The GENI Experiment Engine

Andy Bavier
PlanetWorks, LLC and
Princeton University
acb@cs.princeton.edu

Jim Chen
iCAIR,
Northwestern University
jim-chen@northwestern.edu

Joe Mambretti
iCAIR,
Northwestern University
j-mambretti@northwestern.edu

Rick McGeer
Communication Design Group
US Ignite, USA
rick.mcgeer@us-ignite.org

Sean McGeer
Duck Syrup
seanmcgeer@gmail.com

Jude Nelson
Princeton University
jcnelson@cs.princeton.edu

Patrick O'Connell
Duck Syrup
patrickboconnell@gmail.com

Glenn Ricart
US Ignite
glenn.ricart@us-ignite.org

Stephen Tredger
University of Victoria
stredger@gmail.com

Yvonne Coady
University of Victoria
ycoady@cs.uvic.ca

ABSTRACT

We describe the GENI Experiment Engine, a Distributed-Platform-as-a-Service facility designed to be implemented on a distributed testbed or infrastructure. The GEE is intended to provide rapid and convenient access to a distributed infrastructure for simple, easy-to-configure experiments and applications. Specifically, the design goal of the GEE is to permit experimenters and application writers to: (a) allocate a GEE slicelet; (b) deploy a simple experiment or application; (c) run the experiment; (d) collect the results; and (e) tear down the experiment, starting from scratch, within five minutes. The GEE consists of a set of cooperating services over the GENI infrastructure, which together with rapidly-allocated slicelets and a rapidly-allocated network offers a complete, ready to use, sliceable platform over the GENI Infrastructure. The GEE is designed to use off-the-shelf components and infrastructure; unlike previous PaaS offerings, it can be nested nicely inside a GENI slice, or any other IaaS infrastructure. Further, the GEE's southbound interface is extremely small and lightweight, making it portable to other underlying infrastructures.

1. INTRODUCTION AND MOTIVATION

Over the course of the past few years, we have participated in the development of a number of demonstrations of the Global Environment for Network Innovateion (GENI) infrastructure [7] , including a distributed geographic information system, a distributed Map/Reduce system over the wide area, and a distributed media transcoding system. Many of these were built on our earlier TransCloud system [5]. The first and most common question we got after each demo was: "Can I use the infrastructure you built on top of GENI?" This question was understandable: GENI is a distributed, highly-configurable Infrastructure-as-a-Service (IaaS) Cloud with deeply-programmable networking, designed to permit any experimenter to construct his own Internet on the GENI substrate. This platform offers great power and flexibility to its users, experimenters, and application developers, but at a price: allocating and configuring a GENI slice is a far more cumbersome task than the relatively lightweight mechanisms that characterize PlanetLab and other Cloud infrastructures that offer less freedom to users than GENI does. All we needed was a way to deploy VMs across the wide area, and on GENI we found deployment and maintenance of our demonstrations to be a significant challenge. Further, all our demonstrations had the same essential components: a network of virtual machines (or, more precisely, some platform isolated from others in a multi-tenant environment); some form of wide-area messaging system; a conceptually- (and, generally, physically-) distributed store; an orchestration system, typically a bunch of Python or Perl scripts which invoked ssh commands on the various nodes. We were building on top of what was essentially PlanetLab [25] plus a few relatively standardized services. It would be convenient if a permanent infrastructure were available that had those services.

The flexibility of the underlying GENI infrastructure permits users to allocate expensive resources which are in short supply, primarily physical machines and heavyweight virtual machines. Since expensive resources cannot be allocated for long periods of time, lease times for GENI slices are short, requiring frequent slice renewal. Given the range of experiments and applications targetted by GENI, this is a requirement for the underlying infrastructure: one can't design an infrastucture anticipating long-duration use of lightweight resources and easily accomodate short-term requests for expensive resources. It is very hard to turn part of an aparment building into a mansion. However, many GENI experiments and applications don't require these expensive resources. For these users, the full flexibility and freedom GENI offers little benefit, and the machinery this freedom requires imposes nontrivial burdens.

TRIDENTCOM 2015, June 24-25, Vancouver, Canada
Copyright © 2015 ICST
DOI 10.4108/icst.tridentcom.2015.259974

An IaaS platform which offers heavyweight resources can unintentionally encourage their use. People will follow the path of least resistance when creating and deploying their experiment or application, especially in a free infrastructure such as GENI. If it's easier to deploy an experiment using only bare-metal machines, many will, even if the experiment could be done much more efficiently using virtual machines or containers. However, lightweight, efficient resource usage requires preplanning. To return to our previous analogy, one can't rent an apartment unless there are apartment buildings. The use of VMs or containers require an infrastructure that supports these as a central feature, with a ready supply always available.

All of these considerations point to the need for a lightweight, easy-to-use infrastructure for potentially long-duration use of inexpensive resources, given use cases for long-running experiments and applications. Experience with PlanetLab suggests that there are many such applications, including:

- Content Distribution Networks
- Distributed hash tables
- Wide-Area stores
- Network observation platforms
- Distributed DNS
- Distributed messaging services
- Multicast overlays
- Wide-area programming environments

IaaS platforms inherently support overlay platforms as a service, and this is exploited both in the academic and commercial sectors. The overlay platforms are always specializations of the underlying infrastructure: one can limit the capabilities and flexibility in an overlay, for ease of use and to encourage the use of specific types of resources; it's difficult to enhance capabilities not present in the underlying platform. Further, GENI was specifically designed to permit the construction of overlay Clouds built within GENI itself; after all, Cloud research is a major driving use case for GENI.

