Managing Distributed Applications Using Gush

Jeannie Albrecht and Danny Yuxing Huang

Williams College, Williamstown, MA

Abstract. Deploying and controlling experiments running on a distributed set of resources is a challenging task. Software developers often spend a significant amount of time dealing with the complexities associated with resource configuration and management in these environments. Experiment control systems are designed to automate the process, and to ultimately help developers cope with the common problems that arise during the design, implementation, and evaluation of distributed systems. However, many of the existing control systems were designed with specific computing environments in mind, and thus do not provide support for heterogeneous resources in different testbeds. In this paper, we explore the functionality of Gush, an experiment control system, and discuss how it supports execution on three of the four GENI control frameworks.

1 Introduction

As network technologies continue to evolve, the need for computing testbeds that allow for experimentation in a variety of environments also continues to rise. In recent years, there has been significant growth in the number of experimental facilities dedicated to this purpose around the world, including GENI in the U.S. [1], FIRE in Europe [2], AKARI in Japan [3], and CNGI in China [4]. These testbeds play a crucial role in the development of the next generation Internet architecture by giving researchers a way to test the performance of new protocols and services in realistic network settings using diverse resources.

While these new testbeds offer many benefits to developers with respect to experimentation capabilities, they also introduce new complexities associated with managing computations running on hundreds of computing devices worldwide. For example, consider the task of running an experiment on one of these testbeds, which involves first installing the required software and then starting the computation on a distributed set of resources. When running a computation on a single resource, it is trivial to download any needed software and start an execution. However, ensuring that hundreds of devices are configured correctly with the required software is a cumbersome task that is further complicated by the heterogeneity—in terms of both hardware and software—of the resources hosting the experiment. Similarly, starting a computation requires synchronizing the beginning of the execution across a distributed set of resources, which is especially difficult in wide-area settings due to the unpredictable changes in network connectivity among the resources involved in the experiment [5].

In addition to configuration and deployment, there are many other challenges involved with keeping an experiment running in distributed environments, such as failure detection and recovery. In experiments that only involve a single resource, monitoring an execution and reacting to failures typically consists of watching a small set of processes and addressing any problems that arise. In experiments involving distributed applications, monitoring an execution consists of watching hundreds of processes running on resources around the world. If an error or failure is detected among these processes, recovering from the problem may require stopping all processes and restarting them again. The difficulties associated with these tasks are frustrating to developers, who end up spending the majority of their time managing executions and coping with failures, rather than developing new optimizations for increased application performance.

Experiment control frameworks are often used to alleviate the burdens associated with installing and executing software in distributed environments. They are designed to simplify the tasks associated with software configuration, resource management, failure detection, and failure recovery. However, many of these frameworks are designed with a single execution environment in mind, and thus are not adaptable to other environments with different resources. This limitation reduces the overall usefulness of the framework, and restricts its use to a single deployment platform. Further, as computers become more ubiquitous and the diversity of network-capable computing devices continues to grow, there is an increasing need for extensible management frameworks that support execution and experimentation in a variety of environments.

In response to the limitations of other application management frameworks, we developed Gush [6], an experiment control system that aims to support software configuration and execution on many different types of resources. Gush leverages prior work with Plush [7,8], and is being developed as part of the GENI project. Gush accomplishes experiment control through an easily-adapted application specification language and resource management system. The resource management system abstracts away the resource-specific details and exports a simple, generic API for adding and removing resources from an application's *resource pool*. The Gush *resource matcher* then uses the resources in the resource pool and the application's requirements as defined in the application specification to create a *resource matching*.

In this paper, we summarize the basic operation of Gush, with an emphasis on PlanetLab [9] resource management and experiment configuration in Section 2. In Section 3, we examine how the Gush resource matcher interacts with different types of resources in GENI (in addition to PlanetLab) to construct valid matchings and run experiments in two other GENI control frameworks: ORCA [10] and ProtoGENI [11]. We then discuss related work in Section 4. The fourth GENI control framework, ORBIT [12], focuses on wireless resources, which are currently not supported by Gush. In Section 5 we discuss how Gush can be extended to support execution in this environment and other wireless environments as well, and make general conclusions.

