Distributed Application Management Using Plush

Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat

University of California, San Diego
{*jalbrecht, ctuttle, snoeren, vahdat*}@*cs.ucsd.edu*

Abstract. Although a number of solutions exist for subtasks of application deployment and monitoring in large-scale, distributed environments, few tools provide a unified framework for distributed application management. Many of the existing tools address the management needs of a single class of applications or services that run in a specific environment and are not extensible enough to be used for other applications. In this paper, we discuss the design and implementation of Plush, a fully configurable application control infrastructure designed to meet the general requirements of several different classes of distributed applications. The paper discusses how users specifically define the flow of control needed using application building blocks provided by Plush. We also take a closer look at a few specific distributed applications to gain an understanding of how Plush provides support for each.
Keywords: application management, peer-to-peer, workflow, grid, PlanetLab

1 Introduction

Installing, configuring, and executing a distributed application on federated computation infrastructures such as PlanetLab [3, 18] and the Grid [10] is currently a time-consuming and error-prone process. After the initial configuration, the application must address the inevitable failures endemic to such environments. Hence, 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: appropriate computing resources must be discovered and acquired, files distributed, and each machine appropriately configured (and re-configured when operating conditions change). It is not surprising, then, that a number of tools have been developed to address various aspects of the process, but no solution has yet been presented that flexibly automates the application deployment and management process.

Presently, most applications that successfully exploit the resources available in these heterogeneous distributed environments take one of two approaches. On PlanetLab, most researchers and service operators address deployment and monitoring in an *ad hoc*, application-specific fashion. Grid researchers, on the other hand, leverage one or more toolkits for application development and deployment. Custom implementations like those employed on PlanetLab vary greatly in their sophistication, but all share the same shortcoming: they must be rewritten when application requirements or network conditions change. Conversely, while Grid toolkits deliver significant functionality, they require tight integration with not only the infrastructure, but the application itself. Hence, applications must generally be custom tailored for a given toolkit.

We present Plush, a framework of tools that, when taken together, provide a unified environment to support the distributed application design and deployment life cycle.

Plush users describe distributed applications using an extensible XML specification language. Unlike typical Grid 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. This functionality can be used, for example, to specify a particular resource discovery or allocation tool to use during application deployment. In addition, Plush provides extensive failure management support to automatically adapt to failures in both the application and the underlying computational infrastructure. Critically, applications do not need to be written to a specific API; instead, Plush interacts with applications through standard POSIX process-control techniques.

Plush manages resource discovery and acquisition, software distribution, and process execution in a fully configurable fashion. Users describe their applications using combinations of Plush "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 can perform a number of user-configurable recovery actions, such as restarting the application, automatically reconfiguring it, or even searching for alternate resources. For applications requiring wide-area synchronization, Plush provides several efficient synchronization primitives. In particular, Plush provides two new barrier semantics, which relax traditional barrier semantics for increased performance and robustness in failure-prone environments.

The remainder of this paper discusses the architecture of Plush. We motivate the design in Section 2 by enumerating a set of general requirements for managing distributed applications. Section 3 details the design and implementation of Plush, and Section 4 discusses how we address fault tolerance and scalability. We provide specific application case studies and uses of Plush in Section 5 before discussing related work in Section 6, and wrapping up in Section 7.

2 Application controller requirements

The low-level details for managing distributed applications depend on the characteristics of the target application. For example, short-lived applications and network experiments prefer powerful machines and abort when failures are detected, long-running services prefer reliable machines and attempt to silently recover from failures, and Grid applications prefer powerful machines and need the ability to detect both slow and failed machines. But at a high level, the requirements for each example are largely similar. Rather than reinvent the same infrastructure for each application separately, we set out to identify commonalities across all classes of distributed applications, and build an application control infrastructure that supports the general requirements of each.

Application Specification. A generic application controller must allow the user to customize control flow for each application. This *application specification* identifies all aspects of the execution and environment needed to successfully deploy, manage, and maintain the application. It describes the software required to run the application—including how to access and install it—and processes that will run on each machine. To support a variety of environments, the application description language should be extensible so that it is easy to add environment specific details if desired. The language must also be able to define detailed resource specifications, including how the resources should be acquired and any credentials required for authentication or authorization.

The complexity of distributed applications varies greatly from simple, single-process applications to complex, parallel applications. Thus, an application controller must support arbitrary process inter-dependencies. Since many applications require synchronization across machines, the application specification language should also describe application synchronization requirements. Similarly, the ability to distribute 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. In addition to all of these requirements, the application specification language must be simple enough for users to understand, yet expressive enough to run complex scenarios.