This made our strategy obvious: construct an easy-to-use Cloud within GENI that offered long-duration slices of virtual machines or containers, distributed throughout the GENI infrastructure. PlanetLab is that: a Cloud that offered long-duration slices of distributed containers, with a large user base, a decade of 24/7 operation, and a toolchain that could be easily adapted to a PlanetLab-within-GENI architecture.

That was our first motivation, and it was rapidly supplemented by a second: PlanetLab has aged. When it was developed in 2003, there was no such thing as a "Cloud". Planetlab and Emulab [37] were pioneers in this area, and, as is often the case with pioneering software artifacts, would be built differently given modern technologies. PlanetLab relied on a custom configuration and management using VServer, which tied its users to a specific Fedora Core image. This also makes it difficult to layer PlanetLab on top of other Cloud platforms, since it must be compatible with the node management system of the underlying infrastructure. Its database and portal are both built on decade-old technologies.

These considerations led us to our second major goal: to build an updated, lightweight, embeddable PlanetLab that leveraged modern tools for custom image management and be easily deployed over multiple underlying Cloud infrastructures. The development of modern Cloud management, deployment, and configuration technologies such as Fabric [16], Ansible [3], Chef [12] and Puppet [26], meant that most of the deployment and configuration effort involved in maintaining PlanetLab could now be done by publicly-available tools with a widespread user base and significant external support.

Similarly, host and container management tools such as Docker [14] subsume many of the functions of the PlanetLab node manager as well as providing many value-added functions for users of the platform. One specific example is an efficient, network-accessible image management and deployment system, DockerHub [15], which permits users to easily store, version, and deploy customized images for their experiment using a novel combination of image management and GitHub [17].

PlanetLab was designed as a standalone infrastructure. Our update, the GENI Experiment Engine (the GEE), has been designed to be embedded in the GENI infrastructure in its first incarnation. Given the plethora of Cloud systems now available in every field, we could assume the existence of an underlying Cloud infrastructure for any deployment of the GEE. In this circumstance, many of the features of PlanetLab Central were not only redundant, but undesirable from both a system and user perspective. For example, the MyPLC database stores authentication information for users, whereas an embedded platform can and should use the underlying platform for user authentication. For example, the Google App Engine uses Google's underlying infrastructure for user authentication.

These considerations brought the design goals of the GENI Experiment Engine into sharp focus. We wanted an infrastructure which accomplished the goals of PlanetLab – to offer distributed systems experimenters containers spread over the wide area, in a lightweight, easily-allocated manner, with a number of new features:

- The infrastructure should be embedded in an underlying Infrastructure-as-a-Service offering

- The containers at a site should be deployed in a lightweight manner within a virtual machine
- It should be easy for experimenters to create, version, manage, and deploy their own node images
- There should be support for modern configuration management and orchestration tools
- The interface to the underlying infrastructure should be small and standards-based, to permit ports to other infrastructures
- Value-added services such as wide-area stores and pluggable HTTP servers should, where possible, be deployed in slices which are easily accessible to other slices. Where not, support should be given for rapid deployment via orchestration tools.
- Simple, understandable networking.
- An experimenter should be able to create a slice, use it to run "Hello, World" across the GEE infrastructure, and tear it down, within five minutes

The remainder of this paper is organized as follows. In Section 2, we describe the process of using GEE: how to allocate, use, and tear down a GEE "slicelet". In Section 3, we describe the architecture and services of the GEE. In Section 4, we describe related work in the testbed and cloud arenas. In Section 5, we describe the current status of the GEE and its future.

2. RUNNING A GEE EXPERIMENT

The easiest way to get a feel for an architecture like GEE is to consider its usage. To use the GEE, a user logs in to the GEE portal using her GENI credentials. The GEE portal stores no user information or credentials; instead, OpenID[27] is used to call back to the GENI portal, and the user's email is the userid for the purposes of the GEE. The user is then directed to a dashboard, where, with the click of a button, she can allocate a GEE Slicelet. When this process is completed (within a few seconds), a download link to a zip file appears on her dashboard. The user can then download the file to his computer. The zip file contains five files:

1. The slicelet's private key
2. An ssh-config file which contains configuration directives to log into slivers on the slicelet. A typical use would be `$ssh -i ./id_rsa -F ./ssh-config slice295.pcvm4-1.utahddc.geniracks.net`, which logs in to slicelet 295's sliver on pcvm4-1 on the InstaGENI rack in the Utah downtown data center.
3. A Fabric[16] file, with ssh configuration, host, and key variables pre-populated, that the experimenter can use to deploy software, configure the slivers, or run experiments on the slicelet. Of course, the Fabric file references the generated ssh-configuration file, and so the same symbolic names are used in place of IP address and port number
4. An Ansible hosts file, for those users who use Ansible for slice configuration and deployment.
5. A README file.

Of the five items, only the first two are required to access the slice. The remaining three are convenience items to get "Hello, World" up and running.

