchers who want to evaluate their applications in wide-area distributed environments take one of three management approaches. On PlanetLab [6, 26], 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 toolkits (such as the Globus Toolkit [12]) for application development and deployment. These toolkits often require tight integration with not only the infrastructure, but the application itself. Hence, applications must be custom tailored for a given toolkit, and can not easily be run in other environments. Similarly, system administrators who are responsible for configuring resources in machine-room settings often use remote execution tools such as cfengine [9] for managing and configuring networks of machines. As in the other two approaches, however, the configuration files are tailored to a specific environment and a particular set of resources, and thus are not easily extended to other platforms.

Motivated by the limitations of existing approaches, we believe that a unified set of abstractions for achieving availability, scalability, and fault tolerance can be applied to a broad range of distributed applications, shielding developers from some of the complexities of large-scale networked environments. The primary goal of our research is to understand these abstractions and define interfaces for specifying and managing distributed computations run in *any* execution environment. We are not trying to build another toolkit for managing distributed applications. Rather, we hope to define the way users think about their applications, regardless of their target platform. We took inspiration from classical

operating systems like UNIX [28] 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 testbeds do not provide global file system abstractions, which complicates data management.

To this end, we present Plush [27], a generic application management infrastructure that provides a unified set of abstractions for specifying, deploying, and monitoring distributed applications. Although Plush was initially designed to support applications running on PlanetLab [2], Plush now provides extensions that allow users to manage distributed applications in a variety of computing environments. 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 the deployment life cycle to fit the needs of an application and its target infrastructure. Users can, for example, specify a particular resource discovery service to use during application deployment. Plush also provides extensive failure management support to automatically adapt to failures in the application and the underlying computational infrastructure. Users interact with Plush through a simple command-line interface or a graphical user interface (GUI). Additionally, Plush exports an XML-RPC interface that allows users to programmatically integrate their applications with Plush if desired.

Plush provides abstractions for managing resource discovery and acquisition, software distribution, and process execution in a variety of distributed environments. Applications are specified using combinations of Plush application "building blocks" that define a custom control flow. Once an application is running, Plush monitors it for failures or application-level errors for the duration of its execution. Upon detecting a problem, Plush performs a number of user-configurable recovery actions, such as restarting the application, automatically reconfiguring it, or even searching for alternate resources. For applications requiring wide-area synchronization, Plush provides several efficient synchronization primitives in the form of partial barriers, which help applications achieve better performance and robustness in failure-prone environments [1].

The remainder of this paper discusses the architecture of Plush. We motivate the design in the next section by enumerating a set of general requirements for managing distributed applications. Subsequently, we present details about the design and implementation of Plush and then provide specific application case studies and uses of Plush. Related work is shown in the next section which is followed by the conclusion.

Application Management Requirements

To better understand the requirements of a distributed application management framework, we first consider how we might run a specific application in a widely-used distributed environment. In particular, we investigate the process of running SWORD [23], a publicly-available resource discovery service, on PlanetLab. SWORD uses a distributed hash table (DHT) for storing data, and aims to run on as many hosts as possible, as long as the hosts provide some minimum level of availability (since SWORD provides a service to other PlanetLab users). Before starting SWORD, we have to find and gain access to PlanetLab machines capable of hosting the service. Since SWORD is most concerned with reliability, it does not necessarily need powerful machines, but it must avoid nodes that frequently perform poorly over a relatively long time. We locate reliable machines using a tool like CoMon [25], which monitors resource usage on PlanetLab, and then we install the SWORD software on those machines.

This software installation involves downloading the SWORD software package on each host individually, unpacking the software, and installing any software dependencies, including a Java Runtime Environment. After the software has been installed on all of the selected machines, we start the SWORD execution. Recall that reliability is important to SWORD, so if an error or failure occurs at any point, we need to quickly detect it (perhaps using custom scripts and cron jobs) and restore the service to maintain high availability.

Running SWORD on PlanetLab is an example of a specific distributed application deployment. The low-level details of managing distributed applications *in general* largely depend on the characteristics of the target application and environment. For example, long-running services such as SWORD prefer reliable machines and attempt to dynamically recover from failures to ensure high availability. On the other hand, short-lived scientific parallel applications (e.g., EMAN [18]) prefer powerful machines with high bandwidth/low latency network connections. Long term reliability is not a huge concern for these applications, since they have short execution times. At a high level, however, if we ignore the complexities associated with resource management, the requirements for managing distributed applications are largely similar for all applications and environments. Rather than reinvent the same infrastructure for each class separately, our goal is to identify common abstractions that support the execution of many types of distributed applications, and to build an application-management infrastructure that supports the general requirements of all applications. In this section, we identify these general requirements for distributed application management.

Specification. A generic application controller must allow application operators to customize the

control flow for each application. This *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, including the software required to run the application, the processes that will run on each machine, the resources required to achieve the desired performance, and any environment-specific execution parameters. User credentials for resources must also be included in the application specification in order to obtain access to resources. To manage complex multi-phased computations, such as scientific parallel applications, the specification must support application synchronization requirements. Similarly, distributing computations among pools of machines requires a way to specify a workflow – a collection of tasks that must be completed in a given order – within an application specification.

The complexity of distributed applications varies greatly from simple, single-process applications to elaborate, parallel applications. Thus the challenge is to define a specification language abstraction that provides enough expressibility for complex distributed applications, but is not too complicated for single-process computations. In short, the language must be simple enough for novice application developers to understand, yet expose enough advanced functionality to run complex scenarios.

Resource Discovery and Acquisition. Another key abstraction in distributed applications are *resources*. Put simply, resources are computing devices that are connected to a network and are capable of hosting an application. Because resources in distributed environments are often heterogeneous, application developers naturally want to find the resource set that best satisfies the demands of their application. Even if hardware is largely homogeneous, dynamic resource characteristics such as available bandwidth or CPU load can vary over time. The goal of resource discovery is to find the best *current* set of resources for the distributed application as described in the specification. In environments that support dynamic virtual machine instantiation [5, 30], these resources may not exist in advance. Thus, resource discovery in this case involves finding the appropriate physical machines to host the virtual machine configurations.

