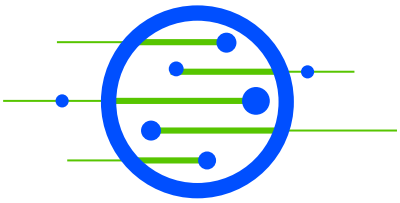

PlanetLab Architecture: An Overview

Larry Peterson*, Steve Muir*, Timothy Roscoe†, Aaron Klingaman*

* Princeton University
† Intel Research – Berkeley

PDN–06–031
May 2006

Status: Ongoing Draft.

PlanetLab Architecture: An Overview

Larry Peterson, Steve Muir, Timothy Roscoe, Aaron Klingaman

May 5, 2006

Contents

1 Introduction

PlanetLab has evolved rapidly over the past three years according to a set of design principles [9], but without formal documentation of its underlying architecture. This document addressees this shortcoming by defining the main architectural elements of PlanetLab. It is the first in a series of documents that collectively define Verion 4 of the PlanetLab architecture.[1] Subsequent documents define specific elements (and their interfaces) in detail:

- Slice and Management Authorities: Interface Specifications [4].
- Node Manager and Slice Creation Service: Interface Specifications [6].
- Securely Booting PlanetLab Nodes: Reference Implementation [3].
- PlanetFlow and Audit Archive: Reference Implementation [2].
- Virtualized Execution Environments: Reference Implementation.

Note that with the exception of the last document, this series defines PlanetLab at a global level; it is not intended to serve as a reference manual for typical users. Also note that while pseudo-code corresponding to various interface calls are included in this document for illustrative purposes, the specification documents referenced above should be consulted for the authoritative definitions.

We use two sidebars to discuss tangential issues:

> **Implementation Note:** Identifies ways in which the current implementation falls short of the architecture, or ways in which the current implementation might be trivially extended to support a cleaner architectural definition.

and

> **Evolution Note:** Discusses ways in which the architecture and implementation might evolve in the future.

2 Organizing Principles

PlanetLab is designed around five organizing principles, each stemming from either a user requirement or a constraint of the environment in which it operates. This section identifies these requirements and the organizing principles they inspire. In the process, it introduces some of the key concepts underlying PlanetLab.

[1] Version 3.2 is currently deployed on PlanetLab. Version 4 is based on feedback from Version 3 [8], rationalized to better support federation and to provide a more complete security architecture.

2.1 Distributed Virtualization

PlanetLab's primary goal is to provide a global platform that supports broad-coverage *services* that benefit from having multiple points-of-presense on the network. PlanetLab simultaneously supports two usage models. PlanetLab *users*—typically researchers and service developers—either run short-term experiments, or deploy continuously running services that support a client workload.

PlanetLab supports this usage model by supporting *distributed virtualization*—each service runs in a *slice* of PlanetLab's global resources. Multiple slices run concurrently on PlanetLab, where slices act as network-wide containers that isolate services from each other.

2.2 Unbundled Management

PlanetLab faces a dilemma: it is designed to support research in broad-coverage network services, yet its management plane is itself a widely distributed service. It was necessary to deploy PlanetLab and start gaining experience with network services before we fully understood what services would be needed to manage the platform. As a consequence, PlanetLab had to be designed with explicit support for evolution.

To this end, PlanetLab decomposes the management function into a collection of largely independent *infrastructure services*, each of which runs in its own slice and is developed by a third-party, just like any other service. We refer to this decoupling as *unbundled management*. For example, an infrastructure service might create slices on a set of nodes; buy, sell and trade node resources; keep the code running in a slice up-to-date; monitor a slice's behavior, and so on.

2.3 Chain of Responsibility

PlanetLab takes advantage of nodes contributed by research organizations around the world. These nodes, in turn, host services on behalf of users from other research organizations. The individual *users* are unknown to the node *owners*, and to make matters worse, the services they deploy are likely to send potentially disruptive packets into the Internet. The *PlanetLab Consortium* (PLC) plays the role of a trusted intermediary, thereby freeing each owner from having to negotiate a hosting agreement with each user.

An important responsibility of PlanetLab is to preserve the *chain of responsibility* among all the relevant principals. That is, it must be possible to map externally visible activity (e.g., a transmitted packet) to the principal(s) i.e., users, responsible for that packet. The ability to do this is essential to preserving the trust relationships among various parties. Note that the chain of responsibility does not attempt

to eliminate the possibility that bad things might happen, it just requires that the system be able to identify the responsible party when something does go wrong.

2.4 Decentralized Control

PlanetLab is a world-wide platform constructed from components owned by many autonomous organizations. Each organization must retain some amount of control over how their resources are used, and PlanetLab as a whole must give countries and geographic regions as much autonomy as possible in defining and managing the system.

As a consequence, PlanetLab must support *decentralized control*, which in turn requires minimizing the aspects of the system that require global agreement. Doing so allows autonomous organizations to federate in the formation of a global facility and for these organizations to define peering relationships with each other.

2.5 Efficient Resource Sharing

While a commercial variant of PlanetLab might have sufficient resources (and cost recovery mechanisms) to ensure that each slice can be guaranteed all the resources it needs, PlanetLab must operate in an under-provisioned environment. This means conservative allocation strategies are problematic, and it is necessary to promote efficient resource sharing. Guarantees should be possible, but the system must also support "best effort" slices and a flexible way for slices to acquire needed resources.

In this context, PlanetLab adopts adopts a two-pronged resource allocation strategy. First, it decouples slice creation and resource allocation. This means all slices are given only best effort promises when they are first created. They then acquire and release resources over time, assisted by available *brokerage services*. It is not the case that a slice is bound to a guaranteed set of resources during its entire lifetime. Second, even though some slices can acquire guaranteed resources for a period of time, we expect overbooking to be the norm. PlanetLab must provide mechanisms that recover from thrashing due to heavy resource utilization.

3 Principals and Trust Relationships

The PlanetLab architecture recognizes three main principals:

- An *owner* is an organization that hosts (owns) one or more PlanetLab nodes. Each owner retains ultimate control over their own nodes, but delegates responsibility for managing those nodes to the trusted PLC intermediary. PLC

provides mechanisms that allow owners to define resource allocation policies on their nodes.

- A *user* is a researcher that deploys a service on a set of PlanetLab nodes. PlanetLab users are currently limited to individuals employed by research organizations (e.g., universities, non-profits, and corporate research labs), but this is not an architectural requirement. Users create slices on PlanetLab nodes via mechanisms provided by the trusted PLC intermediary.

