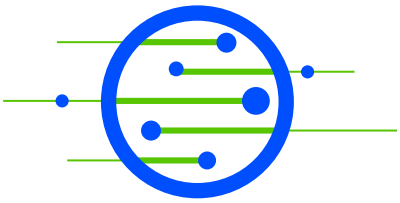

Towards a Comprehensive PlanetLab Architecture

Larry Peterson, Andy Bavier, Marc Fiuczynski, Steve Muir
Princeton University

Timothy Roscoe
Intel Research – Berkeley

PDN–05–030
June 2005

Status: Ongoing Draft.

Towards a Comprehensive PlanetLab Architecture

Larry Peterson, Andy Bavier, Marc Fiuczynski,
Steve Muir, and Timothy Roscoe

June 7, 2005

1 Introduction

PlanetLab has evolved rapidly over the past two years according to a set of design principles [12], but with minimal attention paid to distinguishing between its underlying architecture and its current implementation. This documents attempts to address this shortcoming by defining the PlanetLab architecture (as of Version 3), and in the process, identifying various implementation artifacts.

This report comes with three caveats. First, we make no claim that the Version 3 architecture is the final word; this document should be interpreted as a starting point for a general discussion of what the PlanetLab architecture should be. Second, this document is not intended as a user reference guide; it describes PlanetLab from a management perspective rather than the user's perspective. Third, the interfaces presented in this report are simplified abstractions of the actual interfaces; the syntax of actual calls are documented on the PlanetLab web site.

The central goal of PlanetLab is to support *distributed virtualization*—allocating a widely distributed set of virtual machines to a user or application, with the goal of supporting broad-coverage services that benefit from having multiple points-of-presence on the network. This is exactly the purpose of the PlanetLab *slice* abstraction [3]. The central challenge of PlanetLab is to provide decentralized control of distributed virtualization.

PlanetLab's model for decentralized management is, in turn, guided by two design requirements. The first is that management of distributed virtualization should be decomposed into a large number of separate functions: discovering resources, creating slices on a set of nodes, buying and selling node resources, keeping the code running in a slice up-to-date, monitoring a slice's behavior, and so on. To preserve generality, PlanetLab attempts to keep these functions separate. Consequently there is no single "management entity" as such, even though particular

systems may collapse many of these functions into one component. We refer to this decoupling as *unbundled management*.

The second requirement is that systems must 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 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 Base Elements

This section outlines the base architectural elements of PlanetLab. For each, we give examples of how the element might be implemented, including how the element is implemented in the current version, as well as other possible realizations.

A *node* is a machine capable of hosting one or more virtual machines. There is currently a one-to-one mapping between nodes and physical machines, but a node might be implemented by a cluster of machines, where the node manager running on each node is responsible for instantiating virtual machines on some processor in the cluster (and possibly migrating the virtual machine from processor to processor over time). Nodes also need not be of the same machine architecture, although the current implementation is limited to x86 processors. In fact, a single node might include a heterogeneous collection of processing elements, such as a general-purpose processor with one or more network processors.

A *virtual machine* (VM) is an execution environment in which a slice runs on a particular node. VMs are implemented by a *virtual machine monitor* (VMM) running on the node. It is expected that the VMM provides some level of isolation between the VMM's it hosts. Each VM is specified by a set of *attributes*, called a *resource specification* (*rspec*), that defines how much of the node's resources are allocated to the VM. The *rspec* also specifies the VM's *type*. PlanetLab currently supports a single Linux-based VMM, and so defines a single VM type (linux-vserver), but other types are possible (e.g., xen-domain). A given node might support more than one VM type.

A *node manager* (NM) is a program running on each node that creates virtual machines on the node, and controls the resources allocated to those VMs. All operations that manipulate virtual machines on a node are made through the node manager; the native VMM interface is not called directly. There is a one-to-one mapping between nodes and node managers.

A *slice* is a set of virtual machines, with each element of the set running on

a unique node. Each slice runs a network *service*. The specification for a slice is given by a slice-wide *rspec*. A slice is bound to a set of principals that are responsible for developing the service that runs in the slice. Today all VMs belonging to a slice must be of a single type. This is because PlanetLab has only one type, of course, but also because an rspec is defined on a per-slice basis rather than on a per-VM basis. That is, a single rspec is used for all VMs belonging to a slice. This obviously leads to a simpler implementation, but is not strictly necessary.