Resource discovery systems often interact directly with resource acquisition systems. Resource acquisition involves obtaining a lease or permission to use the desired resources. Depending on the execution environment, acquisition can take a number of forms. For example, to support advanced resource reservations as in a batch pool, resource acquisition is responsible for submitting a resource request and subsequently obtaining a lease from the scheduler. In virtual machine environments, resource acquisition may involve instantiating virtual machines, verifying their successful creation, and gathering the appropriate information (e.g., IP address, authentication keys) required for access. The challenge facing an application management framework is to provide a generic resource-management interface. Ultimately, the complexities associated with creating and gaining access to physical or virtual resources should be hidden from the application developer.

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 any necessary executables to start the application. This involves copying, unpacking, and installing the software on the target hosts. The application controller must support a variety of different file-transfer mechanisms for each environment, and should react to failures that occur during the transfer of software or in starting executables.

One important aspect of application deployment is configuring the requested number of resources with compatible versions of the software. Ensuring that a minimum number of resources are available and correctly configured for a computation may involve requesting new resources from the resource discovery and acquisition systems to compensate for failures that occur at startup. Further, many applications require some form of synchronization across hosts to guarantee that various phases of computation start at approximately the same time. Thus, the application controller must provide mechanisms for loose synchronization.

Maintenance. Perhaps the most difficult requirement for managing distributed applications is monitoring and maintaining an application after execution begins. Thus, another abstraction that the application controller must define is support for customizable application maintenance. One key aspect of maintenance is application and resource monitoring, which involves probing hosts 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.

In some cases, system failures may result in a situation where application requirements can no longer be met. For example, if an application is initially configured to be deployed on 50 resources, but only 48 can be contacted at a certain point in time, the application controller should adapt the application, if possible, and continue executing with only 48 machines. Similarly, different applications have different policies and requirements with respect to failure recovery. Some applications may be able to simply restart a failed process on a single host, while others may require the entire execution to abort in the case of failure. Thus, in addition to the other features previously described, the application controller should support a variety of options for failure recovery.

Plush: Design and Implementation

We now describe Plush, an extensible distributed application controller, designed to address the requirements of large-scale distributed application management discussed in the second section.

To directly monitor and control distributed applications, Plush itself must be distributed. Plush uses a client-server architecture, with clients running on each resource (e.g., machine) involved in the application. The Plush server, called the *controller*, interprets input from the user (i.e., the person running the application) and sends messages on behalf of the user over an overlay network (typically a tree) to Plush *clients*. The controller, typically run from the user's workstation, directs the flow of control throughout the life of the distributed application. The clients run alongside each application component across the network and perform actions based upon instructions received from the controller.

Figure 1a shows an overview of the Plush controller architecture. (The client architecture is symmetric to the controller with only minor differences in functionality.) The architecture consists of three main sub-systems: the application specification, core functional units, and user interface. Plush parses the application specification provided by the user and stores

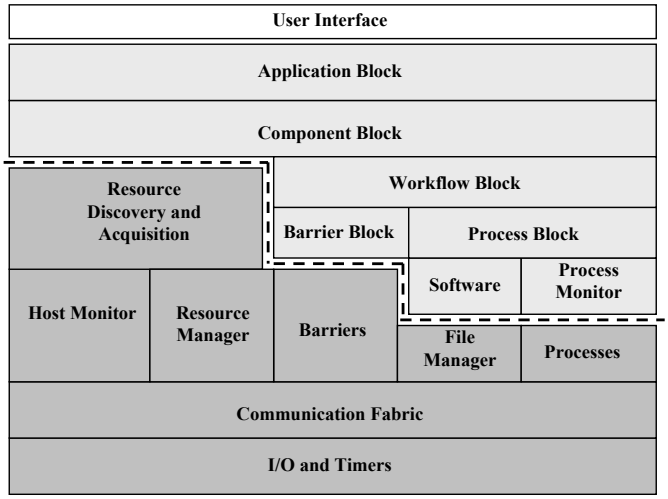

Figure 1a: The architecture of Plush. The *user interface* is shown above the rest of the architecture and contains methods for interacting with all boxes in the lower sub-systems of Plush. Boxes below the user interface and above the dotted line indicate objects defined within the *application specification* abstraction. Boxes below the line represent the *core functional units* of Plush.

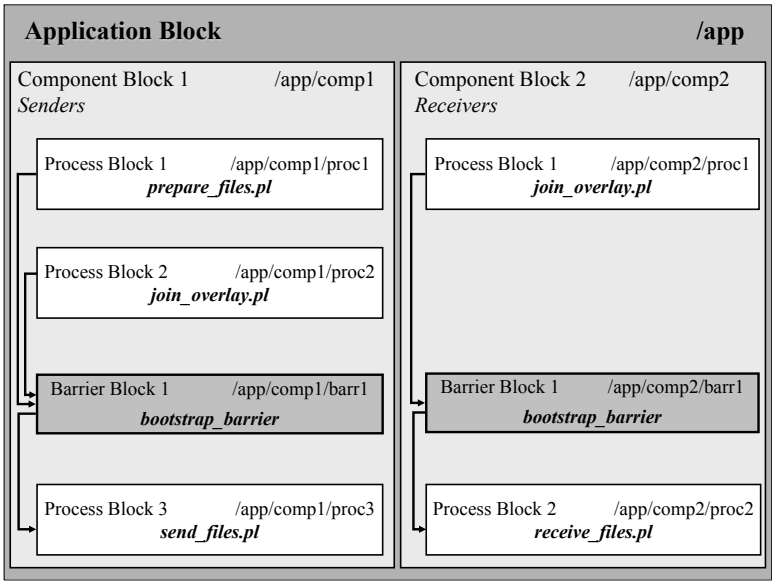

Figure 1b: Example file-distribution application comprised of application, component, process, and barrier blocks in Plush. Arrows indicate control-flow dependencies (i.e., X → Y implies that X must complete before Y starts).

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 physical machines, 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 sub-systems.[1]

Application Specification

Developing a complete, yet accessible, application specification language was one of the principal challenges in this work. Our approach, which has evolved over the past three years, consists of combinations of five different abstractions:

1. **Process blocks** – Describe the processes executed on each machine 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 remote machine.
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 block may contain process and barrier blocks. For example, a workflow block might describe a set of input files over which a process or barrier block will iterate during execution.
4. **Component blocks** – Describe the groups of resources required to run the application. This includes expectations specific to a set of metrics for the target resources. In the case of compute nodes in a cluster, for example, these 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 a simple file-distribution application as shown in Figure 1b. This application consists of two groups of machines. One group, the senders, stores the files, and the second group, the receivers, attempt 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 example, there are two phases of execution. In the first phase, all senders and receivers join the overlay before any transfers begin, and the senders must prepare the files for transfer before the receivers start receiving files. In the second phase, the receivers begin receiving the files. No new senders or receivers are allowed to join the network during the second phase.

