PlanetIgnite: A Self-Assembling, Lightweight, Infrastructure-as-a-Service Edge Cloud

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

Rick McGeer
US Ignite
rick.mcgeer@us-ignite.org

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

***Abstract*—PlanetIgnite is a general-purpose, Infrastructure-as-a-Service, self-assembling, lightweight edge cloud on virtualized infrastructure with support for single-pane-of-glass distributed application configuration and deployment. This is an entirely new concept. PlanetLab[32], GENI[7], [22], and SAVI[19] are general-purpose IaaS edge clouds, but require top-down installation and dedicated hardware resources at each site and do not offer single-pane-of-glass application deployment. Seattle[11] is a lightweight self-assembling edge cloud that offers single-pane-of-class configuration and control, but developers are restricted to using a subset of Python. PlanetIgnite is a Containers-as-a-Service Edge Cloud which offers Docker Containers to each PlanetIgnite user. A PlanetIgnite node is an off-the-shelf Ubuntu 14.04 Virtual machine with Docker installed, meaning it can be installed on any edge node where a VM with a routable v4 address is available. Adding a PlanetIgnite node to the infrastructure is simple: a site wishing to host a PlanetIgnite node simply downloads the image; on boot, the new PlanetIgnite node registers with the PlanetIgnite portal, which runs a series of acceptance tests. Once complete, the image is registered and the node is added to the set of PlanetIgnite sites.**

I. Introduction and Motivation

Computing systems have traditionally been logically centralized systems, either running on a single computer or with components separated by a few hundred microseconds. However, tomorrow's landscape will be dominated by truly distributed systems – systems whose components are separated by tens to hundreds of milliseconds. This sea change will occur – indeed, has already started – because of the dramatic drop in price of computation, storage, and sensors relative to communication.

The cost for communication has dropped dramatically over the past twenty years: the amount of bandwidth that cost $100 in 1997 now costs only $4. However, the cost of computation has dropped even more dramatically: the amount of computation that could be purchased for $100 in 1997 now costs three *cents*. Both have dropped dramatically, but *computation has dropped by two orders of magnitude more than communication*[14]. This has a profound impact, because the price:performance of computation governs the rate of data production, consumption and storage. The price:performance of communication governs the rate of its transmission.

The general effect of programs is to reduce data taken from sensors or produce data to be consumed by people. The dramatic drop in the price of computation relative to communication means that in the future it will make much more sense to send programs to data and to people rather than to send data to programs or connect people to distant programs, an inversion of both the traditional role of networks and common IT practice. These price/performance trends imply that the future of networks is distributed systems, exploiting ubiquitous computation to minimize data transmission.

A network tuned to distributed systems will accelerate the trend towards thin clients as personal systems. Thin clients and Cloud applications have a number of advantages over traditional fat-client applications: data is protected against local failure, is available everywhere there is network connectivity, and is device-independent. Collaboration is a natural, almost built-in feature of Cloud applications. However, fat-client systems such as personal computers are still in widespread use, in part because connectivity is not yet ubiquitous and because today's centralized cloud imposes bandwidth limits and latency lower bounds on client-server communications. Only applications which are either latency-insensitive or whose data can be cached on the client can be used on thin clients. The utility of owning a personal fat-client computing devices is being steadily eroded by utility and cloud computing, with the principal limitations on the latter being the programmability of the browser and the responsiveness of the network, primarily due to latency concerns. A network with POPs everywhere dramatically increases the responsiveness of the network and relieves pressure on the browser platform.