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

Resource discovery often interacts directly with resource acquisition systems. Resource acquisition involves obtaining a lease or permission to use the desired resources. Acquisition can be accomplished in a number of ways. For example, if advanced resource reservations are required, such as in a batch pool, the resource acquisition mechanism is responsible for submitting a resource request on the user's behalf and subsequently obtaining a lease from the scheduler. Some environments may not require advanced reservations for use, and therefore do not require additional steps for acquisition. In local site clusters, the application control infrastructure may implement its own scheduling mechanism.

Application Deployment. Once a set of resources have been located, the next step involves preparing the physical resources with the correct software and data files, and then running the executable to start the application. This involves copying, unpacking, and installing the software on the target hosts. The system must handle a variety of different file transfer protocols for each environment, and must react to failures that occur during the transfer of software or in starting individual executables.

One important aspect of application deployment is ensuring that the correct number of resources are running compatible versions of the required software. Ensuring that a minimum number of hosts are available for a distributed computation may involve requesting new resources from the resource discovery and acquisition phase to compensate for failures that occur at startup. Further, many applications require some form of loose synchronization across hosts to guarantee that various phases of computation start at approximately the same time.

Application Maintenance. Perhaps the most difficult requirement for managing distributed applications is monitoring an application after it has been started. Monitoring involves probing the hosts for failure due to network outages or hardware malfunctions, querying the application for indications of failure during execution, and providing hooks into application-specific code for observing the progress of an execution. This allows for much more specific error reporting, which simplifies the debugging process for users. The goal of application maintenance is to maintain application liveness, provide detailed error information, and achieve forward progress in the face of failures.

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 machines, but only 48 can be contacted at a certain point in time, the application controller should contact the user, and, if possible, adapt the application appropriately to continue executing with only 48 machines. Similarly, different applications have different policies 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 be aborted across all hosts.

3 Design and implementation

Given the requirements presented in Section 2, we now describe Plush, an extensible application controller for large-scale distributed systems. The hosts involved in a Plush-managed application form a peer-to-peer overlay network. One overlay participant, the *controller*, parses user input and sends messages on behalf of the user to the remaining participants, the *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 on machines spread across the network, and perform actions based on instructions received from the controller.

Figure 1(a) shows an overview of the Plush controller architecture[1]. The architecture consists of three main sub-systems: the application specification, core functional units, and user interface. Plush parses the application specification to store data structures and objects specifically defined by the user. 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 hosts to query the status of the distributed application. The user interface subsystem provides users with the functionality to interact with the other parts of the architecture, allowing the user to maintain and manipulate the application during execution.

3.1 Application specification

One requirement for controlling a distributed application is maintaining the control flow, defined in the Plush application specification. Developing a complete, yet accessible, specification description was one of the significant challenges in this work. Our approach, which has evolved over the past two years, consists of combinations of five different abstractions or blocks:

1. **Process blocks** - The processes executed on remote hosts. The process abstraction includes parameters, path variables, runtime environment details, file and process I/O information, and the specific commands needed to start a process.

[1] Although we do not include a detailed overview of the client architecture, the client architecture is symmetric to the controller with only minor differences in functionality.

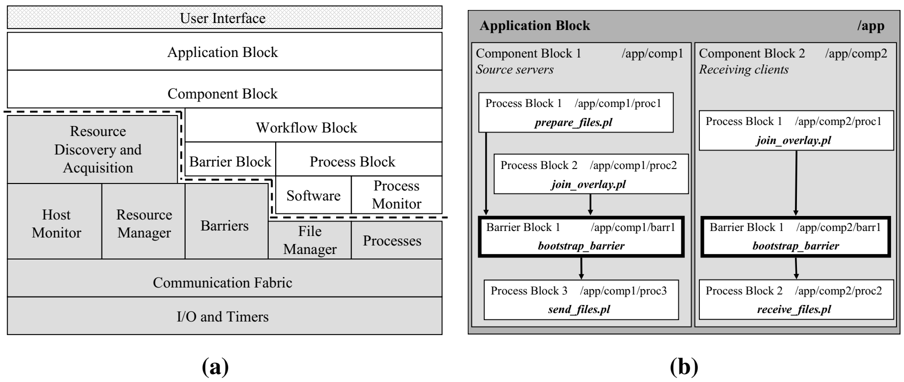

Fig. 1. (a) The architecture of Plush. A box shown above another box indicates that the top box requires the functionality provided by the lower box for successful operation. (b) Example file distribution application comprised of application, component, process, and barrier blocks in Plush.