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

The application block also contains one or many component blocks that describe the groups of resources (i.e., machines) required to run the application. Our application consists of a set of senders and a set of receivers, and two separate component blocks describe the two groups of machines. The sender component block defines the location and installation instructions for the sender software, and includes authentication information to access the resources. Similarly, the receiver component block defines the receiver software package. In our example, it may be desirable to require that all machines in the sender group have a processor speed of at least 1 GHz, and each sender should have sufficient bandwidth for sending files to multiple receivers at once. These types of machine-specific requirements are included in the component blocks. Within each component block, a combination of workflow, process, and barrier blocks describe the distributed computation.[2]

Plush process blocks describe the specific commands required to execute the application. Most process blocks depend on the successful installation of **software** packages defined in the component blocks. Users specify the commands required to start a given process, and actions to take upon process exit. The exit policies create a Plush **process monitor** that oversees the execution of a specific process. Our example has several process blocks. In the sender component, process blocks define processes for preparing the files, joining the overlay, and

[1] Note that the components within the sub-systems are highlighted using boldface throughout the text in the remainder of this section.

[2] Although our example does not use workflow blocks, they are used in applications where data files must be distributed and iteratively processed.

sending the files. Similarly, the receiver component contains process blocks for joining the overlay and receiving the files.

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

In Figure 1b, the outer application block contains our two component blocks that run in parallel (since there are no arrows indicating control-flow dependencies between them). Within the component blocks, the different phases are separated by the bootstrap barrier that is defined by barrier block 1. Component block 1, which describes the senders, contains process blocks 1 and 2 that define perl scripts that run in parallel during phase one, synchronize on the barrier in barrier block 1, and then proceed to process block 3 in phase two which sends the files. Component block 2, which describes the receivers, runs process block 1 in phase one, synchronizes on the barrier in barrier block 1, and then proceeds to process block 2 in phase two which runs the process that receives the files. In our implementation, the blocks are represented by XML that is parsed by the Plush controller when the application is run. We show an example of the XML later.

We designed the Plush application specification to support a variety of execution patterns. With the blocks described above, Plush supports the arbitrary combination of processes, barriers, and workflows, provided that the flow of control between them forms a directed acyclic graph. Using predecessor tags in Plush, users specify the flow of control and define whether processes run in parallel or sequentially. Arrows between blocks in Figure 1b, for example, indicate the predecessor dependencies. (Process blocks 1 and 2 in component block 1 will run in parallel before blocking at the bootstrap barrier, and then the execution will continue on to process block 3 after the bootstrap barrier releases.) Internally, Plush stores the blocks in a hierarchical data structure, and references specific blocks in a manner similar to referencing absolute paths in a UNIX file system. Figure 1b shows the unique path names for each block from our file-distribution example. This naming abstraction also simplifies coordination among remote hosts. Each Plush client maintains an identical local copy of the application specification. Thus, for communication regarding control flow changes, the controller sends the clients messages indicating which ‘‘block’’ is currently being executed, and the clients update their local state information accordingly.

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 1a shows these units as shaded boxes below the dotted line. The functional units manipulate the objects defined in the application specification to manage distributed applications. In this section, we describe the role of each of these units.

Starting at the highest level, the Plush **resource discovery and acquisition** unit uses the resource requirements in the component blocks to locate and create (if necessary) resources on behalf of the user. The resource discovery and acquisition unit is responsible for obtaining a valid set, called a *matching*, of resources that meet the application’s demands. To determine this matching, Plush may either call an existing external service to construct a resource pool, such as SWORD or CoMon for PlanetLab, or use a statically defined resource pool based on information provided by the user. The Plush *resource matcher* then uses the resources in the resource pool to create a matching for the application. All hosts involved in an application run a Plush **host monitor** that periodically publishes information about the host. 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 1a are responsible for application deployment and maintenance. These units connect to resources, install required software, start the execution, and monitor the execution for failures. One important functional unit used for these operations is the Plush **barrier manager**, which provides advanced synchronization services for Plush and the application itself. In our experience, traditional barriers [17] are not well suited for volatile, wide-area network conditions; the semantics are simply too strict. Instead, Plush uses partial barriers, which are designed to perform better in volatile environments [1]. Partial barriers ensure that the execution makes forward progress in the face of failures, and improve performance in failure-prone environments using relaxed synchronization semantics.

The Plush **file manager** handles all files required by a distributed application. This unit contains information regarding software packages, file transfer methods,

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

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

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

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.

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 machines spread across the wide-area. In this section we explore fault tolerance and scalability in Plush.

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 ever needing to involve the user running the application. Rather than enumerate all possible failures that may occur, we will discuss how Plush handles three common failure classes – process, host, and controller failures.

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

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

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

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

This technique works if the host experiences a temporary power or network outage, and is only unreachable for a short period of time. If the controller is unable to reconnect to the host, the next option is to rematch in an attempt to replace the failed host with a different host. In this case, Plush reruns the resource matcher to find a new machine. Depending on the application, the entire execution may need to be restarted across all hosts after the new host joins the control overlay, or the execution may only need to be started on the new host. If the controller is unable to find a new host to replace the failed host, Plush aborts the entire application.

In some applications, it is desirable to mark a host as failed when it becomes overloaded or experiences poor network connectivity. The Plush host monitor that runs on each machine is responsible for periodically informing the controller about each machine's status. If the controller determines that the performance is less than the application tolerates, it marks the host as failed and attempts to rematch. This functionality is a preference specified at startup. Although Plush currently monitors host-level metrics including load and free memory, the technique is easily extended to encompass sophisticated application-level expectations of host viability.

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

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

Scalability

In addition to fault tolerance, an application controller designed for large-scale environments must scale to hundreds or even thousands of participants. Unfortunately there is a tradeoff between performance and scalability. The solutions that perform the best at moderate scale typically do not scale as well as solutions with lower performance. To balance scalability and performance, Plush provides users with two topological alternatives.

By default, all Plush clients connect directly to the controller forming a star topology. This architecture scales to approximately 300 remote hosts, 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 host failure, only the failed host is affected. Further, the time required for the controller to exchange messages with clients is short due to the direct connections.