- The *PlanetLab Consortium* (PLC) is a trusted intermediary that manages nodes on behalf a set of owners, and creates slices on those nodes on behalf of a set of users.

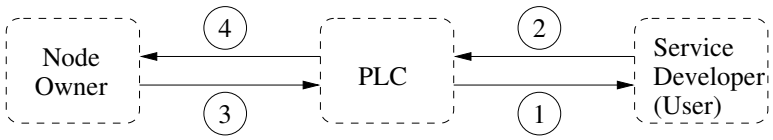

Figure 1: Trust relationships among principals.

Figure 1 illustrates the trust relationships between node owners, users, and the PLC intermediary. In this figure:

1. PLC expresses trust in a user by issuing it credentials that lets it access slices. This means that the user must adequately convince PLC of its identity (e.g., affiliation with some organization or group).

2. A user trusts PLC to act as its agent, creating slices on its behalf and checking credentials so that only that user can install and modify the software running in its slice.

3. An owner trusts PLC to install software that is able to map network activity to the responsible slice. This software must also isolate resource usage of slices and bound/limit slice behavior.

4. PLC trusts owners to keep their nodes physically secure. It is in the best interest of owners to not circumvent PLC (upon which it depends for accurate policing of its nodes). PLC must also verify that every node it manages actually belongs to an owner with which it has an agreement.

While PLC was originally the only trusted intermediary in PlanetLab, it is rapidly becoming the case that other other entities (e.g., national and regional

centers, private intranets) wish to play a PLC-like role. This means PlanetLab is quickly becoming a federation of such systems. While this move to federation is still in progress, and we cannot accurately predict its final form, we can start by identifying the distinct roles PLC plays, and explicitly decoupling them in the architecture.

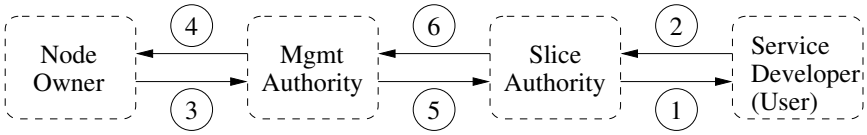

Figure 2: Trust relationships among principals.

Specifically, PLC plays two roles: it manages a set of nodes on behalf of owners, and it creates slices on behalf of users. Accordingly, PLC is both a *management authority* (MA) and a *slice authority* (SA). Explicitly decoupling these two roles allows new PLC-like entities to evolve these two elements independently. Figure 2 shows PLC split into these two authorities. The split adds a new link in the chain of trust relationships between owners and users:

5. To support auditing, each management authority (and indirectly, the owners it represents) trusts slice authorities to reliably map slices to users. An owner may choose a different management authority if it cannot reliably resolve complaints, or may bypass its management authority and configure its nodes to blacklist certain slices or slice authorities.

6. Each slice authority (and indirectly, the users it represents) may trust only certain management authorities to (a) provide it with working VMs, and (b) not falsely accuse its users of out-of-bounds behavior. A user is always free to avoid nodes run by rogue management authorities.

From an architectural standpoint, the key feature of the trust relationships is the chain of pairwise trust relationships stretching across principals. In other words there are *no* required trust dependencies between owners and users. Of course, the slice and management authorities are just agents, and owners and users are always free to bypass their agent authorities and forge direct trust relationships (e.g., a node owner can blacklist slices). However, this should not be necessary if the authority is doing its job. A secondary feature is that there is no inherent dependency between management and slice authorities. They are free to evolve independently of each other.

This decoupling offers an important degree of freedom as the system evolves. For example, multiple autonomous regions can run their own management authorities, while a single global slice authority grants service developers access to nodes across management boundaries. This distributes the node management problem without balkanizing slices. Similarly, there can be multiple slice authorities that may or may not be coupled with management domains. For example, there might remain a single "research" slice authority, corresponding to today's PLC, along with a "public good" slice authority that approves only those service developers that offer network reports or SETI@HOME-style services, and a "commercial" slice authority that represents for-profit services. There can also be private slice authorities that allow users at a site to access site-specific resources not made available outside the site. In a sense, a slice authority is analogous to a virtual organization [1].

4 Architectural Components

This section describes the architectural elements that collectively define PlanetLab. These elements consist of two major pieces: (1) a set of software modules that run on each node, and (2) a collection of global mechanisms that implement PLC.

4.1 Node

A *node* is a machine capable of hosting one or more *virtual machines* (VM). A node must have at least one non-shared (not NAT'ed) IP address. Each node is identified by a unique node_id, which is bound to a set of attributes for the node; e.g., the hosting site (denoted by a network prefix), the node's current IP address, the node's MAC address, and so on. With the exception of the site network prefix, these attributes are allowed to change over time. This effectively allows the hardware that implements the node to be replaced, either incrementally or in whole, but a node (and its node_id) cannot migrate from one site to another.

It is not a requirement that a node's IP address be universally routable, but it all nodes must be reachable from the site at which PLC components run (see Section 4.7 and 4.8).

> **Implementation Note:** Today, a node's IP address must be statically assigned, although it can be served by DHCP. This limitation does not seem to be fundamental, and is likely to be removed in the near future.

There is usually a one-to-one mapping between nodes and physical machines, but this is not a requirement. It is possible for a node to be implemented inside

a virtual machine, perhaps even one that migrates from one physical machine to another, as might be the case on a cluster. In such a situation, however, a non-shared IP address must be assigned to the node. As a practical matter, changing the IP address assigned to a node is expected to be an infrequent operation. There is no requirement that all nodes be of the same machine architecture, although the current implementation is limited to x86 processors.

The architecture assumes that there is a means by which PLC can remotely reboot a node. The preferred implementation is an on-machine reboot capability (e.g., HP's Lights-Out product), although other implementations are possible. The mechanism should also support console logging.

A node boots with three pieces of state in a persistent, write-protected, store: a bootfile, a network configuration file named plnode.txt, and a public key for PLC. All three pieces of information are created through an offline process involving the node owner and PLC, and installed in the node by the owner.