2. **Barrier blocks** - Barriers synchronize the various phases of execution within a distributed application.
3. **Workflow blocks** - The flow of data in a distributed computation, including how the data should be processed. Workflow blocks may contain process and barrier blocks. For example, a workflow block might describe a set of input files over which a given process or barrier block will iterate during execution.
4. **Component blocks** - The groups of physical resources required to run the application. This includes expectations specific to a set of metrics. In the case of compute nodes, for example, these metrics might include requirements for load and free memory. 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** - High-level information about a distributed application. This includes one or many component blocks, as well as attributes to help automate failure recovery.

To better illustrate the use of these blocks in Plush, consider building the specification for the simple file distribution application shown in Figure 1(b). This simple application consists of two groups of machines. One group is the server machines that store the files, and the second group is the receiving client machines that need to get the files from the servers. The goal of the application is to experiment with the use of an overlay network to send files from the source to the receiving clients using some new file distribution protocol. In this example, all senders and receivers must join the overlay before any transfers begin. Also, the servers must prepare the files for transfer before the receivers can start.

The first step in building the corresponding application specification in Plush is to define an application block. The application block defines specific characteristics including the general liveness properties of the application, which help to determine the default behaviors to take during failure recovery. For this example, we choose the default behavior of "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 if a single failure occurs.

The application block also describes the groups of resources (*i.e.*, physical machines) required to run the application, encoded as component blocks. The application consists of a set of servers and a set of clients, and two separate component blocks describe the two groups of machines. The server component block defines the location and installation instructions for the server software, and includes authentication information required to access the resources. Similarly, the client component block defines the client software package. In our example, for instance, it may be desirable to require that all machines in the server group have a processor speed of at least 1 GHz. Machine-specific requirements are specified in component blocks.

Within each component block, a combination of workflow, process, and barrier blocks describe the computation that will occur on each machine in the set. Though workflow blocks are not used in our example, workflow blocks are used in applications where data files must be distributed and iteratively processed. We will consider an example employing a workflow block in Section 5.

Plush process blocks describe the specific commands required to execute the application. Most process blocks depend on the successful installation of **software** packages defined in the component blocks. Users specify the commands required to start a given process, and actions to take upon process exit. The exit policies create a Plush **process monitor** that oversees the execution of a specific process. Our example defines several process blocks. In the server component, process blocks define processes for preparing the files, joining the overlay, and sending the files. Similarly, the client 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 has completed, barrier blocks define loose synchronization semantics between process and workflow blocks. In our example, a barrier ensures that all clients and servers join the overlay before beginning the file transfer. Notice that although each barrier block is uniquely defined within the component block, it is possible for the same barrier to be referenced in multiple component blocks. For our example, both barrier blocks refer to the same barrier, which means that the application will wait for all clients and servers to reach the `bootstrap_barrier` before allowing either component to start sending or receiving files.

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 and successor tags in Plush, users specify the flow of control and define whether processes run in parallel or sequentially. Internally, Plush stores the blocks in a hierarchical data structure, and references specific blocks in a manner similar to referencing absolute paths in a file system. Figure 1(b) shows the unique path names for each block from our file distribution example. Plush also simplifies coordination among remote hosts, since each host maintains an identical local copy of the application specification.

3.2 Core functional units

In addition to the abstractions defined by the user within the application specification, Plush contains a core set of functional units that perform the operations required for the remaining requirements outlined in Section 2. These units are shown as shaded boxes below the dotted line in Figure 1(a). The functional units manipulate the objects defined in the application specification to manage distributed applications.

Starting at the highest level, the Plush **resource discovery and acquisition** unit uses the resource requirements in the component blocks to locate 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 demand. To determine this matching, Plush may either call an existing external service, such as SWORD [17] or MDS [1], or use a simple internal default matcher. 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 find the best matching. Upon acquiring a resource, a Plush **resource manager** stores the lease, token, or any necessary user credential needed for accessing that resource to allow Plush to perform actions on behalf of the user in the future.