At larger scales, network and file descriptor limitations at the controller become a bottleneck. To address this, Plush also supports tree topologies. In an effort to reduce the number of hops between the clients and the controller, we construct ''bushy'' trees, where the depth of the tree is small and each node in the tree has many children. The controller is the root of the tree. The children of the root are chosen to be well-connected and historically reliable hosts whenever possible. Each child of the root acts as a ''proxy controller'' for the hosts connected to it. These proxy controllers send invitations and receive joins from other hosts, reducing the total number of messages sent back to the root controller. Important messages, such as failure notifications, are still sent back to the root controller. Using the tree topology, we have been able to use Plush to manage an application running on 1000 ModelNet [29] emulated hosts, as well as an application running on 500 PlanetLab clients. We believe that Plush has the ability to scale by another order of magnitude with the current design.

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

Running An Application Using Plush

In this section, we will discuss how the architectural components of Plush interact to run a distributed application. When starting Plush, the user's workstation becomes the controller. The user submits an application specification to the Plush controller. The controller parses the specification, and internally creates the objects shown above the dotted line in Figure 1a.

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

If a Plush client agrees to run the application, the client sends a JOIN message back to the controller accepting the invitation. Next, the controller sends a PREPARE message to the new client, which contains a copy of the application specification (XML representation). The client parses the application specification,

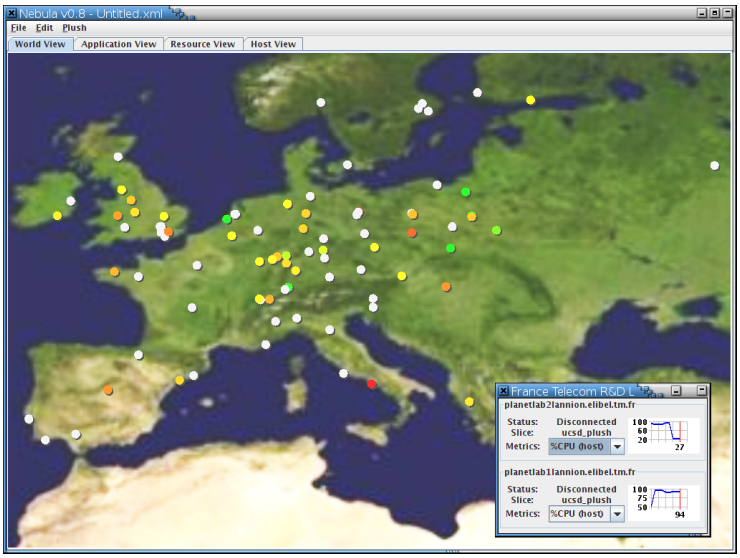

Figure 2a: Nebula World View tab showing an application running on PlanetLab. Different colored dots indicate PlanetLab sites in various stages of execution.

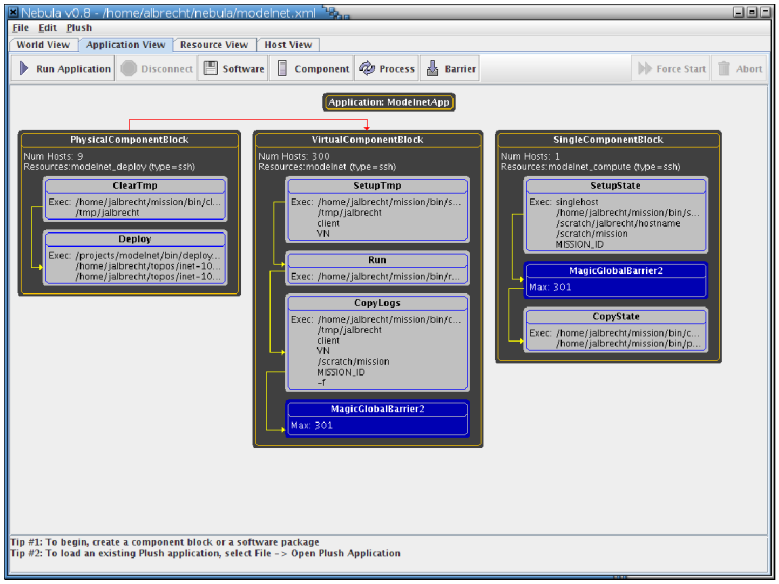

Figure 2b: Nebula Application View tab displaying Plush application specification.

starts a local host monitor, sends a PREPARED message back to the controller, and waits for further instruction. Once enough hosts join the overlay and agree to run the application, the controller initiates the beginning of the application deployment stage by sending a GO message to all connected clients. The file managers then begin installing the requested software and preparing the hosts for execution.

In most applications, the controller instructs the hosts to begin execution after all hosts have completed the software installation. (Synchronizing the beginning of the execution is not required if the application does not need all hosts to start simultaneously.) Since each client has now created an exact copy of the controller's application specification, the controller and clients exchange messages about the application's progress using the block naming abstraction (i.e., /app/comp1/proc1) to identify the status of the execution. For barriers, a barrier manager running on the controller determines when it is appropriate for hosts to be released from the barriers.

Upon detecting a failure, clients notify the controller, and 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 for some failure conditions. When the application completes (or upon a user command), Plush stops all associated processes, transfers output data back to the controller's local disk if desired, performs user-specified cleanup actions on the resources, disconnects the resources from the overlay by closing the TCP connections, and stops the Plush client processes.

User Interface

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

Figure 1a shows the user interface 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 forces the resource discovery and acquisition unit to find a new set of resources using a terminal command. We designed Plush in this way to provide maximum control over the application. At any point, the user can override a default Plush behavior. The effect is a customizable application controller that supports a variety of distributed applications.

Graphical User Interface

In an effort to simplify the creation of application specifications and help visualize the status of executions running on resources around the world, we implemented a graphical user interface for Plush called Nebula. In particular, we designed Nebula (as shown in Figures 2a, 2b, and 3) to simplify the process of specifying and managing applications running across the PlanetLab wide-area testbed. Plush obtains data from the PlanetLab Central (PLC) database to determine what hosts a user has access to, and Nebula uses this information to plot the sites on the map. To start using Nebula, users have the option of building their Plush application specification from scratch or loading a preexisting XML document representing an application specification. Upon loading the application specification, the user runs the application by clicking the Run button from the Plush toolbar, which causes Plush to start locating and acquiring resources.