The bootfile is an executable image that the node is configured to run whenever it boots. This program causes the node to interact with PLC to download all necessary software modules (see Section 4.8). plnode.txt is a node-specific text file that is read by the executable booted from the bootfile. It gives various attributes for the node, including its node_id, DNS name, and IP address, along with a unique nodekey generated by PLC and assigned to the node. The following is an example plnode.txt file:

```
IP_METHOD = "dhcp"
IP_ADDRESS = "128.112.139.71"
HOST_NAME = "planetlab1.cs.foo.edu"
NET_DEV = "00:06:5B:EC:33:BB"
NODE_KEY = "79efbe871722771675de604a2..."
NODE_ID = "121"
```

> **Implementation Note:** Today, the bootfile is available on a CD, and plnode.txt is available on either a floppy or USB device. In general, they could be combined on a single device.

The node uses the nodekey from plnode.txt during the boot process to authenticate itself to PLC (see Section 4.8). PLC trusts that the node is physically secure, corresponding to edge (4) in Figure 1.

4.2 Virtual Machine

A *virtual machine* (VM) is an execution environment in which a slice runs on a particular node. VMs are typically implemented by a *virtual machine monitor* (VMM)

running on the node, but this is not a requirement. Whatever the implementation, the architecture makes the following assumptions about the VMs that a node hosts:

- VMs are isolated from each other, such that
  - the resources consumed by one VM do not unduly effect the performance of another VM;
  - one VM cannot eavesdrop on network traffic to or from another VM; and
  - one VM cannot access objects (e.g., files, ports, processes) belonging to another VM.
- Users have the ability to securely access (remotely log into) VMs created on their behalf.
- The VM executes a user-provided bootscript each time the VM is started.
- Users are allowed to install software packages in their VM without consideration for the packages installed in other VMs running on the same node.
- Outgoing network traffic can be audited, such that it is possible to determine time, duration, and flow (source/destination addresses and ports) of all traffic originating from a given VM.
- To support infrastructure services that require access to private state in another VM, it should be possible to grant one VM access to state in another VM. Such access should be granted in adherence to the principle of least privilege.

The architecture is somewhat vague on the performance isolation issue because we wish to permit overbooking. A more accurate statement is that it should be possible to completely isolate a VM from the resources consumed by other VMs—i.e., it must be possible to make a resource guarantee to a VM—but not all VMs will take advantage of this option. Among a set of VMs without guarantees, the expectation is that resources are shared fairly.

> **Evolution Note:** While VMs are typically implemented by a VMM, it should be possible to allocate an entire physical processor to a VM. In this case, it must be possible to reclaim the processor from one VM and allocate it to another. If we take this broader view of an execution environment, then we should probably return to PlanetLab's original

terminology of calling the instantiation of a slice on a single node a *sliver*, but we retain the VM-based nomenclature for the purpose of this document.

Each VM is specified (abstractly represented) by a set of *attributes*, called a *resource specification* (RSpec). An RSpec defines how much of the node's resources are allocated to the VM; it also specifies the VM's *type*. PlanetLab currently supports a single Linux-based VMM, and so defines a single VM type (linux-vserver-x86). However, other VM types are possible (e.g., linux-xen-x86).

In general, the precise execution environment provided by a VM can be defined along several dimensions, and hence, the type system for VMs could become arbitrarily complex. This is not a desirable situation, as one of PlanetLab's most important properties today is that VMs are homogeneous. As our two examples illustrate, we currently define a VM along three dimensions: the operating system (which roughly defines the available programming interface), the underlying VMM (which indicates whether the user is allowed to install new software in the kernel and mediates how VMs communicate with each other), and the machine architecture (which defines what binaries run in the VM).

Evolution Note: Going forward, it seems reasonable to require that each node supports at least one of a small set of default VM types, and this set should most likely include linux-*-x86. Perhaps this is a requirement of all nodes on the "public" PlanetLab managed by PLC, while "private" PlanetLabs are free ignore this requirement.

A given physical machine might support more than one VM type, which forces us to be careful in how we describe the situation. Suppose a single machine runs Xen as its VMM, and hosts multiple VMs of type linux-xen-x86. Suppose one of the machine's Xen domains runs a Linux-based VMM that is capable of allocating VMs of type linux-vserver-x86. This physical machine could be treated as a single node that hosts two VM types, but since Xen allocates a unique IP address to each domain, it is more accurately viewed as two separate nodes: one virtual and supporting VMs of type linux-vserver-x86, and one physical and supporting VMs of type linux-xen-x86.

Each VM is initialized with two pieces of state, both provided in the RSpec given when the VM is created: a set of keys and a bootscript. Both are installed in the VM when it is created, and remain in the VM's permanent store until it is destroyed. The keys allow the users associated with the slice to remotely access the VM using a VM-specific mechanism (e.g., ssh). The bootscript is executed in a VM-specific way each time the VM boots.

There is a distinguished VM running on each node that provides the node owner with access to the node. This VM is known as site_admin for the purpose of remotely logging into the node, although we sometimes refer to it as the "owner-VM". The owner-VM is granted certain privileges not afforded other VMs, including the right to inspect network traffic belonging to any local VM (e.g., run tcpdump) and the right to execute certain privileged operations on the local node manager (see Section 4.3). Owners use this latter capability to run a script that expresses the owner's preferences on how the node's resources are allocated.

4.3 Node Manager

A *node manager* (NM) is a program running on each node that creates VMs on that node, and controls the resources allocated to those VMs. All operations that manipulate VMs on a node are made through the node manager; the native VMM interface is not called directly by any other module defined in this document. There is a one-to-one mapping between nodes and node managers. The NM interface can only be called locally; VMs are remotely created indirectly through one or more infrastructure services running in regular (unprivileged) VMs. The NM typically runs in a privileged (root) VM of the underlying VMM.

The NM provides an interface by which infrastructure services running on the node create VMs and bind resources to them. In addition to providing a VMM-neutral interface for creating VMs, the node manager also supports a *resource pool* abstraction: a collection of resources not (yet) associated with a VM. Both VMs and resource pools are described by a *resource specification* (RSpec).

Each NM-supported object—VMs and resource pools—can be referenced with a *resource capability* (rcap). An rcap is a 128-bit random string, where knowledge of an rcap corresponding to a particular object (RSpec) lets the bearer perform any operation supported by the object. The node manager is responsible for maintaining a persistent table of rcap/RSpec bindings.