The remaining functional units in Figure 1(a) are responsible for application deployment and maintenance. These units are used to connect to resources, install the required software, start the execution, and monitor the execution for failures. One important functional unit used for these operations is the Plush **partial barrier** abstraction. Traditional barriers [14] are not well suited for volatile, wide-area network conditions; the semantics are simply too strict. In order to achieve better resilience in the presence of failures, Plush extends traditional barrier semantics with two new relaxations. The first relaxation primitive, *early entry*, allows hosts that reach a barrier to be released before all hosts have arrived, and thus enter a "critical section" of activity early. This prevents progress from stalling due to a small subset of delayed hosts. The second primitive, *throttled release*, allows the user to control the rate of release from a barrier. These relaxed barrier semantics target distributed applications willing to tolerate less strict synchronization guarantees to achieve better performance.

Figure 2 shows part of the Plush partial barrier API. When defining a barrier, the application specifies the barrier's name, the maximum number of hosts expected to arrive at the barrier, the amount of time the application should wait for additional hosts to arrive, the percentage of hosts required for correctness, and the minimum amount of time to wait before releasing a barrier early. These values allow the barrier manager (who is also the Plush controller) to perform early entry. This primitive is especially useful, for example, if there is a failure that causes a network partition. Rather than wait an infinite amount of time for the remaining hosts to arrive at the barrier, the early entry primitive will release the hosts that have already arrived and thus continue to make progress even in adverse conditions. The remaining methods are used to implement the throttled release primitive. As hosts reach the barrier, they call `enter()` to notify the barrier manager of their arrival. `setThrottleReleasePercent()` specifies what percentage of hosts should be released at once, rather than releasing all hosts simultaneously. `setThrottleReleaseCount()` defines a specific number of hosts

```
class PlushBarrier {
    Barrier(string name, int max, int timeout, int percent, int minWait);
    void enter(string label, string Hostname);
    void setThrottleReleasePercent(int percent);
    void setThrottleReleaseCount(int count);
    void setThrottleReleaseTimeout(int timeout);
}
```

Fig. 2. Plush partial barrier API specification. The methods shown relax traditional barrier semantics for better performance in volatile, wide-area network conditions.

for release. Lastly, `setThrottleReleaseTimeout()` specifies the frequency of release. More information about partial barriers can be found in [2].

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. Processes require information about the runtime environment needed for an execution including the working directory, path, command line arguments, environment variables, file I/O, and the command itself.

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 currently a star, although we also provide support for tree topologies for increased scalability. In the case of a star topology, all clients connect directly to the controller. 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 enter 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. They fire at specific instances of time, rather than waiting on a queue for an unknown amount of time.

3.3 User interface

Plush streamlines the develop-deploy-debug cycle for distributed application development through a simple terminal interface where users can deploy, run, monitor, and debug their distributed applications running on hundreds of remote machines. In many ways, Plush combines the functionality of a distributed shell with the power of an application controller, to provide a robust execution environment for users to run their applications. From a user's standpoint, the Plush terminal looks like a shell. Plush supports several commands for monitoring the state of an execution, as well as commands for modifying the current application specification. Table 1 shows a subset of the available commands.

Table 1. Plush terminal commands

Command	**Description**
`load <filename>`	Read an XML project file
`connect <hostname>`	Start and connect to a Plush client on a remote host
`disconnect`	Close all open client connections
`info control`	Print the controller's state information
`run`	Start executing the application in the active project
`shell <quoted string>`	Run "quoted string" as a shell command on all hosts

In addition to the terminal interface, users can also interact with hosts running Plush-managed applications using a web interface. This interface presents detailed information about processes, file transfers, host monitoring, and application status. The web interface provides a friendlier front-end to the terminal commands that Plush supports. Most of these commands can be executed by simply clicking the various links and buttons on the web pages.

In Figure 1(a), the user interface is shown above all other parts of Plush. In reality, the user can interact with every box shown in the figure through simple terminal commands. For example, the user can force the resource discovery and acquisition unit to find a new set of resources using a terminal command. We designed Plush in this way to give the user maximum control over the application. At any point, the user can override a default Plush behavior. The overall effect is a customizable application controller that has the ability to support a variety of distributed applications.

3.4 Running an application

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 in the form of an XML document to the Plush controller. The XML document is a representation of the application specification block hierarchy previously described. We will consider a specific XML document in Section 5. The controller parses the specification, and internally creates the objects shown above the dotted line in Figure 1(a).

After parsing the application specification, the controller runs the resource discovery and acquisition phase 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 use 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 control overlay.