The main Nebula window contains four tabs that show different information about user's application. In the ''World View'' tab, users see an image of a world map with colored dots indicating PlanetLab hosts. Different colored dots on the map indicate sites involved in the current application. In Figure 2a, the dots (which range from red to green on a user's screen) show PlanetLab sites involved in the current application. The grey dots are other available PlanetLab sites that are not currently being used by Plush. As the application proceeds through the different phases of execution, the sites change color, allowing the user to visualize the progress of their application. When failures occur, the impacted sites turn red, giving the user an immediate visual indication of the problem. Similarly, green dots indicate that the application is executing correctly. If a user wishes to establish an SSH connection directly to a particular resource, they can simply right-click on a host in the map and choose the SSH option from the pop-up menu. This opens a new tab in Nebula containing an SSH terminal to the host. Users can also mark hosts as failed by right-clicking and choosing the Fail option from the pop-up menu if they are able to determine failure more quickly than Plush's automated techniques. Failed hosts are completely removed from the execution.

Users retrieve more detailed usage statistics and monitoring information about specific hosts (such as CPU load, free memory, or bandwidth usage) by double clicking on the individual sites in the map. This opens a second window that displays real-time graphs based on data retrieved from resource monitoring tools, as shown in the bottom right corner of Figure 2a. The second smaller window displays a graph of the CPU or memory usage, and the status of the application on each host. Plush currently provides built-in support for monitoring CoMon [25] data on PlanetLab machines, which is the source of the CPU and memory data. Additionally, if the user wishes to view the CPU usage or percentage of free memory available across all hosts, there is a menu item under the PlanetLab menu that changes the colors of the dots on the map such that red means high CPU usage or low free

memory, and green indicates low CPU usage and high free memory. Users can also add and remove hosts to their PlanetLab account (or *slice* in PlanetLab terminology) directly by highlighting regions of the map and choosing the appropriate menu option from the PlanetLab menu. Additionally, users can renew their PlanetLab slice from Nebula.

The second tab in the Nebula main window is the "Application View." The Application View tab, shown in Figure 2b, allows users to build Plush application specifications using the blocks described previously. Alternatively, users may load an existing XML file describing an application specification by choosing the Load Application menu option under the File menu. There is also an option to save a new application specification to an XML file for later use. After creating or loading an application specification, the Run button located on the Application View tab starts the application.

The Plush blocks in the application specification change to green during the execution of the application to indicate progress. After an application begins execution, users have the option to "force" an application to skip ahead to the next phase of execution (which corresponds to releasing a synchronization barrier), or aborting an application to terminate execution across all resources. Once the application aborts or completes execution, the user may either save their application specification, disconnect from the Plush communication mesh, restart the same application, or load and run a new application by choosing the appropriate option from the File menu.

The third tab is the "Resource View" tab. This tab is blank until an application starts running. During execution, this tab lists the specific machines that are currently involved in the application. If failures occur during execution, the list of machines is updated dynamically, such that the Resource View tab always contains an accurate listing of the machines 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, shown in Figure 3. This tab contains a table that displays the hostname of all available resources. In the right column, the status of the host is shown. The purpose of this tab is to give users another alternative to visualize the status of an executing application. The status of the host in the right column corresponds to the color of the dot in the "World View" tab. This tab also allows users to run shell commands simultaneously on several resources, and view the output. As shown in Figure 3, users can select multiple hosts as once, run a command, and the output is shown in the text-box at the bottom of the window. Note that hosts do not have to be involved in an application in order to take advantage of this feature. Plush will connect to any available resources and run commands on behalf of the user. Just as in the World View tab, right-clicking on hosts in the Host View tab opens a pop-up menu that enables users to SSH directly to the hosts.

Command-line Interface

Motivated by the popularity and familiarity of the shell interface in UNIX, Plush further streamlines the develop-deploy-debug cycle for distributed application management through a simple command-line interface where users deploy, run, monitor, and debug their distributed applications running on hundreds of remote machines. Plush combines the functionality of a distributed shell with the power of an application

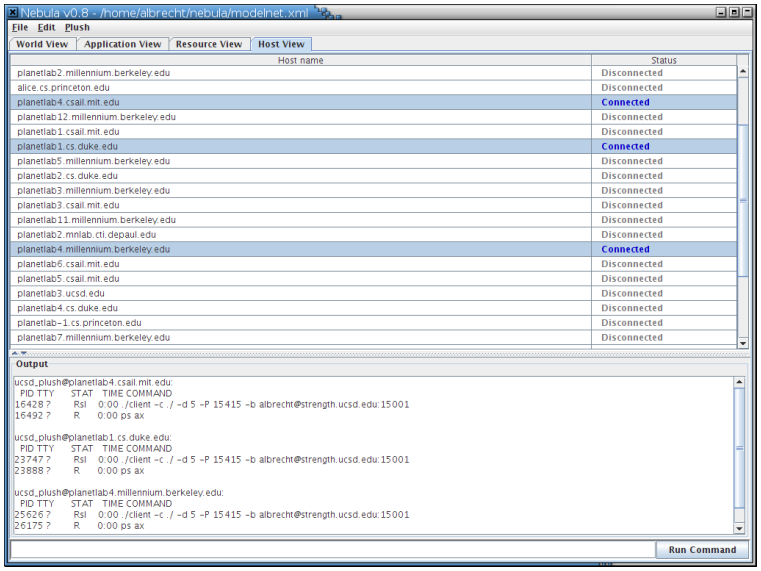

Figure 3: Nebula Host View tab showing PlanetLab resources. This tab allows users to select multiple hosts at once and run shell commands on the selected resources. The text-box at the bottom shows the output from the command.

controller to provide a robust execution environment for users to run their applications. From a user's standpoint, the Plush terminal looks 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 1 shows some of the available commands.

Command	Description
load <file>	Load application specification
connect <host>	Connect to host and start client
disconnect	Close all connections and clients
info nodes	Print all resource information
info mesh	Print communication fabric status info
info control	Print application control state info
run	Start the application (after load)
shell <cmd>	Run "cmd" on all connected resources

Table 1: Sample Plush terminal commands.

Programmatic Interface

Many commands that are available via the Plush command-line interface are also exported via an XML-RPC interface to deliver similar functionality as the command-line to those who desire programmatic access. This allows Plush to be scripted and used for remote execution and automated application management, and also enables the use of external services for resource discovery, creation, and acquisition within Plush. In addition to the commands that Plush exports, external services and programs may also register themselves with Plush so that the controller can send callbacks to the XML-RPC client when various actions occur during the application's execution.