The Edge Cloud combines the advantages of the Cloud with low latency and high bandwidth. This enables a broad range of services and applications to migrate to the cloud, which are currently restricted to the client for bandwidth or latency reasons. An edge cloud also permits a broad range of truly distributed services such as Content Distribution Networks[39], [13], Wide-Area Storage Systems[27], cloud-based interactive fat-data applications such as the Ignite Distributed Collaborative Visualizer[9], [8], [14], overlay multicast trees[12], wide-area measurement systems, distributed key-value stores[35], distributed map-reduce systems for in situ data reduction from distributed sensors. Many of these systems have been developed and deployed on standalone edge clouds such as the Global Environment for Network Innovations (GENI) and PlanetLab. Infrastructure-as-a-Service (IaaS) Edge Clouds such as GENI and PlanetLab have spread slowly, in part because hosting a PlanetLab node or GENI Rack is a

978-0-9883045-1-2/16 $31.00 © 2016 ITC

substantial burden. Both rely on a dedicated hardware investment, which in the case of the GENI Rack costs in the tens of thousands of dollars[3], [2], [23], which must be purchased, installed, maintained and refreshed. Even when a rack is donated to an institution, it can take weeks or months before the institution installs it. The value of an edge cloud is largely dependent on its ubiquity; it is therefore of great value to dramatically lower the cost of installing and maintaining an edge cloud node.

Technology has also changed substantially in the 13 years since PlanetLab was first conceived. In 2002, virtualization technology was nascent, and the Cloud didn't exist. Indeed, in some sense PlanetLab and Emulab[40], [36] were the world's first Clouds. In 2002, building an infrastructure required distributed and maintaining dedicated, specialized hardware. But today, clouds and virtualized infrastructure are ubiquitous, and changing the role of hardware resources is commonplace. This opens up a new possibility in deploying infrastructure – as a set of virtual machines which can run on any offered hosts. PlanetIgnite is an update of the PlanetLab/GENI Experiment Engine architecture which does exactly that.

The remainder of this paper is organized as follows. In Section II we describe the architecture of PlanetIgnite. We also describe how each constituent of PlanetIgnite (an application developer, an application deployer, and a host city) sees the infrastructure. In Section III we discuss the implementation of the PlanetIgnite Node Image. In Section IV, we discuss implementing the PlanetIgnite portal and node instantation and management software. In Section V, we discuss the on-node services available to PlanetIgnite developers, including the foundational Slice Control Service. In Section VI, we conclude and discuss future work.

II. PLANET IGNITE: A LIGHTWEIGHT EDGE CLOUD

PlanetIgnite is intended as a general-purpose, Infrastructure-as-a-Service, self-assembling, lightweight edge cloud on virtualized infrastructure with support for single-pane-of-glass distributed application configuration and deployment. This is an entirely new concept. PlanetLab, GENI, and SAVI are general-purpose IaaS edge clouds, but do not offer single-pane-of-glass application deployment, and require top-down installation and dedicated hardware resources at each site. Seattle is a lightweight self-assembling edge cloud that offers single-pane-of-class configuration and control, but developers are restricted to using a subset of Python. We propose an edge cloud as lightweight and easy to install as Seattle, but with the general-purpose properties or PlanetLab or GENI, which we call PlanetIgnite.

PlanetIgnite is based on our previous experience with the GENI Experiment Engine (GEE)[5], [6], [4]. The GENI Experiment Engine is a Containers-as-a-Service infrastructure that runs on the GENI infrastructure. The GEE offers Docker Containers as a Service, one per GEE node, to each user. It offers single pane-of-glass control through the use of off-the-shelf configuration tools, and a number of services instantiated in each user container, including an internode messaging service. The goal of PlanetIgnite is to use the GENI Experiment Engine as the prototype of a self-assembling self-growing edge cloud, extending its footprint well beyond the current GENI-based platform and to make instantiation of applications on the infrastructure far more easy and automatic. To a current GEE developer PlanetIgnite is intended to appear to be almost indistinguishable from the GEE, save that there will be many more GEE nodes; to a user/deployer of a pre-written PlanetIgnite application it is intended to appear to be a local cloud that instantiates applications immediately; to a PlanetIgnite site it is intended to be a VM image installed, as any other. We consider each of these PlanetIgnite constituents in order.

A. PlanetIgnite Application Developer

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[34] 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 Slice. When this process is completed (within a few seconds), a download link to a zip file appears on her dashboard. The user then downloads the file to her computer. The zip file contains files which include directions on using the site, Ansible and Fabric configuration files, Slice Control Service playbooks, and slice-specification authentication credentials.