Once the user has downloaded and unpacked the slice file, she is immediately able to ssh into slivers in the usual fashion, and configure them in the usual way. A user will also be able to use any ssh tool of her choosing to populate or control her slice. However, use of the enclosed Fabric file makes upload and execution as easy and quick (roughly, as easy as uploading a Python program to the Google App Engine).

Fabric is one solution to single pane-of-glass control of a slicelet. It is simply a Python wrapper around ssh commands, which automates the execution of both remote and local commands. We have pre-loaded the Fabric file with a number of commands to both introduce the user to Fabric and to give them out-of-the-box functionality on the site. For example, typing: "fab nmap" runs a script on each host that reports the reachable IP addresses on the private network.

A second solution, with somewhat different semantics, is Ansible. Ansible uses YAML [38] as a declarative description of the node configuration. Rather than issuing ssh commands to the nodes to install and configure software, the user writes a YAML description of the final state of the node, and Ansible issues the necessary commands (a mix of prebuilt and user-specified commands) to build and configure the node. Both Ansible and Fabric are supported by, and used by, the GEE.

The user can tear down her experiment by using the "free slicelet" button on her dashboard.

No configuration of the slivers in the slicelet is required: the user simply runs her experiment. Indeed, if the software the experiment requires is pre-installed on a basic Ubuntu 14.04 LTS distribution, the user need not install any software at all.

3. ARCHITECTURE AND IMPLEMENTATION OF THE GEE: THE FAD ARCHITECTURE

The GEE is a set of multiple services, which offer various features to users. The bedrock is a Compute Service, which allocates GEE "slicelets" and configures

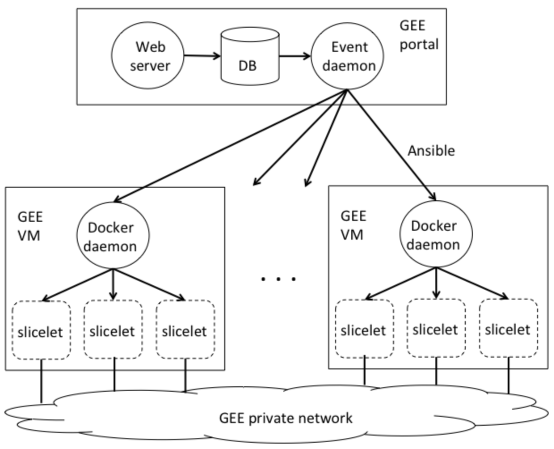

Figure 1: The Architecture of the GEE Compute Service

them. The remaining services, which will grow over time, are either deployed in their own slicelets or offered as deployment options within a slicelet. The former is preferred; the latter is chosen when offering as a separate slicelet is infeasible due to protection or other considerations. An example of a service which is deployed via slicelet configuration is the Message Service, described below: it is offered as a deployment option because the simple, efficient message service we chose to support is single-tenant, and deploying in a separate slice introduced complexity without adding functionality. Slicelets already are partitioned from other GEE tenant slicelets; there was no reason to add an extra layer of security merely for the architectural convenience of using a separate slicelet.

3.1 The GEE Compute Service (FAD Architecture)

The overall architecture of the GEE Compute Service is shown in Figure 1. We use Docker as the container manager service on the slice. Docker is essentially an overlay on a linux container solution, either using `libvirt` and `LXC` [13], or using the built-in `libcontainer` library. Despite its relative youth – the first release was in March of 2013 – it has become an extremely popular virtualization solution, with over 16,000 deployed images on DockerHub. Its primary use is to provide isolation for multiple processes running within a virtual machine, and this has been responsible for most of its uptake. Docker's web page advertises that "'Dockerized' apps are completely portable and can run anywhere" but currently support is limited to Linux. A Dockerized application is independent of the underlying OS. Each Docker "virtualized application" carries only its libraries, without an underlying guest OS. This gives significant size savings. The Ubuntu 14.04 Docker container is about 255 MB, compared to at least 1 GB of disk space for an Ubuntu VM.

Though Docker has been primarily used in the enterprise IT space to scale individual applications seamlessly within a VM, the functions of the Docker Engine are quite similar to those of the Node Manager of PlanetLab – to instantiate and deploy containers and populate them with images. It was easily adapted to managing a PlanetLab-style multi-tenant container node.

The Docker Engine comes in two parts: an on-node Docker daemon, which creates, manages, and destroys the containers, and populates them with images; and a client that issues Docker commands to the daemons. Our base installation for a node image is a GEE-customized version of an Ubuntu-based Docker image, available on DockerHub at gee-project/phusion-baseimage. We use Ansible playbooks as the interface to Docker to create and delete containers and build the slicelet zip file from templates.

The value of Ansible and Docker was easy to see: the Ansible slice-creation YAML [38] file was only 57 lines of markup , and the script which invoked Ansible and created the slice tarball was only 18 lines of bash.

This remarkable economy is also due to our ability to configure slivers post-instantiation through the use of Fabric and/or Ansible commands and scripts. To install the GEE Message Service we wrote a Fabric command which installed the appropriate server package, started it, and installed the Python client libraries on the hosts. This combination of three tools – Docker for sliver management and image manipulation, Ansible for sliver creation and post-creation customization, and Fabric for post-creation customization and experiment control – led us to name this the *Fabric, Ansible, Docker (FAD)* architecture for embedded distributed infrastructures.