Figure 4 shows the Plush XML-RPC API. The functions shown in the PlushXmlRpcServer class are available to users who wish to access Plush programmatically in scripts, or for external resource discovery and acquisition services that need to add and remove resources from the Plush resource pool. The plush AddNode(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.

Aside from resource discovery and acquisition services, the XML-RPC API allows for the implementation of different user interfaces for Plush. Since almost all of the Plush terminal commands are available as XML-RPC function calls, users are free to implement their own customized environment specific user interface without understanding or modifying the internals of the Plush implementation. This is beneficial because it gives the users more flexibility to develop in the programming language of their choice. Most mainstream programming languages have support for XML-RPC, and hence users are able to develop interfaces for Plush in any language, provided that the chosen language is capable of handling XML-RPC. For example, Nebula is implemented in Java, and uses the XML-RPC interface shown in Figure 4 to interact with a Plush controller. To increase the functionality and simplify the development of these interfaces, the Plush XML-RPC server has the ability to make callbacks to programs that register with the Plush controller via setXmlRpcClientUrl(string). Some of the more common callback functions are shown in the bottom of Figure 4 in class PlushXmlRpcCallback. Note that these callbacks are only useful if the programmatic client implements the corresponding functions.

```
class PlushXmlRpcServer extends XmlRpcServer {
    void plushAddNode(HashMap properties);
    void plushRemoveNode(string hostname);
    string plushTestConnection();
    void plushCreateResources();
    void plushLoadApp(string filename);
    void plushRunApp();
    void plushDisconnectApp(string hostname);
    void plushQuit();
    void plushFailHost(string hostname);
    void setXmlRpcClientUrl(string clientUrl);
}
class PlushXmlRpcCallback extends XmlRpcClient {
    void sendPlanetLabSlices();
    void sendSliceNodes(string slice);
    void sendAllPlanetLabNodes();
    void sendApplicationExit();
    void sendHostStatus(string host);
    void sendBlockStatus(string block);
    void sendResourceMatching(HashMap matching);
}
```

Figure 4: Plush XML-RPC API.

Implementation Details

Plush is a publicly available software package consisting of over 60,000 lines of C++ 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).

In addition to the main C++ codebase, Plush uses several simple perl scripts for interacting with the PlanetLab Central database and bootstrapping resources. 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. Currently Plush is being used to manage applications on

PlanetLab, ModelNet, and Xen virtual machines [5] in our research cluster.

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

Usage Scenarios

One of the primary goals of our work is to build a generic application management framework that supports execution in any execution environment. This is mainly accomplished through the Plush *resource* abstraction. In Plush, resources are computing devices capable of hosting applications, such as physical machines, emulated hosts, or virtual machines. To show that Plush achieves this goal, in this section we take a closer look at specific uses of Plush in different distributed computing environments, including a live deployment testbed, an emulated network, and a cluster of virtual machines.

PlanetLab Live Deployment

To demonstrate Plush's ability to manage the live deployment of applications, we revisit our previous example from the second section and show how Plush manages SWORD [23] on PlanetLab. Recall that SWORD is a resource discovery service that relies on host monitors running on each PlanetLab machine to report information periodically about their resource usage. This data is stored in a DHT (distributed hash table), and later accessed by SWORD clients to respond to requests for groups of resources that have specific characteristics. SWORD is a service that helps PlanetLab users find the best set of resources based on the priorities and requirements specified, and is an example of a long-running Internet service.

The XML application specification for SWORD is shown in Figure 5. Note that this specification could be built using Nebula, in which case the user would

```
<?xml_version="1.0"encoding="utf"−8?>
<plush>
    <project_name="sword">
        <software_name="sword_software" type="tar">
            <package_name="sword.tar" type="web">
                <path>http://plush.ucsd.edu/sword.tar</path>
                <dest>sword.tar</dest>
            </package>
        </software>
        <component_name="sword_participants">
            <rspec>
                <num_hosts_min="10" max="800"/>
            </rspec>
            <resources>
                <resource_type="planetlab" group="ucsd_sword"/>
            </resources>
            <software_name="sword_software"/>
        </component>
        <application_block_name="sword_app_block" service="1"
            reconnect_interval="300">
            <execution>
                <component_block_name="participants">
                    <component_name="sword_participants"/>
                    <process_block_name="sword">
                        <process_name="sword_run">
                            <path>dd/planetlab/run sword</path>
                        </process>
                    </process_block>
                </component_block>
            </execution>
        </application_block>
    </project>
<plush>
```

Figure 5: SWORD application specification.

never have to edit the XML directly. The top half of the specification in Figure 5 defines the SWORD software package and the component (resource group) 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, a range of acceptable values is used rather than a single number. In this case, as long as 10 hosts are available, Plush will continue managing SWORD. Since the max is set to 800, Plush will not look for more than 800 resources to host SWORD. Since PlanetLab contains less than 800 hosts, this means that SWORD will attempt to run on all PlanetLab resources.

The lower half of the application specification defines the application block, component block, and process block that describes the SWORD execution. The application block contains a few key features that help Plush react to failures more efficiently for long-running services. 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, since there is no reason to synchronize the execution across all hosts. Further, if a process fails when the service attribute has been specified, the controller attempts to restart SWORD on that host without aborting the entire application.

The reconnect_interval specifies the period of time the controller waits before rerunning the resource discovery and acquisition unit. For long running services, hosts often fail and recover during execution. The reconnect_interval attribute tells the controller to check for new hosts that have come alive since the last run of the resource discovery unit. The controller also unsets any hosts that had previously been marked as ''failed'' at this time. This 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.

Evaluating Fault Tolerance

To demonstrate Plush's ability to automatically recover from host failures for long running services, we ran SWORD on PlanetLab with 100 randomly chosen hosts, as shown in Figure 6. 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 they each begin downloading the SWORD software package (38 MB).

It takes approximately 1000 seconds for all hosts to successfully download, install, and start SWORD. At time $t = 1250s$, we kill the SWORD process on 20 randomly chosen hosts to simulate host failure. Normally, Plush would automatically try to restart the SWORD process on these hosts. However, we disabled this feature to simulate host failures and force a rematching. The remote Plush clients notify the controller that the 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 $t = 2200s$, the service is restored on 100 machines.