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 Fabric file downloaded from the GEE site makes upload and execution as easy and quick (roughly) as uploading a Python program to the Google App Engine.

Fabric is one solution to single pane-of-glass control of a slice. 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 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 to build and configure the node. Both Ansible and Fabric are supported and used by, the GEE.

The user can tear down her experiment by using the "free slice" button on her dashboard. No configuration of the slice 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.

1) The Slice Directory: As mentioned above, once an application developer has allocated a slice, she is able to download a zip file, which, when unzipped, contains all of the configuration and authentication information needed to access and manipulate her slice from her local laptop. Here, we detail the contents of the slice file: the contents offer a concrete guide to the services available to the developer.

File	Purpose
id_rsa	Private key for slice access
id_rsa.pub	Public key corresponding to id_rsa
ansible.cfg	Configuration file for Ansible
ansible-hosts	Inventory of hosts for use with Ansible playbooks
fabfile.py	Starter Fabric file with configuration information
ssh-config	SSH Configuration file for ssh and ssh-tool access to the slice
slice-hosts.yaml	Example Ansible playbook
message-server.yaml	Ansible playbook to set up a message server in the slice
message-client.yaml	Ansible playbook to install a message client in the slice
README.txt	File detailing the contents and use of the directory

The first file is just the authentication information required to use the slice. The third and fourth files are used to configure Ansible; ansible.cfg directs Ansible to use slice<slice_num> as the remote user, and ansible-hosts is the inventory file for ansible. So, for example, `$ ansible -i ./ansible-hosts -m ping nodes` runs the `ping` module on every ansible host, verifying the ability to connect to it. fabfile.py is a Fabric file, pre-loaded with authentication information and with a few sample commands. `$ fab uptime` runs the `uptime` command on all hosts. `ssh-config` is a standard ssh configuration file for the nodes in the slice, with helpful shorthand access. `$ ssh -F ./ssh-config ig-uwashington` logs in to the GEE node at the University of Washington.

The three `.yaml` files are Ansible playbooks. The first is simply an example playbook, which runs the `ping` module on every node in the slice. It verifies connectivity for the user, offers a very simple example of how to write playbooks for this infrastructure, and creates a local Python file with a dictionary of hostnames and ip addresses for subsequent use by Python programs running locally or in the slice. `ansible-playbook -i ./ansible-hosts slice-hosts.yaml` creates this file, slice-hosts.py. The second playbook installs the Beanstalk[37] client on every node in the slice; the third installs a Beanstalk server on every node chosen as a `message_server` in the Ansible playbook.

We describe Beanstalk in more detail below. Here, we focus on the architecture of providing the end user with scripts to install optional services rather than pre-configuring them in the slice. First, this keeps slices lean: if a user doesn't want a service, she simply doesn't run the script. Second, this makes the slices highly customizable: the user is free to make her own choices for a messaging server rather than taking ours. Third, it makes adding services a lightweight and easy process: we simply add an Ansible playbook to the slice tarball.

B. PlanetIgnite User/Application Deployer

One of the motivating factors for PlanetIgnite was to broaden the footprint of US Ignite, and make it much easier for communities to use GENI services and offer instantly-deployable GENI applications. A key part of this integration is with the Ignite Smart Gigabit App Store. A PlanetIgnite User/Application deployer follows a similar path to a GEE Developer through allocating a slice. However, once the slice is allocated, he deploys an application simply by choosing a pre-packaged application from the Ignite Gigabit App Store.