In way of overview, the NM interface consists of five operations for creating and manipulating resource pools and virtual machines:

```
rcap = CreatePool(rspec, slice_name)
rcap[ ] = GetRcap( )
rspec = GetRspec(rcap)
rcap = SplitPool(rcap, rspec)
Bind(rcap, slice_name)
```

A root resource pool is created on behalf of some slice_name using the CreatePool operation. This operation can only be invoked by the privileged owner-VM (site_admin) on the node, and is generally used when the node is initialized.

Typically, the owner creates a resource pool for each trusted infrastructure service, and creates additional resource pools on behalf of any particular service that the owner wants to run on the node.

At some future time, the slice named in the CreatePool operation retrieves the set of rcaps for pools allocated to it by calling the GetRcap operation. Once a slice has retrieved the rcap for a pool of resources, it can learn the RSpec associated with the pool by calling the GetRspec operation.

A slice that possesses an rcap for a pool can create a new 'sub-pool' by calling SplitPool. This operation takes an rspec for the new sub-pool as an argument, and returns an rcap for the new pool. The RSpec for the original pool is reduced accordingly (i.e., calling GetRspec on the original rcap returns an RSpec corresponding to the remainder of resources after the split).

A pool of resources is bound to a VM using the bind operation. If the VM does not already exist, this operation also creates the VM. If the VM does exist, the resources represented by the rcap are added to those currently bound to it.

4.4 Slice

A *slice* is a set of VMs, with each element of the set running on a unique node. The individual VMs that make up a slice contain no information about the other VMs in the set, except as managed by the service running in the slice. As described in Section 4.2, each VM is initialized with a set of keys that allow the users that created to slice to remotely log into it, and with a user-provided bootscript that runs when the VM starts. Otherwise, all slice-wide state is maintained by the *slice authority* that created the slice on the users' behalf, as described in Section 4.7.

Slices are uniquely identified by name. Slice names are hierarchical, with each level of the hierarchy denoting the slice authority that is responsible for the behavior of all slices lower in the hierarchy. For example, plc.princeton.codeen names a slice created by the PLC slice authority, which has delegated to Princeton the right to approve slices for individual projects (services), such as CoDeeN. PLC defines a set of expectations for all slices it approves, and directly or indirectly vets the users assigned to those slices. Owners trust PLC to ensure that these expectations are met, as outlined in Section 3.

> **Implementation Note:** This description of slice names is clearly different from the current implementation, which does not explicitly include the plc prefix, and uses '_' as a deliminator instead of '.'. We expect the implementation to change to match this specification in the near future.

Although not currently the case, it is possible that PLC delegates the ability to approve slices to a regional authority, resulting for example in slice names like

plc.japan.utokyo.ubiq

and

plc.eu.epfl.fabius.

Note that while plc is currently the only top-level authority recognized on PlanetLab, this need not always be the case. There are at least two possibilities. One is that alternative "global" slice authorities emerge, such as an authority that authorizes commercial (for-profit) slices (e.g., com.startup.voip). Another possibility is that individual sites want to authorize slices that are recognized only within that site, that is, on a "private PlanetLab" (e.g., epfl.chawla).

Clearly, there needs to be a single global naming authority that ensures all top-level slice authority names are unique. Today, PLC plays that role.

Slice names serve as unique identifiers for slices, but there is no single name resolution service. Instead, the way in which slice names are interpreted depends on the context in which they are used. For example,

- when presented to the remote login service running on a node, the user gains access to the corresponding VM;
- when presented to a slice authority (e.g., PLC) to contact the users responsible for a slice, the authority returns email addresses for that set of users; and
- when presented to the NM as part of the RSpec in a CreatePool call, the name identifies a slice authority that the owner is allowing to create slices on its node.
- when presented to the node manger upon node startup, the NM returns an rcap for the resource pool bound to the named slice.

Note that all of the four examples above take fully specified slice names, while the third example allows the caller to name either a slice (thereby assigning resources to a specific slice) or a slice authority (thereby delegating to an authority like plc or plc.japan the right to create slices). Specific operations on slice names are presented throughout this document, as well as in the associated interface specifications.

> **Implementation Note:** We need to reconcile the fact that Unix logins cannot contain periods and must be no longer than 32 characters.

4.5 Slice Creation Service

A *slice creation service* (SCS) is an infrastructure service running on each node. It is typically responsible, on behalf of PLC, for creation of the local instantiation of a slice, which it accomplishes by calling the local NM to create a VM on the node. It also installs the keys and bootscript associated with the slice in the corresponding VM, thereby giving the the users associated with the slice access to the VM.

There may be more than one SCS running on a given node, but for the purpose of this discussion, we describe the default SCS, known as plc.scs.[2] Upon startup, plc.scs invokes two operations on the NM:

```
rcap[ ] = GetRcap( )
for each rcap[i]
   rspec[i] = GetRspec(rcap[i])
```

to retrieve the RSpec for each resource pool that the node owner has allocated to plc.scs. Each such *slice pool* RSpec identifies a slice authority server and contains an SSL certificate that plc.scs uses to create a secure connection to the SA server. The SCS then uses the service authentication protocol (see Section 5.5) to identify itself as running on a particular node, and subsequently may perform a limited set of operations using the SA interface to determine the set of slices that should be created on the node.

Users may also contact the SCS directly if they wish to synchronously create a slice on a particular node. To do so the user presents a cryptographically-signed *ticket*, which represents the ability to create a specific slice; these tickets are essentially RSpec's that have been signed by some slice authority and can be verified using the same certificate associated with the slice pool. We return to the issue of how slice authorities and plc.scs cooperate to create slices in Section 4.7.

4.6 Auditing Service

PLC audits the behavior of slices, and to aid in this process, each node runs an *auditing service* (AS). The auditing service records information about packets transmitted from the node, and is responsible for mapping network activity to the slice that generates it. Looking at the expectations in more detail, the node owner trusts PLC to (1) ensure that only the authorized users associated with a slice can access the corresponding VMs, (2) audit each VM's network activity, (3) map a VM to a slice name, and (4) map a slice name into the set of responsible users. Ensuring each of these expectations hold, it is possible to provide the owner with a

[2]This slice is actually called pl_conf today, but we adopt the proposed naming scheme for this slice.

trustworthy audit chain:

packet signature ⟶ slice name ⟶ users

where a packet's signature consists of a source address, a destination address, and a time. This is the essential requirement for preserving the chain of responsibility.

The auditing service offers a public, web-based interface on each node, through which anyone that has received unwanted network traffic from the node can determine the responsible users. Today, PlanetLab exports an SQL-based interface via port 80. PLC archives this auditing information by periodically downloading an entire MySQL table from each node. The architecture is neutral on the exact way in which the auditing service is queried.