A second simplification is due to the embedded nature of the GEE. Since the GEE is embedded, its containers run in VMs allocated by the underlying infrastructure. Connectivity to the VMs is maintained by the underlying infrastructure, relieving the GEE from maintaining and repairing this connectivity; instead, we can rely on the underlying GENI tools. InstaGENI PlanetLab was always envisioned as a subservice running over ProtoGENI [29] on the InstaGENI racks [4], and a convenient way for GENI users to run VM-based experiments. The GEE now replaces InstaGENI PlanetLab in this system, with four major enhancements:

1. Update of the standard VM to a modern image
2. VM enhancements with pre-installed services
3. Optional pre-allocation of GEE slicelets
4. Use-once credentials for access to GEE slicelets

3.1.1 *The GEE Portal*

The interface for a user to create and manage slicelets is through the GEE Portal, at `http://www.gee-project.org`. The Portal itself runs in two Docker containers inside a VM on the Stanford VICCI cluster [34, 24]. We use Docker both for its convenience as an execution environment and to gain hands-on experience with features such as inter-container networking, which we will employ for services deployed in slicelets

The first container has a Mongo [21] database, which is used to register users, slices, and slice manipulation requests. No credential information for the user is stored; the only records are the user name, email, and the slice, if any, which he has created.

In addition to its usual tasks, the database is designed to be an intermediate representation for stateful processes, primarily slice creation and deletion. When a user makes a slice request (other than renewal, which is handled entirely by the database itself), the portal issues a request into the database which a daemon process subsequently services; the slice status is kept in a database field. This architecture was chosen to permit the portal to respond instantly to a user request, without waiting for back-end processes to complete.

The second VM contains the webserver and associated scripts. Database requests are made through the networking architecture of Docker, and the connections are made at boot time for the two containers.

Use of Docker within a VM has had a number of benefits, in addition to familiarizing us with the slivers' execution environment. The first is that we are able to use the portal VM itself as a test system. We actually maintain two sets of Docker containers, one for test and one for production, and use other Docker-based hosts on the VICCI cluster as a test production system. This has meant that any enhancements to or tests of the portal can be run in a nearly-perfect *in situ* environment, leading to rapid debug and reliability cycles.

3.1.2 Authentication and User Access

Authentication and user access were questions that we considered carefully. We wanted to offer the GEE to any user with GENI access, without maintaining a separate database of authentication information. This was chosen for reasons of user convenience, maintainability, and user security. Users, once they have registered with GENI, should not need to add themselves to a separate database. Further, delegating authentication promotes maintainability, and not keeping user authentication information afforded attackers one fewer place to obtain ssh keys and passwords.

To authenticate users we used an OpenID callback to the GENI portal, obtaining the minimum information needed to create and maintain user slices – the user's email address, which was the only indexing information used in the GEE portal database.

3.1.3 Optional Pre-Allocation

The "five-minute rule" has dominated our design consideration. Delay in use of PlanetLab slices after allocation was due to sliver configuration and key propagation. This is a much more rapid process in the FAD-based GEE, but it is still nonzero; further, a number of scenarios (such as, for example, use in tutorials) envision the creation of multiple slicelets more or less simultaneously. We serialize slice creation requests, to avoid excessive network traffic to the GEE nodes, using a daemon on the GEE portal to continuously service incoming requests. Since slice creation is serialized and creation of each slice takes on the order of tens of seconds, we optionally maintain a bank of pre-created slices as a buffer against heavy node creation time.

3.1.4 Use-once Keys

Note that even though we're providing the use-once slice private key as a convenience, for bootstrap and for immediate usability, the user will be able to install keys of her choosing on the slice once she has access to it. One can even remove the use-once key if desired, though this is not recommended. Some actions void warranties.

We used a use-once, or "burner" key for two reasons: speed and security. Speed is obvious: we have pre-propagated the key. Security is nearly as obvious: if a user's slicelet is compromised, or the use-once key is discovered, all that is compromised is the user's slicelet. The GEE portal retains no credential from the user, and so cannot compromise any user credentials. Similarly, compromise of a user's ssh key won't result in an attacker gaining access to a GEE slicelet.

Use-once keys are the infrastructure equivalent of hotel room cardkeys; they are allocated when the slicelet is instantiated, used only to access the slicelet, and are destroyed when the slicelet is de-allocated. As a result, they come with fewer security concerns than do standard keys, just as a hotel is completely unconcerned with travelers departing with cardkeys in their pockets.

3.2 The GEE Message Service

The GEE Message Service is used to route job control messages within a slicelet; this is a common feature of many Cloud systems, and a number of systems are available. The Message Service is a server which can be loaded into the slicelet, and a client library; a user activates the server on whichever nodes in the slicelet she prefers through a Fabric command. We searched for a message service that is well-documented, simple, configures automatically, has a rich set of client libraries, and can be enabled with a `service start` command.

We chose Beanstalk [6]. Beanstalk has libraries in a variety of languages, notably including Python. It installs as a service on Ubuntu, with a configurable port. It has a simple put/get interface and supports a wide variety of use models, including pub/sub.