2 PlanetLab Application Management with Gush

Before discussing how Gush provides support for various types of resources, we first summarize the basic operation of Gush. The purpose of this section is to provide a high-level overview of how Gush works in a typical usage scenario. The design and implementation of Gush greatly leverages our previous work with Plush; however, this paper focuses on GENI-specific extensions to Gush. A detailed discussion of the design and implementation of Plush can be found in [7]. It is important to note that Gush and Plush were both initially designed for the PlanetLab control framework, and the performance and operation of both systems on PlanetLab is largely the same. Thus, this section uses experimentation on PlanetLab as a motivating example. In the next section we discuss how Gush also supports other GENI control frameworks.

Gush is an experiment control framework that aims to simplify the tasks associated with the development, deployment, and maintenance of software running on a distributed set of resources. The two main components in the Gush architecture are the controller and the clients. The Gush controller process is responsible for managing the Gush client processes running on the distributed resources. The controller process is often run on the desktop computer of the Gush user (*i.e.*, the software developer).

The main role of the controller is to receive and respond to input provided by the user and guide the flow of execution on the clients. The clients are lightweight processes that run on specified ports on each resource involved in an experiment. When starting an execution, the controller initiates a separate TCP connection to each client process creating a communication fabric. For the remainder of the execution, the controller sends messages to the clients via the fabric instructing them to run commands and start processes on behalf of the user. The clients also periodically send the controller updates regarding their individual status or in response to failures. Using these status updates, the Gush controller can construct a single, global view of the progress of an experiment.

To manage an experiment using Gush, the user must provide the Gush controller with two pieces of information: an application specification and a resource directory. We discuss each of these components in detail in the following subsections.

2.1 Describing an Experiment in Gush

The *application specification* is an XML file that describes the flow of control for the experiment. In Gush, these are described using a set of "building block" abstractions that describe the required software packages, processes, and desired resources. The blocks can be arbitrarily combined to support a range of experiments in different environments.

Figure 1 shows a sample application specification for a very basic experiment. Starting at the top of the XML, we define the software required for our experiment. The software definitions specify where to obtain the required software, the file transfer method as indicated by the "type" attribute for the package element, and the installation method as indicated by the "type" attribute of the software element. In this particular example, the file transfer method is "web" which means that a web fetching utility such as wget or curl is used by the Gush clients to retrieve the software package from the specified URL. The installation method is "tar." This implies that the software.tar package has been bundled using the tar utility, and installing the package involves running tar with the appropriate extraction arguments.

Moving to the next main section in the XML, we define our experiment's *component*. This is essentially a high-level description of our desired resources. Each component is given a unique name, which is used by the component blocks later to identify which set of resources should be used. Next, the "rspec" element defines "num_hosts," which is the number of resources required in the component. The "software" element within the component specification refers to the "SimpleSoftwareTarball" software package that

```
<?xml version="1.0" encoding="utf-8"?>
<gush>
  <project name="simple">
    <software name="SimpleSoftwareTarball" type="tar">
      <package name="Package" type="web">
        <path>http://sysnet.cs.williams.edu/~jeannie/software.tar</path>
      </package>
    </software>
    <component name="GENIMachines">
      <rspec><num_hosts>20</num_hosts></rspec>
      <software name="SimpleSoftwareTarball" />
      <resources>
        <resource type="planetlab" group="williams_gush"/>
        <resource type="gpeni" group="gpeni_gush"/>
        <resource type="max" group="maxpl_gush"/>
      </resources>
    </component>
    <experiment name="simple">
      <execution>
        <component_block name="compBlock1">
          <component name="GENIMachines" />
          <process_block name="procBlock1">
            <process name="catProc">
              <path>cat</path>
              <cmdline><arg>software.txt</arg></cmdline>
            </process>
          </process_block>
        </component_block>
      </execution>
    </experiment>
  </project>
</gush>
```

Fig. 1. Gush application specification that is used to manage an experiment on 20 GENI resources. This trivial example simply runs “cat software.txt” on each resource.

was previously defined. Lastly, the “resources” element specifies which resource group the Gush controller will use to create a resource matching. In this case we are interested in 20 hosts from one of three GENI resource aggregates (all part of the PlanetLab control framework): PlanetLab [9], GpENI [13], and MANFRED (or MAX) [14]. Specifically, we want to use hosts assigned to the williams_gush, gpeni_gush, or maxpl_gush slices.