4.7 Slice Authority

PLC, acting as a *slice authority* (SA), maintains state for the set of system-wide slices for which it is responsible. There may be multiple slice authorities but this section focuses on the one managed by PLC. For the purpose of this discussion, we use the term 'slice authority' to refer to both the principal and the server that implements it.

The PLC slice authority includes a database that records the persistent state of each registered slice, including information about every principal that has access to the slice. For example, the current PLC implementation includes:

principal = (name, email, org, addr, key, role)
org = (name, addr, admin)
slice = (state, rspec)

where

role = (admin | user)
state = ((delegated | central) & (start | stop) & (active | deleted))

The admin field of each org tuple is a link to a principal with role = admin (this corresponds to the person responsible for all users at the organization) and the user array in the slice tuple is a set of links to principals with role = user (these are the people allowed to access the slice).

> **Evolution Note:** The architecture is currently under-specified with respect to roles. The current implementation includes the roles outlined above, and while we recognize that certain set of standard roles will be necessary to support interoperability, we have not yet defined precisely what they will be (and how they parameterize the operations

given later in this section). However, we expect roles will remain an extensible part of the architecture.

The SA provides an interface by which users register information about themselves, create slices, bind users to slices, and request that the slice be instantiated on a set of nodes. This interface is described in a companion document, although we introduce some of the key operations below.

The current implementation involves a two-level validation process for principals: PLC first enters into an off-line agreement with an organization, which identifies an administrative authority that is responsible for approving additional users at that organization. The SA then lets this admin approve a user at the organization, create slices (this results in a slice record in the database), and associate users with slices. Those users are then free to define the slice's RSpec, associate the slice with a list of users (references to principals), and request that the slice be instantiated on a set of nodes.

> **Evolution Note:** Note that PLC's slice authority implements two levels of naming authorities in a single database: the PLC authority and a set of organization-level authorities. These levels would seem to lend themselves to a distributed implementation.

The RSpec maintained by the PLC slice authority is a superset of the core RSpec defined for use by the node manager. This lets the slice authority use the same structure to record additional information about the slice, and pass this information to the slice creation service running on each node. For example, the RSpec in the PLC slice database includes the slice name, the set of users associated with the slice, and the set of on which nodes the slice is to be instantiated. It also includes a database identifier that can be used to determine when a slice is deleted and recreated.

> **Implementation Note:** There is an open question of who is allowed to modify certain attributes in a slice RSpec: a user assigned to the slice, a administrator of the organization that approved the slice, or some operator working on behalf of PLC. Today, for each RSpec attribute, the SA database maintains a record of the privilege level required to change the associated value; some can be changed by the user and some by PLC operator.

For each slice, the SA maintains a state field that indicates several key properties of the slice. The state is a combination of several independent values, each recording an aspect of the slice's current mode of use. The first bit indicates

whether the creation of VMs for the slice has been delegated to another service, or should be performed by the slice authority's agent on each node (corresponding to value central). The second bit indicates whether the slice is currently started or stopped—a slice may be stopped by either associated users or by PLC, for example, it the slice behaves in a manner that has adverse effects. The third bit indicates whether a slice is active or has been deleted; for auditing purposes we retain all slice records in the database even after they have been deleted by their users. We explain the two ways of instantiating a slice in the next subsection.

There are two ways to instantiate a slice on a set of nodes: *directly* or via *delegation*. Before either can be invoked, however, a record for the slice must exist in the SA database, and an RSpec must be initialized for the slice. This involves multiple calls on the SA:

```
CreateSlice(auth, slice_name)
SetSliceAttr(auth, slice_name, attribute)
AddSlicePrincipal(auth, slice_name, principals[ ])
AddSliceNode(auth, slice_name, nodes[ ])
```

The first creates the slice in the SA database, multiple calls to the second fills out the RSpec for the slice, and the third and fourth operations associate a set of users and nodes with the slice, respectively. For all four operations (as well as those that follow), auth contains authorization information for the call. (See Section 5 for a description of the underlying security architecture.)

One a complete slice record exists in the SA database, the user can enable slice instantiation by invoking:

```
StartSlice(auth, slice_name)
```

This operation sets the slice's state = central in the database, which means that this slice will be included in the set of slices to be created on the appropriate nodes when the SA is queried by the slice creation service on one of those nodes. As described in Section 4.5, the slice creation service running on any nodes that trust this particular slice authority calls the SA periodically, and creates the corresponding set of slices if they do not already exist on the node.

In the the second case, the user acquires a ticket from the slice authority via a

```
ticket = GetSliceTicket(auth, slice_name)
```

operation. The user is then responsible for contacting the slice creation service running on individual nodes to redeem this ticket. This might be done indirectly through a third party service. A ticket is an RSpec for the slice that has been

signed by the slice authority using the private key that corresponds to the public key included in the SA server's SSL certificate.

> **Evolution Note:** By supporting both the GetSliceTicket and StartSlice operations, we have allowed PlanetLab to evolve in two different directions. In one, slice authorities return tickets but are never asked to directly create slices, thereby serving as a pure 'naming authority' for slices. In the other, a slice authority bundles the role of naming slices with the role of creating slices. The current implementation supports both, and while today most slices are created directly by PLC, third-party slice creation services (e.g., that provided by Emulab []) are starting to emerge.

Finally, each slice authority communicates an SSL certificate out-of-band to node owners that are willing to host slices it authorizes. Each node owner includes this certificate in the RSpec for the resource pool allocated to the slice authority. Slice creation services, in turn, use this certificate to securely query the slice authority for the set of slices to be instantiated on the node that the service is running on.

4.8 Management Authority

PLC, acting as a *management authority* (MA), maintains a server that installs and updates the software (e.g., VMM, NM, SCS) running on the nodes it manages. It also monitors these nodes for correct behavior, and takes appropriate action when anomalies and failures are detected. As with a slice authority, we use the term 'management authority' to refer to both the principal and the corresponding server.