If a Plush client agrees to run the application, the client sends a `JOIN` message back to the controller accepting the invitation. Next, the controller sends a `PREPARE` message to the new client, which contains a copy of the application specification (XML representation). The client parses the application specification, starts a local host monitor, sends a `PREPARED` message back to the controller, and waits for further instruction from the controller. 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 scheme (*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 in the application.

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, kills the Plush client processes, and disconnects the hosts from the overlay by closing the TCP connection.

The set of clients managed by a single Plush controller is not limited to one platform. Plush is written in C++, and runs on most UNIX-based platforms, including Linux, FreeBSD, and Mac OS X. The same controller has the ability to manage clients across all supported platforms. Currently, Plush supports execution on PlanetLab, ModelNet [20], and in local UNIX clusters; support for Grid environments is expected to be completed in the near future.

4 Fault tolerance and scalability

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

4.1 Fault tolerance

Plush must be robust to the variety of failures that occur during application execution. When designing Plush, we aimed to provide the functionality needed to detect and recover from most failures without ever needing to involve the user running the application. Rather than enumerate all possible failures that may occur, we will discuss how we handle 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 entire application, or aborting the entire application.

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

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 certainly 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 will try 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 will rerun 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 and the application description specifies a fixed number of required hosts, Plush then finally 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 can tolerate, it may mark the host as failed and attempt to rematch. This functionality is a preference specified at startup. Currently Plush monitors host-level metrics including CPU load and available memory. This technique could be extended to encompass more sophisticated application-level expectations of host viability [19].

Controller failures. Because the controller is responsible for managing the flow of control across all connected clients, a failure at the controller is difficult to recover. 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 in the list 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 a version of this architecture, we are currently exploring other possibilities for handling faults at the controller.

4.2 Scalability

In addition to fault tolerance, an application controller designed for large-scale environments must scale to hundreds or even thousands of participants. 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 with varying levels of scalability and performance.

By default, all Plush clients connect directly to the controller forming a star topology. This architecture scales to approximately 300 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.

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 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

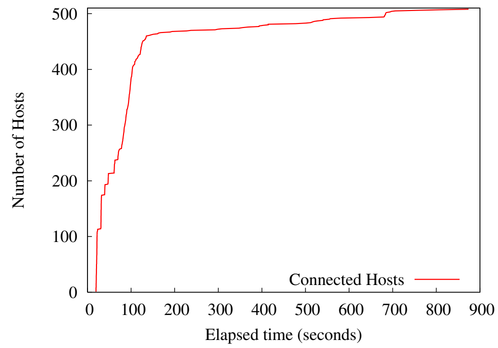

Fig. 3. Plush connecting to 500 clients on PlanetLab. Since it is often difficult to find 500 usable machines on PlanetLab, in this example we use resources from two slices.

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 virtual hosts, as well as an application running on 500 PlanetLab clients, as shown in Figure 3. In this example, we show how long it takes Plush to connect to 500 PlanetLab clients. Since the usable number of machines on PlanetLab is often less than 500, we combine the resources from two PlanetLab slices to eventually achieve our goal of 500 clients. This means that in some instances there are actually two client processes running on different ports on the same physical machine. Further, the default matcher is used in this example, so the machines are chosen randomly. With a smarter resource discovery mechanism, the time to locate 500 usable machines may be reduced. We believe that Plush has the ability to scale by perhaps another order of magnitude with the current implementation and architecture.

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 can have 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 can be 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.

5 Example applications

In this section, we take a closer look at two specific applications that are run on PlanetLab to demonstrate the versatility and flexibility of Plush as a distributed application controller. We also describe how Plush is currently being used within a batch scheduler to manage remote job execution in a local ModelNet cluster.

5.1 Managing EMAN on PlanetLab

EMAN [8] is a publicly available software package that is used for 3D particle reconstruction. EMAN starts with a set of 2D electron micrograph images as input, and runs a series of sequential and parallel computations on the images to reconstruct the 3D model of the original particle. The computationally intense portion of the execution is the refinement stage. This stage is run repeatedly on the 2D images until the desired level of detail is achieved in the 3D model. Refinement can be run in parallel on multiple machines to improve performance. To do this, the raw images are split up, and each machine operates on a fraction of the images. The results are later combined to produce the 3D model. EMAN is an example of a Grid-style parallel application.