1) Smart Gigabit App Store: A smart and connected communities end user can also start an application on GENI or PlanetIgnite through the US Ignite Smart Gigabit App Store. The Gigabit App Store's graphical user interface allows them to choose an application to invoke. The App Store will then collaborate with GEE to instantiate that application on one of the nearest eligible GENI or PlanetIgnite VM instance and provide a link to its browser control port. The browser may provide the application as a high-bandwidth and/or low-latency web app. Alternatively, for some applications, the URL returned will be a URL for a RESTful API instance called by a client application. Finally, for applications using the security and protection of network slicing, the URL will contain the information needed to establish an encrypted tunnel to the slice, or, the slice itself may appear at an end-point specified by the user during the application invocation when slicing can be extended to the end-user. The US Ignite Gigabit Application Store will be built on top of the existing Collaborative Community Exchange (CCX) which can be seen in operation at `https://us-ignite.org/apps`. Instead of getting only a description of the application, a button or link will allow the application to be invoked for the user. During the invocation time, the user will be informed of the slice name being created and its expiration date and time and the specific PlanetIgnite node on which it is being instantiated.

Like GENI, the US Ignite Smart Gigabit App Store will provide single sign-on to apps via Shibboleth[25] and InCommon[17]. Citizens of US Ignite testbed communities will be given extended credentials upon application to their Community Coordinator or designee, and may have to provide evidence of their identity. This approach will automatically allow most college and university faculty,

staff, and students to have access to the Gigabit App Store since they already have InCommon credentials. The CCX currently federates with Mozilla Persona [26] for identity management and will transition to InCommon when Persona is decommissioned in November.

C. PlanetIgnite Node Site

PlanetIgnite is a self-configuring programmable and sliceable server infrastructure designed to self-assemble an interoperable and interconnected software-defined locavore applications platform. GENI and PlanetIgnite servers are in-community and will have direct access to gigabit access networks and services with no backhaul charges. By keeping bits in-community, ultra-low-latency apps will become possible. And communities can become more digitally self-sufficient by bringing the cloud to the community instead of running their apps in a distant cloud.

A PlanetIgnite node can be installed anywhere using a standard open source image. Communities can add PlanetIgnite facilities as they can afford them. Local companies or universities may turn over some of their own VMs to PlanetIgnite instances during off-peak hours. Individual donors can put up one or more servers on their gigabit access networks to help serve the community. Charities might contribute resources to keep gigabit apps of use to low-income and underserved communities available. Libraries may choose to run PlanetIgnite facilities to serve their own patrons, either in or outside the library. PlanetIgnite nodes can be projected into low-income neighborhoods to bridge the digital divide.

PlanetIgnite will reduce the cost of entry for a community to add a platform to execute server apps at a location near the user where there is a clear gigabit path to the user and ultra-low latency. Where the access infrastructure permits, network slicing will be used to provide a protected path for public safety, medical, or sensitive information. Where the access structure does not have this capability built-in, encrypted channels will be used instead until a full sliced infrastructure becomes available.

Mechanically, to become a PlanetIgnite site, a community will simply go to the PlanetIgnite portal and fill out a short web form, download the PlanetIgnite VM image and a boot script derived from the web form. The boot script will contain site configuration parameters. The node will automatically register with PlanetIgnite. Qualification test will be done automatically.

III. The PlanetIgnite Node Image

The PlanetIgnite node image is an Ubuntu 14.04 node image which uses Docker as a container manager. An ongoing issue with PlanetLab, one of our immediate predecessors, is that the set of virtual machine images that may currently be instantiated by users on PlanetLab is limited to those that have been manually installed by PlanetLab's administrators. Creating new images on PlanetLab is not an automated process, and installing new images requires manual intervention. As the set of available images is limited, they often differ from the operating system that a developer is using locally, leading to software compatibility issues when a service is deployed from a local setting to a distributed PlanetLab setting. For example, the stock images available on PlanetLab often lack the specific version of libraries or Java Development Kit (JDK) that a PlanetLab user is expecting. Pushing new images with software patches, such as security updates, is seldom done due to the manual intervention required with image management. Researchers have consistently requested a way of custom-deploying their own filesystem images with the latest Linux distributions and flexible control.

Docker[24] is an open platform for building, shipping, and running distributed applications based on LXC[21]. Docker supports a layered image structure where templates are unioned together to create an image. This allows for easy and space-efficient extension of existing images to form new images. Users can locally instantiate Docker images, so their local environment may be identical to the PlanetIgnite environment, eliminating software compatibility issues. A wide selection of existing Docker images is available, leading to a useful starting point for PlanetIgnite users and developers.