As with many Message Service systems, Beanstalk is configured for a single-tenant environment. Its primary use case (like Docker) is to coordinate multiple tasks within a data center. Its use mode is not that a multi-tenant provider offers messaging-as-a-service, such as IronMQ, but rather that each job or service instantiate its own messaging server accessable only from its own nodes: security is assumed at the slicelet, not the service, level. This dictated our deployment choice: rather than instantiating a GEE- or GENI-wide messaging service, we chose to offer the experimenter a Fabric command to turn the service on in the appropriate slivers, and choose the appropriate server site.

3.3 The GEE Network

The GEE Network is a private layer-2 network spanning the infrastructure on which the GENI Experiment Engine is deployed. Each GEE Sliver has a single interface on this network, with a `10.` address.

GEE Slicelet networks are not completely isolated. One of the major use cases for slices in PlanetLab was slices providing services for other slices: e.g., PsEPR [8] provided monitoring information and Stork [9] loaded images for other slices efficiently. The PlanetLab mantra for services was "put it in a slice", which led to a micro-kernel architecture for a distributed system: if it didn't absolutely need to be in the PlanetLab controller, it was in a services slice. This greatly simplified the design of PlanetLab, permitted experimentation in utilities and services, and contributed to the lifespan and maintainability of the PlanetLab infrastructure.

For these reasons, the GENI Experiment Engine is adopting the same design philosophy. The GENI Storage Service will be deployed in a slicelet, as is the GENI Reverse Proxy Service. Our original intent was to offer the messaging service in a slicelet, but the requirement for a secure multi-tenant service restricted our choices and added unnecessary complexity to what was otherwise a simple, foolproof mechanism: hence our choice to add a service to the slicelet rather than offer a multi-tenant service in its own slicelet.

One difference between PlanetLab and GENI is that GENI has a private network that is used for intra-slice communication. The private network is attractive for two reasons: conservation of port space on routable IPs and security. Public-facing services are under continual attack from botnets, something privately-deployed services need not protect against. The GEE private network is a virtual intranet for GEE slicelets, and we can use it for GEE-specific services.

Fortunately, since Docker was designed to permit multiple cooperating applications to communicate while enjoying near-VM-level isolation, the Docker networking interface readily admits inter-slice communication without foreknowledge of local IP addresses. In fact, we use the Docker mechanism in the GEE portal, to permit the GEE portal to communicate with the database server without use of hard IP addresses.

When a container is instantiated, it is optionally given a name. This name can be bound to a local name in other Docker containers when they are instantiated, and these containers now have a virtual interface to the named container, which is referenced by name. So, for example, we have bound our database container with the name `mongodb` in our portal container, and the GEE portal binds to the database server at hostname `mongodb`, port `12071`. This name is only valid within that container, and is bound on node creation.

We will use the same mechanism for slice-hosted services. On each node, these slices will be given the appropriate symbolic name, and future slices will be created with that name bound on each node.

Implementation of the GEE Network evolves as GENI itself evolves. Our demonstration network used a temporary circuit from Ion, which is unsuitable for production use. Our first production network used a GENI-wide VLAN. In June 2014, we built a version of the GEE integrated with the Virtual Topology Service. GEE-on-VTS is the long-term architecture for the GEE network.

3.4 Controlling GEE Experiments From The Desktop

Scalability of control for a distributed application is critical. Slice management and configuration was the focus of a large number of early PlanetLab efforts [2, 1, 9, 36, 10]. Despite a number of early efforts for unified desktop orchestration, most early experimenters used a combination of Perl, ssh, and python scripts for experiment orchestration and control. The emergence of Cloud platforms and the need for scalable orchestration, configuration and management of very large-scale systems has given rise to a number of open-source and commercial tools for these purposes. We have chosen to support two, Fabric and Ansible.

Both Fabric and Ansible employ Python wrappers over ssh. As with most configuration management and orchestration software, both distinguish between *controllers* and *nodes*. A controller executes configuration commands to configure the nodes. Both are *agentless*: they require no agent on the nodes themselves. Fabric requires only OpenSSH on the nodes; Ansible requires both OpenSSH and Python 2.4 or later.

Fabric is a set of Python libraries which wrap sftp for file transfer and OpenSSH for command execution. As this implies, it offers an imperative semantics for node orchestration and configuration management. Ansible also offers a declarative semantics for known tasks in its *Playbook* abstraction. Whereas a Fabric command to install a webserver would be given by the installation instructions:

```
def install():
    run('sudo␣apt−get␣update')
    run('sudo␣apt−get␣install␣−y␣apache2')
    run('sudo␣apache2ctl␣start')
```

The Ansible declaration would be:

```
tasks:
  − name: install apache
    apt: name=apache2 state=present \
            update_cache=yes
  − name: make sure apache is running
    service: name=httpd state=started
```

Both Ansible and Fabric have reasonable roles to play in coordinating wide-area experiments and distributed applications. Ansible requires installation of Python-based software on the desktop; in contrast, Fabric requires only the installation of a Python library through `pip` or `easy_install`. Our solution was to support both, through the definition of skeleton files which incorporated slice information and rudimentary commands, making it easy for experimenters to extend.