An *infrastructure service* is a "helper" service used by other slices (services). For example, an infrastructure service might create slices on a set of nodes; buy and sell node resources; keep the code running in a slice up-to-date; monitor a slice's behavior, and so on. Multiple competing infrastructure services are possible, and in fact, encouraged by PlanetLab's principle of unbundled management.

3 Principals and Relationships

This section identifies the key principals (and the relationships among them) in the PlanetLab architecture. There are four principals of note: *owners* host one or more nodes; *service providers* implement and deploy network services on a set of nodes; *management authorities* operate a set of nodes on behalf of one or more owners; and *slice authorities* register a set of service providers. These principals have the following responsibilities:

- A management authority is responsible for installing and maintaining the software (e.g., VMM and NM) that runs on the nodes it manages. Through this software, the management authority creates VMs on nodes, monitors these nodes for correct behavior, and takes appropriate action when anomalies and failures are detected. A management authority may also be the owner of some fraction of the nodes it manages.

- A slice authority is responsible for registering providers, creating (naming) slices, and binding a set of providers to each slice. It must be able to map a slice to the providers that are responsible for its behavior. A slice authority also registers credentials for providers; for example, it may collect public keys and distribute them to the nodes that host that slice.

- Each owner retains ultimate control over their own nodes, but selects a single management authority to manage its nodes, and approves one or more slice authorities on whose behalf it is willing to accept slices.

- Each provider implements and deploys a network service. The provider is responsible for the behavior of its service.

As with other aspects of the architecture, we note that some of these roles may be conflated in a particular implementation; e.g., a node owner may act as its own management authority.

To make this discussion more tangible, consider that PlanetLab currently consists of nodes owned by roughly 200 autonomous systems (spanning over 260 sites), and hosts slices affiliated with approximately 200 independent organizations (representing over 450 research projects, or budding service providers). Establishing 200×200 pairwise trust relationships is an unmanageable task: a researcher would have to obtain permission to create VMs on nodes owned by 200 organizations, while a university would need to approve requests for use of its nodes from 200 independent research organizations. (The set of organizations hosting nodes and the set of organizations acquiring slices are almost identical in PlanetLab, but this need not be the case in general.)

A key insight to reducing such an N×N problem into an order N problem is to use trusted intermediaries. The *PlanetLab Consortium* is one such trusted intermediary: node owners trust it to manage the behavior of VMs that run on their nodes, and service providers trust it to provide access to a set of nodes that are capable of hosting their services.[1] Thus, understanding the underlying trust assumptions is the key to defining the PlanetLab architecture.

The following describes the critical trust relationships among principals, and sketches the incentives each party has in not violating this trust. Figure 1 schematically depicts these trust relationships.

1. An owner trusts a management authority to install software that is able to map network activity to the responsible slice. This software must also isolate and bound/limit slice behavior. The owner will find a more reliable management authority if this trust is violated.

2. Each owner trusts certain slice authorities to reliably map slices to providers. An owner will configure its nodes to accept slices only from trusted slice authorities.

3. A provider trusts a slice authority to act as its agent, creating slices on its behalf and checking credentials so that only that provider can install and modify the software running in its slice.

4. A slice authority expresses trust in a provider by issuing it credentials that lets it access slices. This means that the provider must adequately convince

[1] We use "PlanetLab" as a shorthand for "PlanetLab Consortium", the intermediary, but the meaning should be clear from the context.

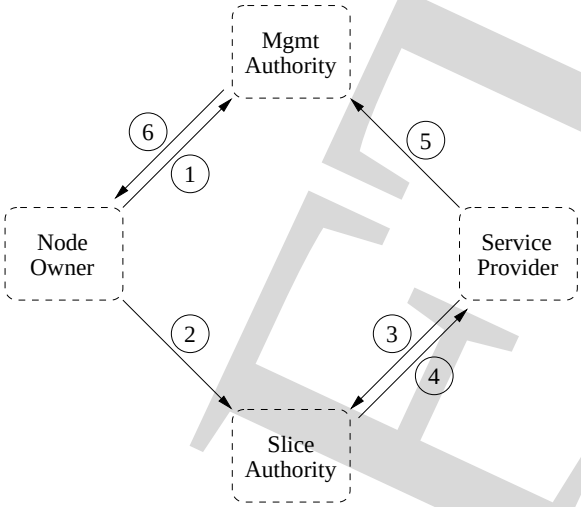

Figure 1: Critical trust relationships among principals.

the slice authority of its identity (e.g., affiliation with some organization or group).