The MA maintains a database of registered nodes. Each node is affiliated with an organization (owner) and is located at a site belonging to the organization. The current MA implementation includes a database with the following tuples:

```
principal = (name, email, org, addr, keys, role)
org = (name, address, admin, sites[ ])
site = (name, tech, subnets, lat_long, nodes[ ])
node = (ipaddr, state, nodekey, nodeid)
```

where

```
state = (install | boot | debug)
role = (admin | tech)
```

Similar to the SA database, the admin field of each org tuple is a link to a principal with role = admin; this corresponds to the primary administrative contact for the

organization. Similarly, the tech field in the site tuple is a link to a principal with role = tech; this is the person that is allowed to define the node-specific configuration information used by the node's slice creation service when the node boots.

> **Implementation Note:** The current implementation uses a single database for both the SA and MA. The only tuple they have in common is the set of principals, and in fact, these are shared, with the role field given by the union of the two role sets. Since organizations that join PlanetLab are both owners and users, the same person is typically the admin in practice, and so there has been no reason to distinguish between these two cases.
>
> Note that all PlanetLab documentation apart from this report refers to the admin role as the *principal investigator* (PI), but we elect to use the former as it is a more generic term for the corresponding principal.

The node state indicates whether the node should (re)install the next time it boots, boot the standard version of the system, or come up in a safe (debug) mode that lets PLC inspect the node without allowing any slices to be instantiated or any network traffic to be generated. The MA inspects this field to determine what action to take when a node contacts it. Nodes are initially marked (in the MA database) as being in the install state.

4.8.1 Public Interface

The management authority supports two interfaces. The first—which is the public interface to the MA—is used by node owners to register their nodes with PLC. An organization (node owner) enters into a management agreement with PlanetLab through an offline process, during which time PlanetLab learns and verifies the identities of the principals associated with the organization: its administrative and technical contacts. These principals are then allowed to use this interface to upload their public keys into the MA database, and create database entries for their nodes. These operations include:

```
node_id = AddNode(auth, node_values)
UpdateNode(auth, node_id, node_values)
DeleteNode(auth, node_id)
```

where node_id is an MA-specific unique identifier for the node and node_values is a structure that includes the node's name and IP address.

This public interface also supports operations that allows users (and slice authorities) to learn about the set of nodes it manages, so that they know the set of nodes available to deploy slices on:

```
node_ids[ ] = GetManagedNode(auth)
node_values[ ] = GetNodes(auth, node_id)
```

Evolution Note: Today, this call returns the DNS name and IP address of a set of nodes managed by PLC. At some point, it makes sense that this file publish a broader set of attributes for available nodes. These attributes would most naturally be represented as an RSpec.

4.8.2 Boot Manager Interface

Nodes use the MA's second interface to contact PLC when they boot. Specifically, the bootfile available on each node (see Section 4.1) contains a minimal Linux system that initializes the node's hardware, reads the node's network configuration information from plnode.txt, and contacts PLC. The MA returns an executable program, called the *boot manager* (approximately 20KB of code), which the node immediately invokes.

The boot manager (running on the node) reads the nodekey from plnode.txt, and uses HMAC [5] to authenticate itself to the MA with this key. Each call to the MA is independently authenticated via HMAC. The MA also makes sure the source address corresponds to the one registered for the node, to ensure that the right plnode.txt has been put in the right machine, but this is only a sanity check, as the server trusts that the node is physically secure.

The first thing the node learns from the MA is its current state. If the state = install, the boot manager runs an installer program that wipes the disk and downloads the latest VMM, NM, and other required packages from the MA, and chain-boots the new kernel. The downloaded packages are also cached to the local disk. A newly installed node changes the node's state at the database to boot so that subsequent attempts do not result in a complete re-install.

If the node is in boot state, the boot manager generates a public-private key pair to be used in authenticating service slices once the node has booted (see Section 5.5), and requests that the MA generate a public key certificate that binds the public key to this particular node. It then contacts the MA to verify whether its cached software packages are up-to-date, downloads and upgrades any out-of-date packages, and finally chain-boots the most current kernel. Otherwise, if the boot manager learns that the node is in debug state, it continues to run the Linux kernel it had booted from the bootfile, which lets PLC operators ssh into and inspect the

node. Both operators at PLC and the site technical contacts may set the node's state (in the MA database) to debug or install, as necessary.

In addition to boot-time, there are two other situations in which the node and MA synchronize. First, running nodes periodically contact the MA to see if they need to update their software, as well as to learn if they need to reinstall. Each node currently does this once a day. Second, whenever the node state is set to debug in the MA database, the MA contacts the node to trigger the boot process, making it possible to bring the node into a safe state very quickly.

4.9 Owner Script

The PlanetLab architecture provides owners with as much autonomy as possible, even to those that offload responsibility for managing the node to PLC. Therefore, owners need some way to communicate how they want their nodes managed. For example, owners might want to prescribe what set of services (slices) run on their nodes, and allocate resources to one or more slice authorities for redistribution to slices.

As described in Section 4.3, a designated VM named site_admin is created on each node, and initialized with the keys belonging to the site's technical and administrative contacts. An *owner script* running in this VM calls the node manager to express its preferences.

> **Implementation Note:** Today, owners confer their wishes out-of-band to PLC, which creates and installs an appropriate owner script on each node. Whether an owner script is edited remotely at PLC and later copied to each node, or the owner logs into its own node and directly edits the ownerscript is an implementation detail.

The owner script typically allocates a set of resources to both slice authorities and to individual slices. It does this using the CreatePool( ) operation. The slice name given as an argument to CreatePool( ) identifies the slice that is allowed to retrieve the rcap and RSpec for the pool using the GetRcap( ) and GetRspec( ) calls, respectively. When allocating capacity to a slice creation service like plc.scs, the RSpec includes three pieces of information: (1) specifications for the resources in the pool; (2) the name of the slice authority that is allowed to create slices using the pool (e.g., plc or plc.japan); and (3) the SSL certificate for the corresponding slice authority.

4.10 Resource Specification

A *resource specification* (RSpec) is an abstract object used throughout PlanetLab. It is the main object maintained by a node manager and slice creation service (representing both virtual machines and resource pools), and it is the primary component of the record maintained for each slice by a slice authority (representing the attributes of a PlanetLab-wide slice). As such, many of the methods supported by these components operate on RSpecs. Similarly, a ticket is an RSpec signed by a slice authority.