Docker includes a registry that stores images, facilitating both storage of private images and sharing of public images. This eliminates the step where PlanetIgnite staff is required to manually install and deploy new images. Image creation with Docker is sufficiently lightweight that it also may solve the software distribution problem, in the respect that a PlanetIgnite user may build custom packages directly into his or her Docker image. By making the image update and deployment process less painful, the frequency and likelihood that new images will be created with patches and security updates will increase.

We leverage Docker on PlanetIgnite to automate image building and distribution tasks. As an initial step we will allow users to select from a number of Docker images that have been vetted by PlanetIgnite staff as providing environments sufficient for the majority of users (e.g., for major distributions like Ubuntu and CentOS). Ultimately we will enable users to supply their own custom Docker images.

We have successfully used Docker in PlanetIgnite's immediate precursor and prototype, the GEE. We use Docker as the container manager service on the slice. Docker is essentially an overlay on a Linux container solution, either using libvirt[20] and LXC 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

flavor of Linux. 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.

A. Monitoring With Collectd

Monitoring is essential to determine the performance and correct functioning of a Slice. Historically, PlanetLab has provided the CoMon tool[29] in order to return information about Slices, such as the amount of memory used, number of processes running, CPU utilization, bandwidth, and so on. CoMon has been decommissioned and this service is no longer available to PlanetLab users, leaving a void to be filled.

The collectd tool[16] is a mechanism for collecting performance data. The focus of collectd is on modularity and expandability. Collection is done via plugins; a robust set of plugins is already available. New plugins may be written and deployed to expand the scope of collection, allowing collectd to be extended to collect PlanetLab-specific performance data. In addition to collecting data, plugins are also used to store and push data, and collectd includes a Network Plugin that may be used to push data to a central server. This provides us with an end-to-end system from data collection on the individual nodes to storage and aggregation on a central server. Another technology, that may be used either in conjunction with collectd or used on its own, is BigQuery[38]. BigQuery is an append-only database service. Its particular focus is on massive datasets, making it an optimal choice for storing data collected by large sets of distributed machines. It supports an SQL-like language for querying the database, allowing users to easily craft tools for generating custom reports. BigQuery is integrated with Google App Engine and Google Spreadsheets, further facilitating reporting of data. By outsourcing the data storage to a distributed service like BigQuery, we gain the advantage of scalability while avoiding the pitfall of having to maintain a reliable scalable service ourselves. We have prototyped a distributed collection system using BigQuery, and a set of views for data analysis. We propose to build a CoMon replacement using collectd and BigQuery.

IV. The PlanetIgnite Back End

Though Docker is 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. Its principal functions are 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 slice zip file from templates.

The value of Ansible and Docker was easy to see: the Ansible slice-creation YAML file was only 57 lines of markup, and the script 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.

A. The PlanetIgnite Portal

The interface for a user to create and manage slices 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[31] cluster. 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 slices.

The first container has a Mongo [33] 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.

1) The Developer and User Facing Portal:

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.

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 slices 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.

Use-once Keys: 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 slice is compromised, or the use-once key is discovered, all that is compromised is the user's slice. 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 slice.

Use-once keys are the infrastructure equivalent of hotel room cardkeys; they are allocated when the slice is instantiated, used only to access the slice, and are destroyed when the slice 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.

The Site Facing Portal: The Site-Facing Portal is designed to permit sites to easily register as a PlanetIgnite site and automatically download the image and install it on a VM. While the portal will export a visual interface for manual entry, registration will primarily be done automatically through the downloaded and installed image.

When a PlanetIgnite image comes up on a local site, it will offer two configuration options. The first is a simple boot script, which takes as an argument the routable IP address of the VM, configures the local network interfaces, and then uses a REST http interface to automatically register the image with the portal. The portal not only register the image but invokes, through the local node manager, a series of tests which determine whether the node is up, functioning, and not behind a firewall which blocks access to required ports. If the node passes these tests, it is added to the database of available PlanetIgnite nodes.