5. A provider may trust only certain management authorities to provide it with working VMs, and to not falsely accuse them of out-of-bounds behavior. The provider will only run a service on those nodes with a trustworthy management authority.

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

The key feature of the trust relationships shown in Figure 1 is that there are *no* trust dependencies between owners and providers. This absence is what allows the system scale in the number of providers and owners it supports. A secondary feature is that, while they could be combined into a single entity, there is no inherent dependency between management and slice authorities. They are free to evolve independent of each other.

Architecturally, the requirement implied by these relationships is a name space for management and slice authorities, where each slice is uniquely identified by the pair:

⟨ slice_name, slice_authority_name ⟩

For simplicity, we identify a slice by its slice_name alone in the remainder of this document.

4 Architectural Components

This section identifies PlanetLab's central architectural components, with a particular focus on the role these components play in managing the trust relationships involved in distributed virtualization.

4.1 Create Slices

A slice authority typically provides an interface (front-end) through which service providers can request that slices be created on their behalf. A slice authority also needs a way to create VMs on individual nodes, which implies that it needs a point-of-presence on each node. We refer to this point-of-presence as a *slice creation service*, and it is an example of an infrastructure service.

Such a slice creation service runs on each node on which the slice authority is allowed to create slices. It invokes the node manager on each node to create a VM belonging to some slice and associates resources with that VM. It also installs credentials, keys or other state belonging to the responsible principals in those slices, thereby giving those principals access to the VMs for the purpose of installing code and starting processes. The slice creation service communicates with the slice authority over a private protocol that is not specified by the architecture.

4.2 Audit Service

Management authorities must audit the behavior of slices. For this purpose, each management authority runs an *auditing service* on each node it manages. 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 the management authority to (1) constrain a service to a VM, (2) audit the VM's network activity, and (3) map a VM to a slice name. The node owner separately trusts the slice authority to map a slice name into the set of responsible principals. Ensuring each of these expectations hold, it is possible to provide the owner with a trustworthy audit chain:

packet signature $\longrightarrow$ slice $\longrightarrow$ provider

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.

4.3 Create and Provision VMs

The management authority must honor requests by a slice creation service to create virtual machines, should the slice authority behind it have the authority to do so. The node manager provides an interface that is used to create VMs on local nodes and bind resources to them. Again, to cleanly separate functionality in the architecture, this interface is by definition invoked only from the local node; remote access is provided indirectly through one or more infrastructure services bootstrapped on the node. Note that the interface of the NM is one of the global "fixed points" of the architecture (along with a common name space for slices), in the sense that PlanetLab requires global agreement on this interface among any set of owners, providers, slice authorities, and management authorities that may want to interact on a set of nodes.

Since a node manager is responsible for associating resources with VMs, resource allocation concepts form major parts of its interface. We have already introduced the notion of an *rspec*, which describes a particular collection of resources. Precisely what a resource specification describes will evolve over time; we describe the current *rspec* used in PlanetLab in Section 5. In general, it consist of a set of (type, value) pairs. The NM manipulates the local resource allocation interface on behalf of individual VMs in order to implement *rspec*s.

The node manager also supports a *resource pool* abstraction. A pool is a collection of resources not (yet) associated with a VM. Any node must manage (via the node manager) its own pool of resources, and must be able to create new pools from this pool to allocate to slices, though we make no assumption here that resource specifications are additive or conserved. We do require that a node manager can provide a pool for any new slice. For generality, we assume that a node manager may create new resource pools from existing ones via a "split" operation. Access to pools is provided by a resource capability (*rcap*), which both names a resource pool and confers the right to use and/or split the pool. Concretely, the NM must be able to map an *rcap* to a pool and implement basic operations on pools for holders of valid *rcap*s.