The parallel refinement stage is a common example of a workflow application in Grid environments. To demonstrate how workflow applications operate in Plush, we will now show how Plush is used to run a single round of the refinement parallel computation in EMAN. For simplicity, we do not show the sequential portions of the EMAN application that are typically run on a single machine.

The first step in running EMAN using Plush is to create the application specification in an XML document. An example XML specification for EMAN is shown in Figure 4. The XML specification contains two main sections of interest. The top section consists of object definitions for software and components. The lower section consists of the Plush application specification blocks that contain these objects. By separating the definition of the objects from the block containers that they are used in, we believe it makes it easier to understand the flow of control within the application block.

Starting at the top of the XML document, the software definition specifies the URL used to locate the required software, the destination file name on the remote host, the type of software package (`tar`), and what file transfer method is desired (in this case, "web" implies `wget` should be used). The component definition specifies the number of hosts required, what software is required on those hosts, and what resource pool to use for discovery and acquisition. Since a different resource discovery service is not specified, the Plush default matcher will be used to find 100 randomly chosen hosts on PlanetLab from the `ucsd_plush` slice.

The remainder of the XML document defines the Plush application specification block hierarchy. The application block contains a component block that refers to the component named `EmanGroup1`. `EmanGroup1` was previously described in the top part of the XML document. The component block also contains a workflow block, which indicates that 100 tasks will be shared among the 100 workers requested in `EmanGroup1`. The workflow block has a process block that contains the actual eman process, consisting of a perl script called `eman.pl` that runs on each remote host. `eman.pl` simply provides a wrapper around the commands provided by the EMAN software package.

```
<?xml version="1.0" encoding="utf-8"?>
<plush>
  <project name="eman_proj">
    <software name="EmanSoftware" type="tar">
      <package name="eman.tar" type="web">
        <path>http://plush.ucsd.edu/eman.tar</path>
        <dest>eman.tar</dest>
      </package>
    </software>
    <component name="EmanGroup1">
      <rspec><num_hosts>100</num_hosts><rspec>
      <software name="EmanSoftware" />
      <resources><resource type="planetlab" group="ucsd_plush" /><resources>
    </component>
    <application_block name="eman_app_block">
      <execution>
        <component_block name="eman_comp_block">
          <component name="EmanGroup1" />
          <workflow_block name="eman_workflow_block" id="eman_wf" num_tasks="100">
            <process_block name="eman_proc_block">
              <process name="eman">
                <path>./eman.pl</path>
                <cmdline>
                  <substitution name="eman_sub" id="eman_wf" type="workflow" flag="--i" />
                </cmdline>
              </process>
            </process_block>
          </workflow_block>
        </component_block>
      </execution>
    </application_block>
  </project>
</plush>
```

Fig. 4. EMAN application specification.

The substitution information in the process definition is used in conjunction with the EMAN Perl script to split the workflow among the hosts. Notice how the workflow block has an `id` attribute that is identical to the `id` attribute in the substitution. In this case, `eman.pl` uses a command line argument to specify the `id` of the task, which is then used to determine what fraction of the data files should be processed by each host. The workflow block substitutes the current task id for the command line argument defined by the "`--i`" flag. For example, the first host in `EmanGroup1` will run "`./eman.pl --i 1`," the second will run "`./eman.pl --i 2`," and so on.

Plush users that manage applications like EMAN have the added benefit of being able to use the Plush barrier abstraction to improve performance. In addition to the user defined parameters for early entry and throttled release, Plush provides automated mechanisms for detecting when the "knee" of the completion curve has been reached. Thus, if a subset of hosts are not operating as quickly as the rest, the tasks assigned to the slow hosts can be reassigned to a faster host that has already completed. Our experiments show that this technique can result in a factor of 3 speedup in EMAN. Details from the EMAN trials and other barrier related experiments are discussed in [2].

5.2 Managing SWORD on PlanetLab

SWORD [17] is a resource discovery tool designed for use on PlanetLab. It 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 allows PlanetLab users to find the best set of resources based on the priorities and requirements specified, and is an example of a long-running Internet service.

The application specification for SWORD is similar to EMAN, except for a few minor differences. Instead of specifying a single value for `num_hosts`, a range of acceptable values is defined, since SWORD is a service that wants to run on as many nodes as possible. Further, when specifying the application block for SWORD, we include special `service` and `reconnect_interval` attributes. The `service` bit tells the Plush controller that SWORD is a long-running service and requires different default behaviors for failure recovery. For example, if a process fails, the controller will attempt to restart SWORD on that host without aborting the whole application. The `service` bit also instructs the controller to periodically probe hosts for liveness. Users can define an application specific liveness monitor within the application block as described in Section 3. Since SWORD is a service, the controller will not wait for all participants to join and install the software before starting everyone simultaneously. Instead, the controller will instruct individual clients to start the application as soon as the client finishes installing the software, since there is no reason to synchronize across all hosts.