The inclusion of Ansible and Fabric in our workflow turned out to have substantial benefits for Slicelet deployment and configuration, and significant simplification of both the core of the GENI Experiment Engine and deployed slices. Rather than pre-installing a great deal of software on the experiment nodes, we could simply incorporate the relevant Ansible or Fabric commands in the files we downloaded to the user.

3.5 Planned, β-Level Services

There are two services that we have tested in preliminary form, and intend to roll out for our users in the next few months as supported services: the GEE Storage Service and one or more Reverse Proxy Services

3.5.1 GEE Storage Service

The GEE Storage Service will be offered by a Python library that can be loaded into the user's slicelet, which presents a filesystem API to the end user. The library itself then makes REpresentational State Transfer (REST) calls to a network of storage proxies inside and outside the GEE system to store and retrieve data. In fact, a storage slicelet is not a requirement: it presence offers a low-latency high-bandwidth cache to the experimenter, and due to resource limitations we will not guarantee its presence on every node.

Requirements.

The GENI Experiment Engine File System (GEE FS) is designed to be an easy-to-use file system provided on all GEE slices. We need the file system to be accessible both inside and outside experiments to allow experimenters to access stored data from inside and outside GENI experiments. The GEE FS provides an accessible, persistent, environment for all GENI experiments. Since we want to make the GEE FS as easy-to-use as possible we have the following design goals:

- Unix-like semantics
- Convenient, reliable, distributed storage
- Accessible from any GENI experiment
- Runs on any reasonable host backend
- Exposed API

The GEE FS is built using OpenStack Swift [31] nodes as a backend for Syndicate [22], a wide-area file system being developed at Princeton University and ON.LAB. Syndicate handles metadata in the file system as well as access control, versioning, and replication, while providing a Unix like interface. Syndicate allows us to use multiple backend services distributed around the GENI network. The remainder of the section looks at each component of the file system in more detail.

Metadata Server.

The most integral component of the file system is the Metadata Server (MS), which handles all file system metadata requests. For this we need a reliable service that can handle concurrent connections and is easily accessible. We use the Syndicate MS [22], which is implemented as a Google App Engine application and stores its data in BigTable. Google App Engine offers efficient scaling and BigTable offers efficient key-value lookups. Users and Groups are handled by the MS regulating user access, locally through the file system client.

Storage Service.

Syndicate has client processes and storage processes. The storage service is a Python process that runs on remote nodes and act as a translator between Syndicate, and the storage service being used. Syndicate writes blocks of data to a storage service, and replicates the data for data durability. A Swift backend is accessed via HTTP and provides a Python API which allows easy integration with Syndicate. Storage services can be added and removed from Syndicate on the fly as the MS handles the actual layout of data (and its replicas) which gives us the flexibility to grow our storage capacity to meet the demands of GENI users. Additionally other storage providers (S3, DropBox, even the local file system) can be integrated into the file system.

3.5.2 The GEE Reverse Proxy Service

Intra-slicelet traffic on the The GENI Experiment Engine will primarily be through a network private to the slice, which is isolated from the public Internet. Routable IPs are notoriously scarce on PlanetLab nodes, and GENI member institutions have been unwilling or unable to devote large banks of routable IP addresses to GENI slices. A GENI rack is capable of running well over 100 slivers, and we believe that 256 is achievable on the InstaGENI racks. We would need a /24 per site to accommodate these slivers, and at many sites we're lucky to get a /27. Clearly, we cannot count on being able to give a routable IP to each sliver.

Though GENI's private network suffices for intra-slicelet traffic, a number of PlanetLab slices and services offered distributed public services. The most notable of these were the Content Distribution Networks (CoDeeN and Coral) [35], End-System Multicast [11], and the Distributed Hash Tables [28]. Clearly, for such services to use the GEE, some method must be found to enable public-facing services at each site.

We don't have enough IP addresses to offer each public-facing service its own routable IP, and it isn't really feasible to assign each its own port: an HTTP service that isn't on port 80 faces multiple logistical problems, from firewalls to configuration of client-side software. The reader might imagine that we are less than enchanted when we hear CIOs argue that there is no use case or demand for IPv6, and there are plenty of v4 addresses; and then in the next breath to refuse to part with a /24 from their campus' /8. Nonetheless, the fact that the problem is an artifact of conflicting University IT policies make it no less a problem: if our experimenters are to offer public-facing services on GEE nodes, we must find a way to give them all access to the same port on the same v4 address. The solution we hit upon was to multiplex the HTTP ports and isolate at the URL level, enabled by the GEE Reverse Proxy.

The GEE Reverse Proxy Service operates a reverse proxy in a sliver on each GEE site. HTTP PUT, GET, and POST requests of the form `http://<hostname>/<sliceletname>/<request>` are caught by the reverse proxy and sent to the http server in the slicelet's sliver over the GEE private network; the returned value is sent back to the requester.