We acknowledge that this trio of rspecs, pools, and rcaps seems somewhat arbitrary and abstract. However, we claim that almost any resource allocation scheme for distributed virtualization can be expressed in these terms, and furthermore, all three concepts are needed to capture the requirements of architectures for distributed virtualization. For example, decoupling pools and virtual machines lets each be created at a different time. A pool can be created as part of a "root alloca-

tion decision" made by the owner at system boot time, while a VM that is going to use that pool is created at a later time by a slice creation service. Similarly, a brokerage service that owns a pool can put the corresponding resources on a market, independent of a slice creation service that is used to create VMs and bind them to resources obtained on that market. Also notice that decoupling pools and VMs means that an existing slice–with its own set of resources bound to it–can own an rcap for a separate pool of resources.

Finally, one link in the chain of responsibility is that the node manager must preserve is being able to map a VM that performs some action to the slice it belongs to. In some systems, the node manager interface can define a level of abstraction that hides the implementation details of the VMM (for example, providing the same interface to a vserver-based VMM and a Xen-based VMM). In any case, the NM must ask the VMM to identify the VM that invoked an operation, and then (since the NM itself created the VM on behalf of a slice) it can map that VM to the calling slice. We use the notion of a *virtual machine identifier* for naming a VM on a particular node.

4.4 Owner Preferences

The PlanetLab architecture intends to provide the owner with as much autonomy as possible, while offloading responsibility for managing the node to a trusted management authority. Therefore, owners need some way of communicating how they want their nodes managed by the management authority. For example, owners might want to prescribe what set of services (slices) run on their nodes, including the slice creation service of whatever slice authorities the owner trusts. The owner might also want to assign resources from a pool to one or more services. This might be implemented with a configuration file, but since virtual machines play such a dominant role in PlanetLab, it is natural to have a privileged *owner-VM* on each node that is automatically started by the node manager each time the node boots. The owner could then specify an initialization script that runs in the owner-VM. This script would configure the node manager to reflect the owner's preferences for the node.

4.5 Putting it all Together

We now put all the pieces together, as schematically depicted in Figure 2. Each management authority maintains: (1) a NM and VMM to run on each client node; (2) an auditing service to run on each client node; and (3) a database of credentials for a set of client owners (and their nodes). It also runs a process to securely boot the NM, VMM, and "bootstrap" services on nodes belonging to owners with

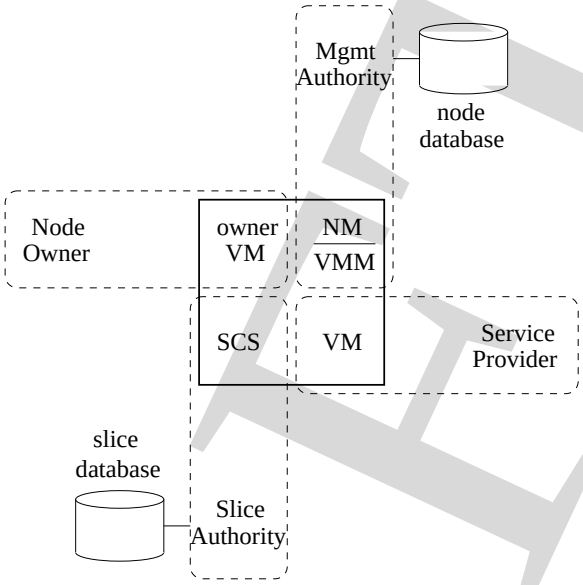

Figure 2: A node (in center) with the components run by each principal. SCS denotes a local point-of-presence of a slice creation service.

which it has a management agreement. It is able to use the auditing service to map network activity on the nodes it manages to the responsible slice.

Each slice authority maintains: (1) a database containing a set of slice-to-provider bindings; (2) a database with the identity of (keys for) each registered provider; and (3) a slice creation service that runs on each node. A slice authority needs to be able to map a slice name to a set of responsible providers. It also has to run a slice creation service on each node that might host one of the slices for which it is responsible.

Finally, each owner maintains: (1) the identity of (keys for) the management authority that it is responsible for its nodes; and (2) a set of "bootstrap" slices (and associated rspec) that are to be instantiated on the node. When a node boots, it contact its management authority, which in turn boots the NM and VMM. Once the system is running, the NM creates the owner-VM from which the owner invokes the NM to create an initial set of VMs, including the "bootstrap" slice creation services provided by the trusted slice authorities.