V. The PlanetIgnite Developer Services

PlanetIgnite will follow the GENI Experiment Engine in providing a number of services to developers on the PlanetIgnite nodes. Here, we detail the initial services.

A. The PlanetIgnite Slice Control Service

The foundational service is the PlanetIgnite Slice Control Service, which permits PlanetIgnite slices to be configured and controlled through a single pane of glass. It is the foundational service because we leverage this service to bootstrap the other services that we offer.

Scalability of control for a distributed application is critical. Slice management and configuration was the focus of a large number of early PlanetLab efforts[1], [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 use two as the basis for the Slice Control Service, 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.

Both Ansible and Fabric have 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 Slice 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. This insight led to the fundamental idea of the Slice Control Service as the foundation on which the other services could be bootstrapped. Rather than pre-loading services into slices, or building services slices, we could simply build Fabric commands or Ansible playbooks to instantiate and install the service. We have tested this approach with the Slice Message Service.

B. The Slice Message Service

The Slice Message Service is used to route job control messages within a slice; 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 slice, and a client library; a user activates the server on whichever nodes in the slice 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[37]. 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 is to coordinate tasks within a data center. Its use mode is not a multi-tenant provider who offers messaging-as-a-service, like IronMQ[18], but rather that each job or service instantiate its own messaging server accessible only from its own nodes: security is assumed at the slice level. This dictated our deployment choice: rather than instantiating a GEE-wide messaging service, we offer the developer an Ansible playbook to turn the service on in the appropriate slivers, and choose the appropriate server site.

C. The Slice Storage Service

PlanetLab users have historically relied on multiple community provided tools and services for deploying data to slivers and getting data back from experiments. These include CoDeploy[28], Stork[10], PLuSH[1], and a variety of ad-hoc ssh scripts. To simplify this common use-case (as well as replace tools that are no longer maintained), we will use Syndicate[27] to give each slice a shared, read-write private storage volume and make available public read-only volumes of popular datasets.

Syndicate is a scalable software-defined storage system, under development at Princeton, that combines existing CDNs and cloud storage into a coherent, wide-area read/write storage medium. With Syndicate, we will augment existing PlanetLab-hosted CDNs (like Coral and CoBlitz[30]) with commodity cloud storage (like Dropbox and Amazon S3) to give each slice a shared private storage space. Syndicate readers pull data from one another via the CDNs, thereby scaling up aggregate read bandwidth beyond what a single sliver can provide. Syndicate writers push data into commodity cloud storage to keep it highly available in the face of node failures. To keep slice data secure, Syndicate implements end-to-end encryption and cryptographic signatures to guarantee data confidentiality, integrity, and authenticity.

Deploying and gathering data with Syndicate is straightforward. Syndicate's client is a FUSE file system, so users deploy data to slivers simply by copying data in, and retrieve sliver-generated data simply by copying data out.

PlanetLab users have frequently requested a global filesystem for sharing data. In addition to moving data between slivers, we will use Syndicate to expose existing, public datasets and repositories as read-only volumes. This will save users the time and effort of having to push data to upwards of 1000 nodes, while ensuring that slivers always have access to it even if they are re-imaged. For example, we will use Syndicate to expose geo-IP databases, software repositories, and public scientific datasets as mountable volumes.

D. The Slice Reverse Proxy Service

Intra-slice traffic on the PlanetIgnite will primarily be through a network private to the slice. Routable IPs are notoriously scarce on PlanetLab nodes, and GENI member institutions have been unable to devote large banks of routable IP addresses to GENI slices. Our goal is to permit sites to easily download and instantiate PlanetIgnite nodes. If a typical site has an eight-core dual-socket node - a typical InstaGENI worker node - we should be able to accommodate between 80 and 160 slivers on this node. This will require a /25 or /24 to give each sliver a routable IP, and many of our candidate sites will not have a /24 sitting in their hip pocket to hand to us. Clearly, we cannot count on being able to give a routable IP to each sliver.