3.5.3 Lively on GEE

The Lively Web [32] is a particularly attractive candidate for a reverse proxy or, more generally, for a multi-tenant distributed systems programming environment. A descendant of the Smalltalk [18], Self [33], and Squeak [19] programming languages and environments, Lively is a media wiki which abstracts HTML and CSS into a single graphical abstraction called a "morph". This permits client-side web programming exclusively in Javascript, without a requirement to frame the client-side code within a styled markup object. Further, the programming environment is simply a Lively web page, with the unified layout and code design which is the hallmark of morphic-based programming environments. Unified client-side and server-side programming is accomplished through a pluggable node.js server, programmable directly in the web page-based programming environment. The pluggable server comes with a variety of builtin language environments, including Ruby, R, and Python, as well as an embedded SQLite server.

4. RELATED WORK

The GENI Experiment Engine is a Platform-as-a-Service (PaaS) operated on top of an Infrastructure-as-a-Service (IaaS) base. In this, it is not unique. The Google App Engine is a very heavily-used PaaS offering on Google's infrastructure. Further, there are deployable PaaS offerings. OpenShift from RedHat is an offering which orchestrates application deployment on the public cloud and offers PaaS on the enterprise cloud. There are many other examples: after all, to a first approximation offering PaaS on an IaaS offering is simply populating component VM's with the appropriate programming environments and platforms and providing orchestration services, notably automated scalability.

Commercial PaaS offerings focus on scalability and automatically scaling applications. In the GENI context, this is not a consideration: we do not have arbitrary resources on any single rack to scale the application; for GENI applications, location matters far more than scalability. Our primary concern is communication across the wide area and network design, concerns that are not relevant for data-center orienter PaaS systems.

AptLab [30] is similar in spirit to the GEE: it is a set of pre-configured "profiles" (essentially, pre-defined slices) on InstaGENI, and is designed to get users up and running fast on InstaGENI.

5. STATUS AND FUTURE WORK

The GEE is being brought up and deployed in stages, as the various services mature. The GEE Portal is up and running, and the GEE Compute Service is functional. We demonstrated the GEE Compute Service and the Fabric-based single-pane-of-glass experiment control at GEC 19 [20]. The GEE Compute Service is now in production at 20 sites across the United States.

The GEE Storage Service is nearly as mature. The integration between the Swift proxies and the Syndicate metadata service is complete, and has been tested on GENI and Emulab. We will be ready to do a beta deployment as soon as the resources (primarily, VMs on GENIRacks) are obtained on a long-term basis. The GEE Proxy Service and the GEE Message Service have both been tested on VICCI slices, and we will soon test in GEE Slicelets.

The GEE is functional and stable because it is built on well-tested infrastructure services and components.

The base for our compute service is Docker; for messaging , we used Beanstalk; and for single-pane-of-glass control, we used Fabric and Ansible.

GEE lives light on the land. Our interface to the underlying infrastructure is a few shell scripts; to ID providers an OpenID callback. Our remaining services are instantiated inside VMs. We are able to bring up the GEE on any distributed infrastructure providing VMs that have public IP addresses and support Docker.

A particularly attractive possibility for the future is to permit users to build and run their own images, using DockerHub to store and deploy them. We intend to explore doing this this year, with students from the University of Victoria. There are two issues: the means by which the user chooses the image or images at Slicelet-creation time, and ensuring that the images have the tools in place for remote access control.

For both these problems, we intend to use a variant of the Emulab/AptLab method of image creation. The user starts with an existing stock base image, then uses this as a basis for building the image he wants. Once this is complete, he uses a web page on the GEE portal to snapshot the image. The snapshot tool invokes the standard Docker mechansims to create, commit, and push an image, using the user-specified name in the gee-project repository on DockerHub. These stock images will then be added to a pulldown menu in the slice-creation page on the GEE portal.

We have beta-tested the GEE Compute Service in the CS 462 class at the University of Victoria in Spring 2015. The GEE was used as the basis of three labs in the course, and as the foundation of a number of student term projects.

We have recently built a wide-area monitoring system on the GEE,inspired by PlanetSeer [39], PsEPR [8] and CoMon [23]. A `monit` daemon runs on each node, reporting status to a Lively website. The results can be viewed (at this writing) at `http://www.lively-web.org/users/rick/gee-monitor.html`

Acknowledgements

The authors thank Leigh Stoller and Rob Ricci of the University of Utah and Niky Riga and Mark Berman of the GPO for much logistical assistance in setting up and maintaining the GEE Slicelets; Niky Riga, Vic Thomas, and Sarah Edwards of the GPO for counsel and logistical assistance in setting up GEE tutorials; Marshall Brinn of the GPO and Nick Bastin of Barnstomer Softworks for productive conversations; Chip Elliott of the GPO for years of guidance; Bill Wallace, Joe Kochan and the staff at US Ignite; Patrick Scaglia, Dan Ingalls, Alan Kay, Sanjay Rajagopalan, and Carlie Pham of SAP/CDG for financial and logistical support and mentoring; Robert Krahn and Marko Roder of SAP/CDG for assistance with our Lively Web monitoring front end; and Jack Brassil of the InstaGENI project and HP Labs for invaluable help. This project was partially funded by the GENI Project Office under subaward from the National Science Foundation.

6. REFERENCES

[1] J. R. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, and A. Vahdat. Remote control: Distributed application configuration, management, and visualization with plush. In *Proceedings of the 21th Large Installation System Administration Conference, LISA 2007, Dallas, Texas, USA, November 11-16, 2007*, pages 183–201, 2007.