5 Implementation

This section outlines the data structures and interfaces used in PlanetLab to implement the components described in the previous section. The description purposely abstracts the implementation in two respects. First, the interfaces are not complete; see the online Guides at www.planet-lab.org/doc/ for the authoritative version of the various interfaces. Second, PlanetLab currently combines one management authority and one slice authority into a single front-end, corresponding to the GUI and programmatic interfaces available at www.planet-lab.org. However, these two components are separable, and will be described as such in this section.

5.1 Node Manager

The PlanetLab node manager implements a uniform interface through which all other components create and manipulate VMs and resource pools, and manipulates the VM control facilities in the kernel.

5.1.1 Rspecs and Rcaps

At the heart of the node manager is a table of

$$\text{rcap} \longrightarrow (\text{rspec}, \text{vm\_id})$$

bindings, where vm_id is the virtual machine identifier for a VM that the NM has created. The rspec is the basis for all object manipulation performed by the node manager. In standard object-oriented terminology it acts as a base class for all other objects (e.g., slices, pools). The PlanetLab node manager implements rcaps as 128-bit random values, where knowledge of an rcap corresponding to a particular object (rspec) lets the bearer perform any operation supported by the object. This lets slices easily transfer the ability to manipulate objects between each other.

As a concrete example of an rspec-subclass, consider the definition of a class representing a slice, which PlanetLab defines to be set of (name, value) pairs, including the following attribute names: vm_type, cpu_share, mem_limit, disk_quota, base_rate, burst_rate, and sustained_rate. Each attribute defines a particular resource allocation or constraint to be applied to instances of this slice. The mapping of each of these values to the underlying VMM parameters is described below.

5.1.2 Interface

The node manager interface consists of five operations for creating and manipulating resource pools and virtual machines:

```
rcap = create_root_pool(rspec, slice_name)
rcap = get_rcap( )
rspec = get_rspec(rcap)
rcap = split_pool(rcap, rspec)
bind(rcap, slice_name)
```

A root resource pool is created on behalf of some slice_name using the create_root_pool operation. This operation can only be invoked by the trusted owner-VM on the node and is generally used when the node is initialized. Typically, the owner-VM creates a slice pool for each trusted slice authority, 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 create_root_pool operation retrieves the rcap for its pool by calling the get_rcap operation. For example, a slice creation service calls this operation each time it runs on the node. Once a slice has retrieved the rcap for a pool of resources, it can learn the rspec associated with the pool by calling the get_rspec operation.

A slice that possesses an rcap for a pool can create a new 'sub-pool' by calling split_pool. 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 (rcap) is reduced accordingly (i.e., calling get_rspec 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.

5.1.3 VMM Support

The node manager invokes native VMM operations to create VMs and allocate resources to them. The current implementation supports only Linux-based VMs. It runs in a privileged VM context, and uses a combination of kernel modules to enforce isolation between unprivileged VMs. Specifically, Linux-based VM isolation is implemented through a combination of vservers [8] providing name space isolation and enforce disk quotas, CKRM [9] providing class-based management of CPU and memory resources as well as limits on per-class task count, and HTB [1] implements bandwidth allocation. As these individual mechanisms have been described elsewhere, we focus on the NM-level semantics of each rspec attribute.

PlanetLab currently supports only a Linux-based node manager, and as such, it defines only one vm_type: linux_vserver. The presence of this attribute in an rspec indicates that the node manager should create a corresponding VM. If the

attribute is not present, the rspec is interpreted to specify a resource pool, but no VM is instantiated.

Each node runs a proportional share CPU scheduler, with cycles allocated according to a cpu_share attribute. Slices are currently granted an equal number of shares, although it is also possible to grant a slice enough shares to guarantee it some fraction of each node's capacity. The scheduler is work conserving, meaning that any unused capacity is shared among active slices in proportion to the number of shares assigned to that slice.

Each slice is given a per-node upper bound on both the amount of disk space it can consume (the current default is disk_quota = 5GB) and the amount of memory it can use (no default is currently set for mem_limit). Disk quotas have not been a problem since most services do not have significant storage requirements; the 5GB limit effectively keeps services from maintaining unbounded log files and exceptions to the 5GB limit is made on a case by case basis.