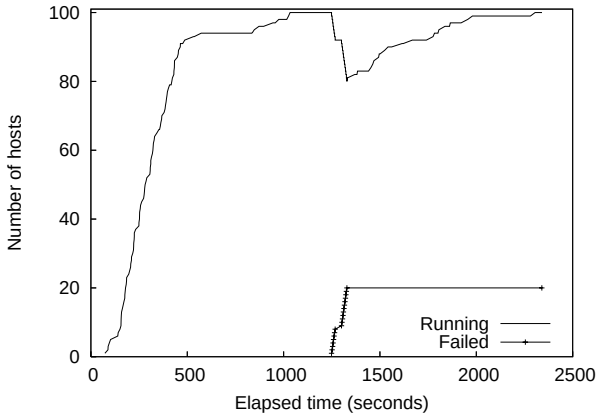

Figure 6: SWORD running on 100 randomly chosen PlanetLab hosts. At t=1250 seconds, we fail 20 hosts. The Plush controller finds new hosts, who start the Plush client process and begin downloading and installing the SWORD software. Service is fully restored at approximately t=2200 seconds.

Using Plush to manage long-running services like SWORD alleviates the burden of manually probing for failures and configuring/reconfiguring hosts. Further, Plush interfaces directly with the PlanetLab Central API, which means that users can automatically add hosts to their slice and renew their slice using Plush. This is beneficial since services typically want to run on as many PlanetLab hosts as possible, including any new hosts that come online. 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 maintain and monitor a service running on hundreds of resources scattered around the world.

ModelNet Emulation

Aside from PlanetLab resources, Plush also supports running applications on virtual hosts in emulated

environments. In this section we discuss how Plush supports using ModelNet [29] emulated resources to host applications. In addition, we will discuss how a batch scheduler uses the Plush programmatic interface to perform remote job execution.

Mission is a simple batch scheduler used to manage the execution of jobs that run on ModelNet in our research cluster. ModelNet is a network emulation environment that consists of one or more Linux edge nodes and a set of FreeBSD core machines running a specialized ModelNet kernel. The code running on the edge hosts routes packets through the core machines, where the packets are subjected to the delay, bandwidth, and loss specified in a target topology. A single physical machine hosts multiple ''virtual'' IP addresses that act as emulated resources on the Linux edge hosts.

To setup the ModelNet computing environment with the target topology, two phases of execution are required: deploy and run. Before running any applications, the user must first *deploy* the desired topology on each physical machine, including the FreeBSD core. The deploy process essentially instantiates the emulated hosts, and installs the desired topology on all machines. Then, after setting a few environment variables, the user is free to *run* applications on the emulated hosts using virtual IP addresses just as applications are run on physical machines using real IP addresses.

A single ModelNet experiment typically consumes almost all of the computing resources available on the physical machines involved. Thus, when running an experiment, it is essential to restrict access to the machines so that only one experiment is running at a time. Further, there are a limited number of FreeBSD core machines running the ModelNet kernel available, and access to these hosts must also be arbitrated. Mission is a batch scheduler developed locally to help accomplish this goal by allowing the resources to be efficiently shared among multiple users. ModelNet users submit their jobs to the Mission job queue, and as the machines become available, Mission pulls jobs off the queue and runs them on behalf of the user, ensuring that no two jobs are run simultaneously.

A Mission job submission has two components: a Plush application specification and resource directory file. For ModelNet, the directory file contains information about both the physical and virtual (emulated) resources on which the ModelNet experiment will run. In the resource directory file, some entries include two extra parameters, ''vip'' and ''vn'', which define the virtual IP address and virtual number (similar to a hostname) for the emulated resources. 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.

The application specification submitted to Mission contains two component blocks separated by a synchronization barrier. The first component block describes the processes that run on the physical machines during the deployment phase (where the emulated topology is instantiated). The second component block defines the processes associated with the target application. 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. This sets the appropriate ModelNet environment variables, ensuring that all commands run on that client on behalf of the user inherit those settings.

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

Plush jobs are currently being submitted to Mission on a daily basis at UCSD. These jobs include experimental content distribution protocols, distributed model checking systems, and other distributed applications of varying complexity. Mission users benefit from Plush's automated execution capabilities. Users simply submit their jobs to Mission and receive an email when their task is complete. They do not have to spend time configuring their environment or starting the execution. Individual host errors that occur during execution are aggregated into one message and returned back to the user in the email. Logfiles are collected in a public directory on a common file system and labeled with a job ID, so that users are free to inspect the output from individual hosts if desired.

Virtual Machine Deployment

In all of the examples discussed above, the pool of resources available to Plush is known at startup. In the PlanetLab examples, Plush uses slice information to determine the set of user-accessible hosts. For ModelNet, the emulated topology includes specific information about the virtual hosts to be created and this information is passed to Plush in the directory file. We next describe how Plush manages applications in environments without fixed sets of machines, but rather underlying capabilities to create and destroy resources on demand.

Shirako [16] 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 [21] is a virtual machine scheduling system for cluster environments. It allows users to create their own virtual machines or clusters. When a user requests a virtual machine, Usher uses data collected by virtual machine monitors to make informed decisions about when and where the virtual machine should run.

We have extended Plush to interface with both Shirako and Usher. Through its XML-RPC interface, Plush interacts with the Shirako and Usher servers. As resources are created and destroyed, the resource pool in Plush is updated to include the current set of leased resources. Using this dynamic resource pool, Plush manages applications running on potentially temporary virtual machines in the same way that applications are managed in static environments like PlanetLab. Thus, using the resource abstractions provided by Plush, users are able to run their applications on PlanetLab, ModelNet, or on clusters of virtual machines without ever having to worry about the underlying details of the environment.

To support dynamic resource creation and management, we augment the Plush application specification with a description of the desired virtual machines as shown in Figure 7. Specifically, the Plush application specification needs to include information about the desired attributes of the resources so that this information can be passed on to either Shirako or Usher. Shirako and Usher currently create Xen [5] virtual machines (as indicated by the ''type'' flag in the resource description) with the CPU speed, memory, disk space, and maximum bandwidth specified in the resource request. As the Plush controller parses the application specification, it stores the resource description. Then when the create resource command is issued either via the terminal interface or programmatically through XML-RPC, Plush contacts the appropriate Shirako or Usher server and submits the resource request. Once the resources are ready for use, Plush is informed via an XML-RPC callback that also contains contact information about the new resources. This callback updates the Plush resource pool and the user is free to start applications on the new resources by issuing the run command to the Plush controller.