[2] J. R. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Planetlab application management using plush. *Operating Systems Review*, 40(1):33–40, 2006.

[3] Ansible api documentation. `http://docs.ansible.com/`.

[4] N. Bastin, A. Bavier, J. Blaine, J. Chen, N. Krishnan, J. Mambretti, R. McGeer, R. Ricci, and N. Watts. The instageni initiative: An architecture for distributed systems and advanced programmable networks. *Computer Networks*, 61(0):24 – 38, 2014. Special issue on Future Internet Testbeds - Part I.

[5] A. Bavier, Y. Coady, T. Mack, C. Matthews, J. Mambretti, R. McGeer, P. Mueller, A. Snoeren, and M. Yuen. Genicloud and transcloud. In *Proceedings of the 2012 Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit*, FederatedClouds '12, pages 13–18, New York, NY, USA, 2012. ACM.

[6] Beanstalk. `http://kr.github.io/beanstalkd/`.

[7] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5 – 23, 2014. Special issue on Future Internet Testbeds - Part I.

[8] P. Brett, R. Knauerhase, M. Bowman, R. Adams, A. Nataraj, J. Sedayao, and M. Spindel. A shared global event propagation system to enable next generation distributed services. In *WORLDS '04*, 2004.

[9] J. Cappos, S. Baker, J. Plichta, D. Nyugen, J. Hardies, M. Borgard, J. Johnston, and J. Hartman. Stork: Package management for distributed vm environments. In *Proceedings of the 21th Large Installation System Administration Conference, LISA 2007, Dallas, Texas, USA, November 11-16, 2007*, 2007.

[10] J. Cappos and J. Hartman. Why it is hard to build a long running service on planetlab. In

Workshop on Real, Large, Distributed Systems (WORLDS '05), December 2005.

[11] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *SOSP '03*, 2003.

[12] Chef. `https://www.chef.io/chef/`.

[13] Lxc: Linux containers. `https://linuxcontainers.org/lxc/downloads/`.

[14] Docker. `https://www.docker.com/whatisdocker/`.

[15] Dockerhub. `https://hub.docker.com/`.

[16] Fabric api documentation. `http://docs.fabfile.org/en/1.8/`.

[17] Github. `https://github.com/`.

[18] D. Ingalls. The smalltalk-76 programming system design implementation. In *ACM Conference on Principles of Programming Languages*, 1978.

[19] D. Ingalls, D. Ingalls, T. Kaehler, T. Kaehler, J. Maloney, J. Maloney, S. Wallace, S. Wallace, A. Kay, and W. D. Imagineering. Back to the future: The story of squeak, a practical smalltalk written in itself. In *In Proceedings OOPSLA âĂŹ97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, 1997.

[20] R. McGeer. Gec 19 gee demo video. `https://www.youtube.com/watch?v=RDnWIqtatkA`.

[21] Mongodb. `http://www.mongodb.org`.

[22] J. Nelson and L. Peterson. Syndicate: Democratizing cloud storage and caching through service composition. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 46:1–46:2, New York, NY, USA, 2013. ACM.

[23] K. Park and V. S. Pai. Comon: a mostly-scalable monitoring system for planetlab. *SIGOPS Oper. Syst. Rev*, 2006.

[24] L. Peterson, A. Bavier, and S. Bhatia. Vicci: A programmable cloud-computing research testbed. Technical Report TR-912-11, Department of Computer Science, Princeton University, 2011.

[25] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building planetlab. In *In Proceedings of the 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI*, 2006.

[26] Open source puppet. `http://puppetlabs.com/puppet/puppet-open-source`.

[27] D. Recordon and D. Reed. Openid 2.0: A platform for user-centric identity management. In *Proceedings of the Second ACM Workshop on Digital Identity Management*, DIM '06, pages 11–16, New York, NY, USA, 2006. ACM.

[28] S. C. Rhea. *Opendht: A Public Dht Service.* PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2005. AAI3211499.

[29] R. Ricci, J. Duerig, L. Stoller, G. Wong, S. Chikkulapelly, and W. Seok. Designing a federated testbed as a distributed system. In *Proceedings of the 8th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, June 2012.

[30] R. Ricci, G. Wong, L. Stoller, K. Webb, J. Duerig, K. Downie, and M. Hibler. Apt: A platform for repeatable research in computer science. *ACM SIGOPS Operating Systems Review*, 49(1), Jan. 2015.

[31] Swift. `https://wiki.openstack.org/wiki/`.

[32] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web browser as an application platform: The lively kernel experience, 2008.

[33] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

[34] Vicci. `http://www.vicci.org/`.

[35] L. Wang, K. S. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the codeen content distribution network. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 14–14, Berkeley, CA, USA, 2004. USENIX Association.

[36] K. Webb, M. Hibler, R. Ricci, A. Clements, and J. Lepreau. Implementing the emulab-planetlab portal: Experiences and lessons learned. In *Workshop on Real, Large, Distributed Systems (WORLDS '04)*, December 2004.

[37] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[38] Yaml: Yaml ain't markup language. `http://www.yaml.org`.

[39] M. Zhang, C. Zhang, V. Pai, L. Peterson, and Y. Wang. Planetseer: Internet path failure monitoring and characterization in wide-area services. In *In OSDI*, pages 167–182, 2004.