The `reconnect_interval` specifies the periodicity that the controller uses for rerunning the resource discovery and acquisition unit. For long running services, hosts are going to fail and recover during execution. The `reconnect_interval` tells Plush to check for new hosts that may have come alive since the last time the resource discovery unit was run. This is the controller's way of "refreshing" the list of available hosts. The controller will continue to search for new hosts unless the maximum value specified in the `num_hosts` tag is achieved.

To demonstrate Plush's ability to recover from host failures, we ran SWORD on PlanetLab with 100 randomly chosen hosts, as shown in Figure 5. The host set includes machines behind DSL links as well as hosts from other continents. When Plush starts the application, it initiates connections to 100 machines and they each begin downloading the SWORD software package (which is 38 MB in size). 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[2]. The remote Plush clients notify the controller that the hosts have failed, and Plush controller begins to find replacements for the failed machines. The replacement hosts join the overlay and start downloading the SWORD software. As before, the replacements are chosen randomly, and low bandwidth/high latency links have a great impact on the time it takes to fully recover from the host failure. At approximately $t = 2200s$, the service is restored on 100 machines.

[2] Normally, Plush would automatically try to restart the SWORD process. However this feature was disabled to simulate host failures and force a rematching.

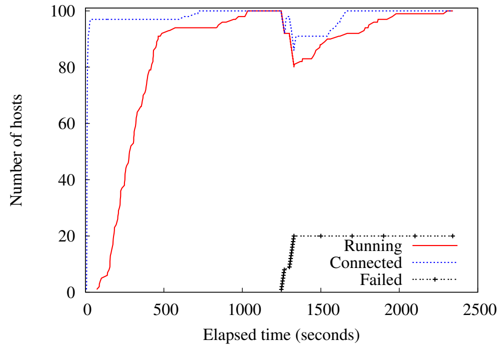

Fig. 5. SWORD running on 100 randomly chosen PlanetLab hosts. At t=1250 seconds, we fail 20 hosts. Plush finds new hosts, who connect to the controller, and begin downloading and installing the software. Service is restored at approximately t=2200 seconds.

5.3 Remote job execution in Mission

Mission is a simple batch scheduler used in our local cluster to manage the execution of jobs that run on ModelNet [20]. ModelNet is a network emulation environment that consists of Linux edge nodes running real code, and a set of FreeBSD core machines running a specialized ModelNet kernel. The code running on the end 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 can host multiple virtual IP addresses on the edge hosts. In order to setup the ModelNet environment, the user must first deploy the topology on each physical machine, including the core. Then, after setting a few environment variables, the user can run applications on the virtual 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 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. Mission is a batch scheduler developed locally that helps accomplish this goal. 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.

Plush is used within Mission for running the jobs and setting up the user environment. A Mission job description is simply a Plush application specification. Using Plush, the deploy and run phases required to use ModelNet can be combined into one application specification with two component blocks. One component block describes the physical machines, and the second component block describes the virtual machines. The process block in the first component block deploys the topology, and the process block in the second component block runs the application on all virtual hosts. Barriers are used to separate the deploy and run phases. To support the virtual hosts using Plush, we must provide a mapping between port numbers on the client machines and ModelNet virtual host information. The mapping is described in a directory file, which is an XML document that supplies Plush with the necessary information to run a ModelNet experiment. Figure 6 shows an example directory file. When the controller starts

```
<?xml version="1.0" encoding="UTF-8"?>
<plush>
 <resource_manager type="ssh">
   <node hostname="sysnet80.ucsd.edu:15400" group="modelnet_deploy"/>
   <node hostname="sysnet81.ucsd.edu:15400" group="modelnet_deploy"/>
   <node hostname="sysnet81.ucsd.edu:15401" vip="10.0.0.1" vn="1" group="modelnet"/>
   <node hostname="sysnet81.ucsd.edu:15402" vip="10.0.0.2" vn="2" group="modelnet"/>
 </resource_manager>
</plush>
```

Fig. 6. ModelNet directory specification. sysnet80 is the FreeBSD core machine. sysnet81 is a Linux edge host that is running 2 virtual hosts.

a Plush client on a virtual host, it specifies extra command line arguments that set the appropriate ModelNet environment variables. This ensures that all commands run on that client on behalf of the user will inherit those settings.