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

The integration of Shirako and Plush allows users to seamlessly leverage the functionality of both systems. While Shirako provides a web interface for creating and destroying resources, it does not provide an interface for using the new resources, so Shirako users benefit from the interactivity provided by the Plush shell. Researchers at Duke are currently using

```
<?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</server>
                </shirako>
            </rspec>
            <resources>
                <resource_type="ssh" group="shirako"/>
            </resources>
        </component>
    </project>
</plush>
```

Figure 7: Plush component definition containing Shirako resources. The resource description contains a lease parameter which tells Shirako how long the user needs the resources.

Plush to orchestrate workflows of batch tasks and perform data staging for scientific applications including BLAST [3] on virtual machine clusters managed by Shirako [14].

Related Work

The functionality that Plush provides is related to work in a variety of areas. With respect to remote job execution, there are several tools available that provide a subset of the features that Plush supports, including cfengine [9], gexec [10], and vxargs [20]. 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 [31] and GCEShell [22]. 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.

In addition to remote job execution tools and distributed shells, projects like the PlanetLab Application Manager (appmanager) [15] and SmartFrog [13] 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 [13] 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.

There are also several commercially available products that perform functions similar to Plush. Namely, Opsware [24] and Appistry [4] provide software solutions for distributed application management. Opsware System 6 allows customers to visualize many aspects of their systems, and automates software management of complex, multi-tiered applications. The Appistry Enterprise Application Fabric strives to deliver application scalability, dependability, and manageability in grid computing environments. In comparison to Plush, both of these tools focus more on enterprise application versioning and package management, and less on providing support for interacting with experimental distributed systems.

The Grid community has several application management projects with goals similar to Plush, including Condor [8] and GrADS/vGrADS [7]. 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 [7] 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.

Within the realm of workflow management, there are tools that provide more advanced functionality than Plush. For example, GridFlow [11], Kepler [19], and the other tools described in [32] 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.

Lastly, the Globus Toolkit [12] is a framework for building Grid systems and applications, and is perhaps the most widely used software package for Grid development. Some components of Globus provide a similar functionality as Plush. With respect to our application specification language, the Globus Resource Specification Language (RSL) provides an abstract language for describing resources that is similar in design to our language. The Globus Resource Allocation Manager (GRAM) processes requests for resources, allocates the resources, and manages active jobs in Grid environments, providing much of the same functionality as Plush does. The biggest difference between Plush and Globus is that Plush provides a user-friendly shell interface where users directly interact with their applications. Globus, on the other hand, is a framework, and 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.

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 and acquisition, software installation, process execution, and failure recovery 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 new relaxed synchronization primitives that help applications achieve good throughput even in unpredictable wide-area conditions where traditional synchronization primitives are too strict to be effective.

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

Although Plush has been in development for over three years now, we still have some features that need improvement. One important area for future enhancements is error reporting. Debugging applications is inherently difficult in distributed environments. We try to make it easier for researchers using Plush to locate and diagnose errors, but 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, and ultimately simplify the task of debugging distributed applications in volatile environments.

Author Biographies

Jeannie Albrecht is an Assistant Professor of Computer Science at Williams College in Williamstown, Massachusetts. She received her Ph.D. in Computer Science from the University of California, San Diego in June 2007 under the supervision of Amin Vahdat and Alex C. Snoeren.

Ryan Braud is a fourth-year doctoral student at the University of California, San Diego where he works under the direction of Amin Vahdat in the Systems and Networking research group. He received his B.S. in Computer Science and Mathematics from the University of Maryland, College Park in 2004.

Darren Dao is a graduate student at the University of California, San Diego where he works under the direction of Amin Vahdat in the Systems and Networking research group. He received his B.S. in Computer Science from the University of California, San Diego in 2006.

Nikolay Topilski is a graduate student at the University of California, San Diego where he works under the direction of Amin Vahdat in the Systems and Networking research group. He received his B.S. in Computer Science from the University of California, San Diego in 2002.

Christopher Tuttle is a Software Engineer at Google in Mountain View, California. He received his M.S. in Computer Science from the University of California, San Diego in December 2005 under the supervision of Alex C. Snoeren.

Alex C. Snoeren is an Assistant Professor in the Department of Computer Science and Engineering at the University of California, San Diego. He received his Ph.D. in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology in 2003 under the supervision of Hari Balakrishnan and M. Frans Kaashoek.

Amin Vahdat is a Professor in the Department of Computer Science and Engineering and the Director of the Center for Networked Systems at the University of California, San Diego. He received his Ph.D. in Computer Science from the University of California, Berkeley in 1998 under the supervision of Thomas Anderson. Before joining UCSD in January 2004, he was on the faculty at Duke University from 1999-2003.

Bibliography

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

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

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

[4] *Appistry*, http://www.appistry.com/ .

[5] Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, ''Xen and the Art of Virtualization,'' *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2003.

[6] Bavier, A., 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,'' *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[7] Berman, F., H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, B. Liu, X. Liu, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, C. Mendes, A. Olugbile, M. Patel, D.

Reed, Z. Shi, O. Sievert, H. Xia, and A. YarKhan, "New Grid Scheduling and Rescheduling Methods in the GrADS Project," *International Journal of Parallel Programming (IJPP)*, Vol. 33, Num. 2-3, 2005.

[8] Bricker, A., M. Litzkow, and M. Livny, "Condor Technical Summary," *Technical Report 1069*, University of Wisconsin-Madison, CS Department, 1991.

[9] Burgess, M., "Cfengine: A Site Configuration Engine," *USENIX Computing Systems*, Vol. 8, Num. 3, 1995.

[10] Chun, B., *gexec*, http://www.theether.org/gexec/ .

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

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

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

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

[15] Huebsch, R., *PlanetLab Application Manager*, http://appmanager.berkeley.intel-research.net .

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

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

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

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

[20] Mao, Y., *vxargs*, http://dharma.cis.upenn.edu/planetlab/vxargs/ .

[21] McNett, M., D. Gupta, A. Vahdat, and G. M. Voelker, "Usher: An Extensible Framework for Managing Clusters of Virtual Machines," *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*, 2007.

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

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

[24] *Opsware*, http://www.opsware.com/ .

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

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

[27] *Plush*, http://plush.ucsd.edu .

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

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

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

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

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