Finally, we define the experiment’s execution using XML, and specify which component we want to use. Our simple example contains one component block that maps to our previously defined component comprised of 20 PlanetLab, GpENI, and MAX machines, and one process block consisting of a single process. This process runs the Unix command “cat software.txt” on each of our machines and then exits. (Note that software.txt is contained in software.tar.) More complicated executions are described in a similar fashion. Examples can be found on the Gush website [6].

2.2 Constructing a Resource Pool

In addition to the application specification, the user must also provide Gush with a resource directory. The resource directory is used to define resource pools in Gush, which are simply groupings of resources that are available to the user and are capable of hosting an experiment. The simplest way to define a resource directory in Gush is by creating another XML file (typically called directory.xml) that lists available resources. This file is read by the Gush controller at startup, and internally Gush creates a *Node* object for each specified resource. A Node in Gush contains a username for logging into the resource, a fully qualified hostname, the port on which the Gush client will run, and a group name. The purpose of the group name is to give users the ability to classify resources into different categories based on application-specific requirements. We discuss how this name is used when creating a matching in the next subsection.

The resource file also contains a special section for defining hosts in the PlanetLab control framework. Rather than specifically defining which PlanetLab hosts a user has access to, the directory file instead lists which *slices* are available to the user. In addition to slice names, the user specifies their login to all available aggregate managers (which internally run their own version of the PlanetLab Central server) as well as a mapping ("port_map") from slice names to port numbers. At startup, the Gush controller uses this login information to contact each manager (PLC) directly via XML-RPC using the API specified by the Slice-based Facility Architecture (also called SFA or geniwrapper [15]). Each PLC server returns a list of hostnames that have been assigned to each available slice. The Gush controller uses this information to create a Node object for each host available to the user. The username for these hosts is the associated slice name, and the port is determined by the port_map. The group name is set as the slice name.

Note that for all PlanetLab aggregate managers, Gush assumes that the experimenter has a slice *a priori*. Since the current SFA does not provide APIs for creating slices, Gush also does not have the ability to create slices. The experimenter must register with the aggregate managers and upload their public key before using Gush. Once a slice has been created, Gush does have the ability to add and remove nodes from the slice. When a node is added to a PlanetLab slice, a Linux vserver [16] is created on the node, and the experimenter's key is eventually copied out to the vserver. However, the SFA does not provide the ability for Gush to receive any notification as to when the vserver is available for use.

Figure 2 shows a resource directory file that could be used in conjunction with the application specification in Figure 1. The first group of resources are identified as standard SSH resources, which means that they can be accessed using the standard SSH protocol with the username specified in the "user" attribute. The port specified indicates the port on which the Gush client process will run. The "group" attribute is optionally used to categorize resources. In our example, we only have two resources in the local group. Note that these resources have nothing to do with GENI. The next three groups of resources are all GENI resources. Gush uses XML-RPC to contact each aggregate manager's PLC database using the email addresses provided in the "user" tags to obtain information about the resources assigned to the williams_gush, gpeni_gush, and maxpl_gush slices respectively. The ports specified in the "port_map" tags indicate the ports on which the Gush clients will run for these slices. (Note that this syntax is likely to change, since the SFA APIs are changing rapidly.)

To gain an appreciation for the flexibility of the resource management abstractions in Gush, consider our example experiment from before as shown in Figure 1. Recall that in this example we are running "cat software.txt" on 20 PlanetLab, GpENI, and

```
<?xml version="1.0" encoding="UTF-8"?>
<gush>
    <resource_manager type="ssh">
        <node hostname="sysnet1.williams.edu:15400" user="jeannie" group="local" />
        <node hostname="sysnet2.williams.edu:15410" user="jeannie" group="local" />
    </resource_manager>
    <resource_manager type="planetlab">
        <user>jeannie@cs.williams.edu</user>
        <port_map slice="williams_gush" port="15415"/>
    </resource_manager>
    <resource_manager type="gpeni">
        <user>jeannie@cs.williams.edu</user>
        <port_map slice="gpeni_gush" port="15416"/>
    </resource_manager>
    <resource_manager type="max">
        <user>jeannie@cs.williams.edu</user>
        <port_map slice="maxpl_gush" port="15417"/>
    </resource_manager>
</gush>
```

Fig. 2. Gush resource directory file specifying different types of resources.