6 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 [6], gexec [11], and vxargs [21]. 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 terminal interface that Plush provides is similar to distributed shell systems such as GridShell [22] and GCEShell [16]. 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 can be customized to work in a variety of environments.

In addition to remote job execution tools and distributed shells, projects like the PlanetLab Application Manager (appmanager) [13] and HP's SmartFrog [12] focus specifically on managing distributed applications. appmanager is a tool designed for PlanetLab that helps users maintain long running services, but does not support short-lived applications as easily. SmartFrog [12] 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 or API for building configurable systems. It does not provide a way to interactively control distributed applications.

The Grid community has several application management projects with goals similar to Plush, including Condor [5] and GrADS/vGrADS [4]. 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 [4] 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 [7] , Kepler [15], and the other tools described in [23] are designed for advanced workflow management in Grid environments. The main difference between these tools and Plush are that they focus solely on workflow management schemes. Thus, while they provide more advanced functionality than Plush with respect to workflow management, they lack much of Plush's functionality for managing other classes of distributed applications that do not contain workflows.

Lastly, the Globus Toolkit [9] 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 can 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.

7 Conclusion

Plush is an extensible application control infrastructure designed to meet the demands of a variety of distributed applications. Through user-defined mechanisms, Plush manages resource discovery and acquisition, software installation, process execution, and failure recovery on behalf of the user. 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 two 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.

Unlike many related tools, Plush does not require applications to adhere to a specific API. In fact, applications managed by Plush do not have to be changed at all. By combining a user-friendly shell interface with a powerful and adaptable application control infrastructure, Plush gives users the control required to successfully manage applications without the hassle that typically comes with running computations on large-scale federated testbeds.

References

1. Globus toolkit monitoring and discovery system: Mds4. `http://www-unix.mcs.anl.gov/~schopf/Talks/mds4SC_nov2004.ppt`.
2. J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Loose Synchronization for Large-Scale Networked Systems. In *USENIX ATC*, 2006.
3. A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating Systems Support for Planetary-Scale Network Services. In *NSDI*, 2004.
4. F. Berman, H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, B. Liu, X. Liu, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, C. Mendes, A. Olugbile, M. Patel, D. Reed, Z. Shi, O. Sievert, H. Xia, and A. YarKhan. New grid scheduling and rescheduling methods in the GrADS project. *IJPP*, 33(2-3), 2005.
5. A. Bricker, M. Litzkow, and M. Livny. Condor Technical Summary. Technical Report 1069, University of Wisconsin–Madison, CS Department, 1991.
6. M. Burgess. Cfengine: a site configuration engine. *USENIX Comp Sys*, 8(3), 1995.
7. J. Coa, S. Jarvis, S. Saini, and G. Nudd. Gridflow: Workflow managament for grid computing. In *CCGrid*, 2003.
8. EMAN. `http://ncmi.bcm.tmc.edu/EMAN/`.
9. I. Foster. A globus toolkit primer, 2005.
10. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. GGF, 2002.
11. gexec. `http://www.theether.org/gexec/`.
12. P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. SmartFrog: Configuration and automatic ignition of distributed applications. In *HP OVUA*, 2003.
13. R. Huebsch. PlanetLab application manager. `http://appmanager.berkeley.intel-research.net`, 2004.
14. H. F. Jordan. A Special Purpose Architecture for Finite Element Analysis. In *ICPP*, 1978.
15. B. Ludscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *CC: P& E*, 2005.
16. M. A. Nacar, M. Pierce, and G. C. Fox. Developing a secure grid computing environment shell engine: Containers and services. *NPSC*, 12:379–390, 2004.
17. D. Oppenheimer, J. Albrecht, D. A. Patterson, and A. Vahdat. Design and implementation tradeoffs for wide-area resource discovery. In *HPDC*, 2005.
18. L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *HotNets*, 2002.
19. P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.
20. A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI*, 2002.
21. vxargs. `http://dharma.cis.upenn.edu/planetlab/vxargs/`.
22. E. Walker, T. Minyard, and J. Boisseau. Gridshell: A login shell for orchestrating and coordinating applications in a grid enabled environment. In *CCCT*, 2004.
23. J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. Technical report, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, 2005.