Though the private network suffices for intra-slice traffic, a number of PlanetLab slices and services offered distributed public services. Clearly, for such services to use PlanetIgnite, 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. IPv6 is the obvious solution, but it isn't implemented on many campuses.

If our developers 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 using the PlanetIgnite Reverse Proxy. The 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>/<slicename>/<request>

are caught by the reverse proxy and sent to the http server in the slice's sliver over the GEE private network; the returned value is sent back to the requester.

VI. Conclusions

PlanetIgnite is currently near-deployment. We plan a graceful transition from the current GEE, using the PlanetIgnite technology ourselves to automatically deploy PlanetIgnite nodes onto ExoGENI and SAVI slices, Chameleon and CloudLab nodes. Once we have this process smoothly automated, we will transition this onto a subset of PlanetLab's existing nodes, then offer this to early beta communities.

At the end of the day, PlanetLab, GENI, the GENI Experiment Engine, and PlanetIgnite are about an idea: it should be as easy for a developer to ship a program to a computational element as it is for a user to download data over the Internet today. In this work, we have largely solved the major technological problem in doing this: we have presented the developer with an homogeneous execution environment across the wide area, much as cellphone OS's created a homogeneous, ubiquitous client. This is, however, only half the battle: what remains is the far more challenging enterprise of persuading sites to permit untrusted third parties to run programs on their sites: in effect, to ask each site to host a node on a nationwide distributed Cloud.

Solving this problem involves a threefold strategy: increasing the value of hosting a Cloud site *to the host institution*; radically reducing the cost of hosting a Cloud site; and developing the web of agreements, acceptable use policies, and liability structures to reduce the risk to the host instituion of hosting a Cloud site.

This paper is targeted at the second element, much as our companion paper[15] addressed the first. We do not neglect the third. At the moment, we use the GENI Acceptable Use Policy, but will work with sites and our partners to derive one suitable for a worldwide infrastructure.

We are committed to offering these services wherever a Docker node can be instantiated. Our view of the cloud is that it should reach as close to the edge as is feasible given the underlying implementation technology. As a result, we are exploring developing PlanetIgnite for Paradrop WiFi routers[41]. An existing use case for the latter is ensuring multicast to co-located high-definition streaming clients.

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; Marshall Brinn of the GPO and Nick Bastin of Barnstomer Softworks for productive conversations; Chip Elliott of the GPO for years of guidance; Joe Kochan and the staff at US Ignite; the technical team at US Ignite, particularly Nishal Mohan and Scott Turnbull; 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.

References

[1] J. 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 *Twenty-first USENIX Large Installation System Administration Conference, LISA '07*, November 2007.

[2] I. Baldin, C. Castillo, J. Chase, V. Orlikowski, Y. Xin, C. Heermann, A. Mandal, P. Ruth, and J. Mills. Exogeni: A multi-domain infrastructure-as-a-service testbed. In *The GENI Book*, chapter 13. Springer-Verlag, New York, 2016.

[3] 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.

[4] A. Bavier and R. McGeer. The geni experiment engine. In *The GENI Book*, chapter 11. Springer-Verlag, New York, 2016.

[5] A. C. Bavier, J. H. Chen, J. Mambretti, R. McGeer, S. McGeer, J. C. Nelson, P. O'Connell, G. Ricart, S. Tredger, and Y. Coady. The GENI experiment engine. In *2014 26th International Teletraffic Congress (ITC), Karlskrona, Sweden, September 9-11, 2014*, pages 1–6, 2014.

[6] A. C. Bavier, J. H. Chen, J. Mambretti, R. McGeer, S. McGeer, J. C. Nelson, P. O'Connell, G. Ricart, S. Tredger, and Y. Coady. The GENI experiment engine. *EAI Endorsed Trans. Ubiquitous Environments*, 2(6):e2, 2015.

[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] S. Bhojwani, M. Hemmings, D. Ingalls, J. Lincke, R. Krahn, D. Lary, P. McGeer, G. Ricart, M. Röder, Y. Coady, and U. Stege. The ignite distributed collaborative scientific visualization system. In *7th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2015, Vancouver, BC, Canada, November 30 - Dec. 3, 2015*, pages 186–191, 2015.