An RSpec can be viewed as a set of attributes, or name/value pairs. For example, the following set of attributes correspond to the default resources allocated to a VM in the current system (the parenthetic comment gives the corresponding unit):

cpu_share = 1
disk_quota = 5 (GB)
mem_limit = 256 (MB)
base_rate = 1 (Kbps)
burst_rate = 100 (Mbps)
sustained_rate = 1.5 (Mbps)

Despite this example, an RSpec is actually not a flat list of attributes; it has some structure. More accurately, an RSpec can be viewed as the base class of an object hierarchy, as shown in Figure 3. Class PoolRSpec represents a resource pool, class SliceRSpec represents a PlanetLab-wide slice, and class VServerSlice represents a vserver-based VM.

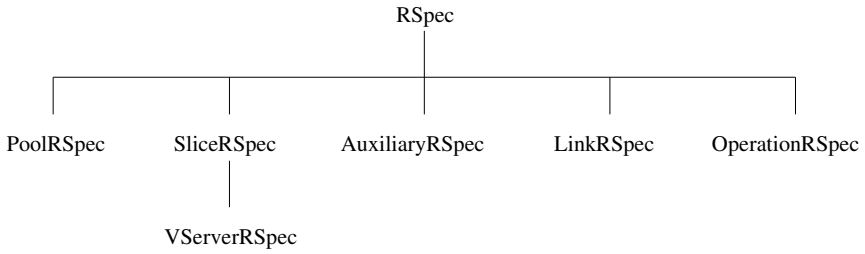

Figure 3: RSpec object hierarchy.

LinkRSpec and OperationRSpec are future additions to the object hierarchy and thus not yet defined. A LinkRSpec represents a reference to another RSpec, possibly with reduced capabilities, that can be created by any client as a means of passing a reference to an RSpec that does not convey the full capabilities of the

RSpec itself. An OperationRSpec is a functional object that permits a client to execute a privileged operation, such as one supported by Proper [7].

Note that an implementation may define other classes within this hierarchy as long as those classes do not affect the behavior of the named classes. For example, the current implementation defines classes ParentRSpec and PrimaryRSpec as base classes for PoolRSpec and SliceRSpec respectively, and a class VmmBase as an intermediate class between SliceRSpec and VServerSlice.

Because RSpecs are commonly passed over the network, for example between a slice authority and a slice creation service, it is necessary to define an external representation. RSpecs are currently represented in XML.

5 Security Architecture

This section summarizes the security architecture supported by the various components just described. It describes how nodes security boot, how users create and access slices, and how services authenticate themselves to each other and to PLC.

5.1 Preliminaries

We begin by outlining the underlying mechanisms, formats, and assumptions of the security architecture.

5.1.1 Static Key Infrastructure

A small number of public key pairs and certificates must be created and distributed among various infrastructure components to establish the basic authentication framework. Two requirements must be satisfied:

- There must be a well-known set of public keys, which we refer to as *root keys*, that can be used to authenticate top-level certificates. This set can include well-known keys for commercial certificate authorities (e.g., VeriSign, Thawte) or well-known keys specific to PlanetLab (i.e., either the global PlanetLab or private instances).

- Every authority—management authority (MA) or slice authority (SA)—that provides an interface for access by services (e.g., management services running in their own slice) must have a public-private key pair, and a certificate signed by one of the certificate authorities (CA) that can be authenticated using the corresponding well-known key.

In certain small-scale environments (e.g., testing and development), it may instead be desirable to add the public keys for every authority to the set of well-known keys, thus eliminating one level of signed certificates. However, this sacrifices some of the benefits of using a root key to sign the authority's certificate, such as the ability to change a particular authority's certificate without updating the global list of root keys.

5.1.2 Certificates

The certificate format is a simple XML document as defined by the XML-SEC standard. The document contains three elements:

subject: Identifies the certificate subject and includes a PEM representation of the subject's public key. The certificate indicates to the recipient that the binding between subject identity and public key was verified by the certificate issuer using out-of-band methods (e.g., some feature of the underlying OS).

issuer: Data used to authenticate the certificate, either an identifier for a well-known public key—indicating that this certificate is a top-level certificate (i.e., was signed by with a root key)—or a certificate that includes the public key for the authority that signed this certificate.

signature: An element in XML-SEC format that specifies a number of signature parameters and includes the cryptographic signature value for this certificate. An authority uses information in the authenticator element—either a public key contained in an embedded cert or a named well-known key—to verify this signature.

Note that because the issuer is verified by either a well-known public key or another certificate, certificates are a recursive data structure.

> **Implementation Note:** The current implementation uses a simple Python script, `mkcert`, to generate or verify XML certificates, hiding the details of recursive certificates from the end-user. The actual signing and verification are performed using the `xmlsec1` utility, which in turn uses the *OpenSSL* cryptographic primitives.
>
> **Evolution Note:** We may want to generalize the architecture to support multiple certificate formats rather than require XML-SEC. For example, the x509 standard for digital certificates provides roughly the same capabilities and could be used as an alternative.

5.2 Booting a Node

Each MA runs a boot server that is contacted by its client nodes whenever they reboot. Each node boots from an immutable filesystem (e.g., a CD-ROM) that bootstraps a boot manager onto the node. The boot manager, in turn, contacts the MA's boot server to download all necessary code and configuration information needed to fully boot the node, as described below. The immutable filesystem also includes a set of certificate authority (CA) certificates corresponding to the well-known root public keys. This set of CA certificates is used by the boot manager to authenticate the boot server. Note that the bootstraping code and certificates can be openly distributed on a CD, for example, and used to boot any node. The information is MA-specific, but not node-specific.

When a node is registered with the MA, a secret key is generated for that node, stored in the MA database, then exported to the node as part of the node configuration file (see Section 4.1). The node key can subsequently be used as a shared secret by the boot manager to authenticate the node to the MA. The node owner must copy the configuratoin file onto a write-protected device, typically a device with mechanical write protection capability. We assume that the node owner physically secures the node so that the write-protected device cannot be accessed by unauthorized users.

When a node boots, the boot manager contacts the MA securely (using SSL) to load all necessary code and current configuration information. The node uses the set of CA certificates to authenticate the MA's boot server, and then uses its secret node key to authenticate itself to the boot server via HMAC. The authenticated node then queries the boot server to determine whether the node should boot into its standard execution mode or a restricted "debug" mode that allows only MA administrators to access the machine. This process is described in more detail in Section 4.8.2.