MAX hosts. Now suppose we want to change our application to instead run on the 2 cluster resources defined in the "local" group in our resource directory. To change our target resources, the only modification required to the application specification is in the component definition. Thus, if we change the value of num_hosts in the component definition in Figure 1 to 2 instead of 20, and also change the resource element to

```
<resource type="ssh" group="local"/>,
```

our experiment will run on our local cluster instead.

In addition to the resources defined in a Gush resource directory file, resources can also be added and removed by aggregate managers at any point during an experiment's execution. This is typically accomplished using an XML-RPC interface provided by the Gush controller. Managers that create virtual resources dynamically based on an experiment's needs, for example, contact the Gush controller with information about new available resources, and Gush adds these resources to the user's resource pool. If these resources become unavailable, the external service calls Gush again, and Gush subsequently removes the resources from the resource pool. This is especially useful for lease-based control frameworks, such as ORCA.

2.3 Creating a Matching

After the Gush controller parses the user's application specification and resource directory file, the controller's *resource matcher* is responsible for finding a valid matching—a subset of resources that satisfy the application's demands—for the experiment being managed. The matcher starts with the user's global resource pool consisting of all available resources, and then filters out the resources that are not in the group specified in the component definition. In our initial example, this includes all hosts not assigned to our three slices. Using the remaining resources in the resource pool, the matcher randomly picks 20 (as specified by num_hosts in our example) Node objects and inserts them into a *resource matching*. The Gush controller then connects to the Gush clients running on the resources, and begins installing and configuring the required software.

If any failures occur during configuration or execution, the controller may choose to requery the resource matcher to find replacement resources. In the case of failure, the matcher sets the “failed” flag in the Node that caused the failure, removes it from the matching, and inserts another randomly chosen resource from the resource pool. This process is repeated for each failure throughout the duration of the experiment’s execution. Note that resources that are marked as failed are never chosen to be part of a matching unless they are explicitly un-failed by the user.

3 Supporting ORCA and ProtoGENI

The preceding section discusses how Gush resources are internally maintained and organized into resource pools. It also describes how the Gush resource matcher uses these resource pools and the component definition section of the application specification to create matchings for the experiments being run. These basic abstractions for managing resources, in addition to our extensible XML-based specification language, are the keys to providing an adaptable framework that supports execution in a variety of environments and across multiple control frameworks. The previous section describes how Gush interacts with resources in the PlanetLab control framework. In this section, we take a closer look at how Gush interacts with slice controllers and aggregate managers in two other GENI control frameworks to select and use sets of resources for hosting experiments on different testbeds.

3.1 ORCA Control Framework

In addition to resources in the PlanetLab control framework, Gush also supports execution on virtual machines (VMs) that are dynamically created by the ORCA control framework [10]. A thorough description of the Gush XML-RPC interface and implementation details are described in [7]; we provide a brief summary here only to help the reader appreciate the flexibility of the Gush resource management framework. ORCA provides users with the ability to create clusters of Xen virtual machines [17] or Linux vservers [16] *on demand*. It is important to note that the VMs do not exist before Gush starts the experiment. Instead, the VMs are created dynamically at startup, and the Gush XML-RPC interface is used to add the new VMs to the Gush resource pool. The XML-RPC interface is also used to remove VMs from the Gush resource pool when the experiment ends or when a resource lease expires.