[9] S. Bhojwani, M. Hemmings, D. Ingalls, J. Lincke, R. Krahn, D. Lary, R. McGeer, G. Ricart, M. Roder, Y. Coady, and U. Stege. The ignite distributed collaborative visualization system. *SIGMETRICS Perform. Eval. Rev.*, 43(3):45–46, Nov. 2015.

[10] 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 *The 21st Large Installation System Administration Conference '07*, 2007.

[11] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: A platform for educational cloud computing. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 111–115, New York, NY, USA, 2009. ACM.

[12] 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.

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

[14] M. Hemmings, R. Krahn, D. Lary, R. McGeer, M. Roeder, and G. Ricart. The ignite distributed collaborative scientific visualization system. In *The GENI Book*, chapter 19. Springer-Verlag, New York, 2016.

[15] M. Hemmings, R. Krahn, R. McGeer, G. Ricart, M. Roeder, and U. Stege. Livetalk: A framework for collaborative

browser-based replicated-computation applications. In *International Teletraffic Congress*, 2016.
[16] K. Horst. *Collectd*. Dign Press, 2011.
[17] Incommon: Security, privacy and trust for the research and education community. "https://www.incommon.org".
[18] Ironmq v2 reference. http://dev.iron.io/mq/.
[19] A. Leon-Garcia and H. Bannazadeh. Savi testbed for applications on software-defined infrastructure. In *The GENI Book*, chapter 22. Springer-Verlag, New York, 2016.
[20] Libvirt: The virtualization api. https://libvirt.org/. Accessed: 2016-03-16.
[21] Linux containers. https://linuxcontainers.org/. Accessed: 2016-03-16.
[22] R. McGeer, M. Berman, C. Elliott, and R. Ricci, editors. *The GENI Book*. Springer-Verlag, New York, 2016.
[23] R. McGeer and R. Ricci. The instageni project. In *The GENI Book*, chapter 14. Springer-Verlag, New York, 2016.
[24] I. Miell and A. H. Sayers. *Docker in Practice*. Manning, 2016.
[25] R. L. Morgan, S. Cantor, S. Carmody, W. Hoehn, and K. Klingenstein. Federated security: The shibboleth approach. *EDUCAUSE Quarterly*, 27(4):12–17, 2004.
[26] Mozilla persona: A better way to sign in. https://login.persona.org/.
[27] 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.
[28] K. Park and V. S. Pai. Deploying large file transfer on an http content distribution network. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS 04)*, 2004.
[29] K. Park and V. S. Pai. Comon: A mostly-scalable monitoring system for planetlab. *SIGOPS Oper. Syst. Rev.*, 40(1):65–74, Jan. 2006.
[30] K. Park and V. S. Pai. Scale and performance in the coblitz large-file distribution service. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation-Volume 3*, pages 3–3. USENIX Association, 2006.
[31] 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, September 2011.
[32] 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.
[33] M. Pirtle. *MongoDB for Web Development (1st ed.)*. Addison-Wesley Professional, March 2011.
[34] 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.
[35] S. C. Rhea. *Opendht: A Public Dht Service*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2005. AAI3211499.
[36] R. Ricci. Emulab. In *The GENI Book*, chapter 2. Springer-Verlag, New York, 2016.
[37] A. Rodrigues. Beanstalkd: a simple and reliable message queue. http://www.artur-rodrigues.com/tech/2015/06/04/beanstalkd-a-simple-and-reliable-message-queue.html.
[38] J. Tigani and S. Naidu. *Google BigQuery Analytics*. Wiley, 2014.
[39] 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.
[40] 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 *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.
[41] D. F. Willis, A. Dasgupta, and S. Banerjee. Paradrop: A multi-tenant platform for dynamically installed third party services on home gateways. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing*, DCC '14, pages 43–44, New York, NY, USA, 2014. ACM.