5.3 Creating a Slice

Slice creation is a multi-stage process, involving the node owner, a slice creation service, and a slice authority. The process includes the following steps:

- The node owner acquires an SSL certificate for the PLC slice authority and/or any other trusted slice authorities through an off-line process. For each slice authority, the owner creates a configuration file that identifies a server for the slice authority and includes the SSL certificate for that authority. The owner script that is run by the node manager whenever a node boots uses these configuration files to create a number of slice pool RSpecs, and also identifies the slice creation service (e.g., plc.scs) entrusted to create

slices on the node (see Section 4.5). Only a single privileged VM belonging to the owner is allowed to initialized a slice creation service in this way.

- When the slice creation service restarts, it determines the set of slice pools created by the node owner and associated with that service. It also creates a public-private key pair and has the node manager sign a certificate binding the public key to the service slice. For each pool, the slice creation service uses the associated SSL certificate to authenticate the corresponding slice authority, authenticates itself to the slice authority using the certificate obtained from the node manager, uses the slice authority's API to determine the set of slices to be created on that node, and finally creates a VM for each slice associated with the node.

- A slice authority may also use the private key corresponding to the public key in its SSL certificate to sign tickets, which grant to the bearer the right to create a specific slice (and the right to allocate resources to a slice). A user may present such a signed ticket to the slice creation service running on a node, which uses the public key contained in the authority's SSL certificate to authenticate the ticket. This allows users to avoid waiting for the periodic updates performed by the slice creation service, but instead create their own slices within the resource allocation constraints imposed by the slice authority.

5.4 Enabling a User

PLC authorizes a site, and its PI, through an offline process. A PLC account is then created for the PI, corresponding to both the PLC slice and management authorities. The PI uses a secure connection to PLC to upload a public `ssh` key, which is then installed in the owner VM on the site's nodes.

Users at the site create a PLC account using the same process, but they are approved by the site's PI. Users then upload their public `ssh` keys. When a user is associated with a slice, the user's `ssh` key is then installed in the VM on each node that is part of the slice. Users subsequently may access their set of VMs via `ssh`.

5.5 Certificate-Based Authentication and Authorization

One challenge that must be addressed in the PlanetLab architecture is how services can perform certain operations, such as invoking various PLC operations, in a secure manner. (Recall that the SA and MA operations in Sections 4.7 and 4.8 take an auth value—a session key—as their first parameter). We do not want to require that each service distribute either a shared secret or a private key to every node that

it executes upon; instead, the infrastructure provides services with a way of leveraging the trusted nature of the node manager and the VMM property that particular types of requests can be associated to the originating slice (with no possibility of malicious spoofing or accidental misattribution).

To address this problem, we adopt the same type of certificate-based authentication scheme as used in systems such as Taos [] and Terra []. Services create a public-private key pair, then request that the node manager sign a certificate binding the public key to the service's slice. The NM uses the low-level VMM interface to identify the slice originating a particular request. The NM then creates a certificate that binds the public key to the slice, and signs that certificate using its own public key. It also includes in the certificate a similar certificate that binds the NM public key to node manager itself, this certificate being signed by the node's MA at boot time, as part of the boot protocol.

When a service wishes to invoke an operation on either the MA or SA, it must first authenticate itself to the authority and get a session key that it passes to subsequent calls. This authentication is performed by calling a

```
session_key = AuthenticateCertificate(cert)
```

operation, with the service's certificate as a parameter. The authority verifies the certificate (i.e., checks that the sequence of signatures in the certificate can be traced back to a well-known CA), creates a session key appropriate for the level of access the caller is to be granted to that particular authority, and uses the public key in the certificate to encrypt the session key. Only the corresponding private key, held by the caller, can be used to decrypt the session key.

Certificates created in this way can also be used in any scenario that requires authentication of the caller's identity, not just interaction between a service and an authority. For example, one service might use such a certificate to identify itself to a resource broker so that the broker can determine which service to charge for resource usage. Note that although this description assumes that certificates are used to verify the identity of a particular service, it could readily be applied to certificates that instead specify particular capabilities be granted to the certificate bearer.

6 Interfaces

This section enumerates PlanetLab's externally visible interfaces and data formats (interface specifications) and its internal (private) interfaces, giving pointers to the corresponding specification or reference implementation, respectively.

Node Manager: Provides an interface used by infrastructure services to create and manipulate virtual machines and resource pools. This interface can only be invoked locally. Most operations take an RSpec as an argument (external interface specified in [6]).

Slice Creation Service: Provides a remotely accessible interface invoked by slice authorities and users to create slices (external interface specified in [6]).

Slice Authority: Provides an interface that is used by research organizations to register their users, by users to their create slices, and by management authorities (and other third parties) to learn the set of users associated with a given slice name (external interface specified in [4]).

Management Authority: Provides two interfaces:

- a public interface used by research organizations to register their nodes with a management authority and both users and slice authorities to learn the set of nodes managed by the authority (external interface specified in [4]); and
- a private interface used by nodes to download and install PlanetLab software and node configuration files (reference implemenation described in [3]).

PlanetFlow Archiving Service: Provides a per-node interface that users can query to learn what slice is responsible for a given packet flow, an interface by which a node uploads its flow information to an archive, and an archive-wide interface that users can query to learn what slice is responsible for a given packet flow (reference implementation described in [2]).

References

[1] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid. *Int'l. Journal Supercomputer Applications*, 15(3), 2001.

[2] M. Huang. PlanetFlow and Audit Archive: Reference Implementation. Technical Report 06-034, PlanetLab, May 2006.

[3] A. Klingaman. Securely Booting PlanetLab Nodes: Reference Implementation. Technical Report 05-026, PlanetLab, Jan. 2005.

[4] A. Klingaman. Slice and Management Authorities: Interface Specifications. Technical Report 06-032, PlanetLab, May 2006.

[5] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication; RFC 2104. *Internet Req. for Cmts.*, Feb. 1997.

[6] S. Muir. Node Manager and Slice Creation Service: Interface Specifications. Technical Report 06-033, PlanetLab, May 2006.

[7] S. Muir, L. Peterson, M. Fiuczynski, J. Cappos, and J. Hartman. Proper: Privileged Operations in a Virtualised System Environment. In *Proc. USENIX '05*, pages 367–370, Anaheim, CA, Apr. 2005.

[8] L. Peterson, A. Bavier, M. Fiuczynski, S. Muir, and T. Roscoe. Towards a Comprehensive PlanetLab Architecture. Technical Report 05-030, PlanetLab, June 2005.

[9] L. Peterson and T. Roscoe. The Design Principles of PlanetLab. *Operating Systems Review (OSR)*, 40(1):11–16, Jan. 2006.