Suppose we want to run our sample application (Figure 1) on a cluster of VMs created by ORCA instead of 20 PlanetLab machines. Using Gush, we do not have to modify our application and execution elements in any way. We only have to add an <orca> element into the component’s rspec definition. The Gush resource management framework abstracts away the low-level details associated with contacting ORCA and configuring the new VMs with the Gush client process. Figure 3 shows the resulting XML. The XML tags that are required in the new <orca> component correspond to supported VM attributes in ORCA, including OS type, memory, bandwidth, CPU share, and requested lease length.

```
<component name="VMGroup1">
  <rspec>
    <num_hosts>20</num_hosts>
    <orca>
       <num_hosts>20</num_hosts>
       <type>1</type>
       <memory>784</memory>
       <bandwidth>300</bandwidth>
       <cpu>75</cpu>
       <lease_length>12000</lease_length>
       <server>http://geni.renci.org/orca:8080</server>
    </orca>
  </rspec>
  <resources>
    <resource type="ssh" group="orca"/>
  </resources>
</component>
```

Fig. 3. Gush component definition that includes an ORCA VM cluster description

3.2 ProtoGENI Control Framework

In the spectrum of architectural design choices between PlanetLab and ORCA, ProtoGENI [11] lies somewhere in the middle. Like PlanetLab, ProtoGENI requires users to register in advance and upload a public key. Unlike PlanetLab, ProtoGENI also requires experimenters to specify a network topology for the nodes involved in the experiment. Gush currently does not provide any support for creating these files, so when using ProtoGENI, any necessary topology files must be created before using Gush. In addition, Gush contacts the ProtoGENI XML-RPC server using SSL; thus, experimenters should download an SSL certificate from the ProtoGENI website and save it locally before starting Gush.

After creating an account on the ProtoGENI website, specifying a network topology, and downloading an SSL certificate, experimenters may begin using Gush to deploy experiments on ProtoGENI. The first step involves configuring the resource directory with the information necessary to communicate with the ProtoGENI XML-RPC server. An example is shown in Figure 4. (Note that Gush currently uses the Emulab API [18] in ProtoGENI, which is why the resource manager's type is "emulab." We are in the process of upgrading Gush to use the new ProtoGENI API.) The pertinent information in this case includes our login name, port number, project ID, experiment ID, and network topology (NS) file. This information is sent to the ProtoGENI server to create and swap in an experiment (if it does not already exist) using the specified topology. Like ORCA, ProtoGENI creates virtual machines dynamically when an experiment is created. Unlike ORCA, the ProtoGENI server has not yet been extended to provide any callbacks to Gush to indicate when the resources are actually ready for use (although the API does provide a blocking RPC call that does not return until all resources are ready). Thus, if a researcher uses Gush to create an ProtoGENI experiment, they must wait some small period of time (less than 5 minutes on average) for the virtual machines to be created and the topology to be instantiated before running any experiments.

At this point, the execution of Gush proceeds in the same way as before: Gush contacts the ProtoGENI server to obtain a list of available resources, and then uses SSH

```
<?xml version="1.0" encoding="UTF-8"?>
<gush>
    <resource_manager type="emulab">
        <user>jeannie</user>
        <port>15420</port>
        <EmulabProjectID>Gush</EmulabProjectID>
        <EmulabExperimentID>gush</EmulabExperimentID>
        <EmulabNSFile>nsfile.ns</EmulabNSFile>
    </resource_manager>
</gush>
```

Fig. 4. Gush directory file that includes ProtoGENI (Emulab) resources

to login to these resources and configure them accordingly. Upon the completion of an experiment, Gush uses the ProtoGENI XML-RPC interface to swap the experiment out, which shuts down all virtual machines associated with the experiment.

4 Related Work

With respect to related work, it is worthwhile to point out the key differences between this paper and [7]. The goal of this paper is to highlight how Gush manages resources in a variety of environments, while the focus of [7] is on the design and implementation of the experiment controller architecture. In addition, this paper discusses several extensions that were added to Gush specifically for GENI that are not part of Plush.

In the context of application management, Gush has similar goals as PlanetLab's appmanager [19] and HP's SmartFrog framework [20]. appmanager is designed exclusively for long-running services on PlanetLab, and thus cannot easily be extended to support different types of resources or control frameworks. SmartFrog is a Java toolkit that can be used to manage a variety of applications on resources that run Java. Unlike Gush, SmartFrog is not process-oriented, however, and thus only supports execution via Java classes and remote method invocation.

In GENI, every project has a set of testbed-specific tools for resource configuration and experiment management. The majority of these tools are not designed for extensibility, and thus do not support execution across testbeds or control frameworks.

5 Future Work and Conclusion

Gush is an application management framework that helps developers deploy and maintain software running on distributed sets of resources. Through a generic resource management interface and an extensible application specification, Gush supports execution on several different types of resources, including PlanetLab machines, ORCA virtual machines, and ProtoGENI hosts. However, some of the fundamental assumptions that Gush makes about resources—such as unlimited energy, always-on network connectivity, and process-oriented execution—do not support an emerging class of wireless and sensor-based resources. Moving forward, we hope to relax these assumptions to allow Gush to perform experiments on a wider variety of resources. For example, to support sensors or configurable routers, Gush may not connect directly to the resources, but instead connect to a client proxy connected to the resource that can interpret Gush commands and perform the corresponding action on the sensor or router. Maintaining persistent TCP connections may also prove to be futile in wireless sensor networks comprised of mobile devices. The overall challenge in these environments is to provide environment-specific support from a common experiment controller framework. We are currently in the process of exploring ways to use Gush in some of these environments, including the ViSE [21] and DOME [22] testbeds. In doing so, our goal is to ultimately provide an adaptive experiment control framework for all GENI users.

References

1. GENI, `http://www.geni.net`
2. FIRE, `http://cordis.europa.eu/fp7/ict/fire/`
3. AKARI, `http://akari-project.nict.go.jp/eng/conceptdesign.htm`
4. CNGI, `http://www.cernet2.edu.cn/en/bg.htm`
5. Albrecht, J., Tuttle, C., Snoeren, A.C., Vahdat, A.: Loose Synchronization for Large-Scale Networked Systems. In: Proceedings of the USENIX Annual Technical Conference (USENIX), pp. 301–314 (2006)
6. Gush, `http://gush.cs.williams.edu`
7. Albrecht, J., Braud, R., Dao, D., Topilski, N., Tuttle, C., Snoeren, A.C., Vahdat, A.: Remote Control: Distributed Application Configuration, Management, and Visualization with Plush. In: Proceedings of the USENIX Large Installation System Administration Conference (LISA), pp. 183–201 (2007)
8. Albrecht, J., Tuttle, C., Snoeren, A.C., Vahdat, A.: PlanetLab Application Management Using Plush. ACM Operating Systems Review (OSR) 40(1), 33–40 (2006)
9. Peterson, L., Bavier, A., Fiuczynski, M., Muir, S.: Experiences Building PlanetLab. In: Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI), pp. 351–366 (2006)
10. Irwin, D., Chase, J., Grit, L., Yumerefendi, A., Becker, D., Yocum, K.: Sharing Networked Resources with Brokered Leases. In: Proceedings of the USENIX Annual Technical Conference (USENIX), pp. 199–212 (2006)
11. ProtoGENI, `http://www.protogeni.net`
12. Orbit, `http://www.orbit-lab.org/`
13. GpENI, `http://wiki.ittc.ku.edu/gpeni`
14. MANFRED, `http://geni.maxgigapop.net/`
15. GeniWrapper, `http://svn.planet-lab.org/wiki/GeniWrapper`
16. Soltesz, S., Potzl, H., Fiuczynski, M., Bavier, A., Peterson, L.: Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In: Proceedings of the EuroSys Conference (EuroSys), pp. 275–288 (2007)
17. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A: Xen and the Art of Virtualization. In: Proceedings of the ACM Symposium on Operating System Principles (SOSP), pp. 164–177 (2003)
18. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An Integrated Experimental Environment for Distributed Systems and Networks. In: Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI), pp. 255–270 (2002)

19. PlanetLab Application Manager,
`http://appmanager.berkeley.intel-research.net`
20. Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P., Toft, P.: SmartFrog: Configuration and Automatic Ignition of Distributed Applications. In: HP Openview University Association Conference (HP OVUA), pp. 1–9 (2003)
21. ViSE Project, `http://vise.cs.umass.edu`
22. Soroush, H., Banerjee, N., Balasubramanian, A., Corner, M.D., Levine, B.N., Lynn, B.: DOME: A Diverse Outdoor Mobile Testbed. In: Proceedings of the ACM International Workshop on Hot Topics of Planet-Scale Mobility Measurements (HotPlanet), pp. 1–6 (2009)