In contrast, for memory usage it has not been possible to set a meaningful upper bound, largely due to "burst" needs of slices as they download software packages. Moreover, even a modest allocation of memory to each slice so seriously overbooks the available capacity as to be meaningless. As a consequence, the node manager lets slices consume as much memory as they need, but resets the VM with the largest memory usage on the node should swap space become 90% full. An email is sent to the service provider of the slice containing a summary of the memory usage of their specific VM. This has forced services to be conservative in their memory usage without requiring hard upper limits. In practice, only slices with memory leaks trigger this mechanism, and the email sent to a slice's service provide has helped in tracking down such bugs.

For bandwidth usage, the default settings are base_rate = 1Kbps, burst_rate = none, and sustained_rate = 1.5Mbps. The first implies that most slices do not receive a meaningful minimum transmission rate, but instead fairly share the available capacity. (A slice given a larger base_rate would be guaranteed that larger rate, plus receive a proportionally larger share of any unused capacity on each node.) Not setting an upper bound on the burst_rate means that each VM can burst outgoing packets up to the rate supported by the node, as set by the node's owner. This value is usually either 10Mbps or 100Mbps. The sustained_rate limits how much outgoing bandwidth the slice can consume, per node, over an extended period of time. The current implementation enforces this limit over a 24 hour period, meaning that a VM can transmit up to 16GB per day, after which it is limited to an overall maximum rate of 1.5Mbps for the remainder of the 24 hour period.

One important message to take away from this discussion is that the management authority–through the NM and VMM it runs on each node it manages–

effectively defines how strong or weak resource guarantees are (and how strong or weak isolation between VMs is). PlanetLab currently takes a largely "best effort" approach (with the option of giving certain slices stronger guarantees), while another management authority might offer stronger guarantees. Both can co-exist within the architectural framework defined in the previous section.

5.2 Management Authority

The PlanetLab management authority is responsible for the software running every node, including the boot process through which nodes download this software. This software includes an auditing service that is able to map network activity to the responsible slice, as well as an owner-VM that lets the node owner control various aspects of the node.

5.2.1 Database

The management authority 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 database includes 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)

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 management authority's slice creation service when the node boots.

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 the PlanetLab management authority inspect the node without allowing any slices to be instantiated or any network traffic to be generated. The management authority inspects this field to determine what action to take when a node contacts it, as described below.

There are both a GUI and programmatic interface to the database, which we do not describe in this report.

5.2.2 Boot Server

An organization (node owner) enters into a management agreement with PlanetLab through an out-of-band 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 upload their public keys into the management authority database, and create database entries for their sites and nodes. The nodes are initially marked (in the database) as being in the install state.

The site technical contact creates a BootCD and a network configuration floppy (pl_node.txt) and uses them to boot each of the site's nodes. The BootCD image is download from the web. PlanetLab provides a GUI interface that allows the technical contact to generate a pl_node.txt file for each of the organization's nodes. The file includes the name and IP address of the node, along with a unique nodekey generated by the PlanetLab management authority. The nodekey is also stored in the corresponding node tuple of the database. The following is an example pl_node.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"
```

Each node is configured to boot from the BootCD, which contains a minimal Linux system that initializes the node's hardware, reads the node's network configuration information from pl_node.txt, and contacts the boot server. The boot server 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 pl_node.txt, and uses HMAC [7] to authenticate itself to the boot server with this key. Each call to the server is independently authenticated via HMAC. The boot server also makes sure the source address corresponds to the one registered for the node, to ensure that the floppy 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 boot server 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 boot server, 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 contacts the boot server to verify whether its cached software packages are up-to-date, downloads any out-of-date packages, and then chain-boots the most current version. Finally, if the boot manager learns that the node is in debug state, it continues to run the Linux kernel it had booted from the CD, which lets PlanetLab operators ssh into and inspect the node. Both operators at PlanetLab and the site technical contacts may set the node's state (in the database) to debug or install, as necessary.

In addition to boot-time, there are two other situations in which the node and boot server synchronize. First, running nodes periodically contact the boot server 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 database, the boot server contacts the node to trigger the boot process, making it possible to bring the node into a safe state very quickly. This mechanism has been used to bring all the nodes on PlanetLab (200 at the time) into a safe state in less than ten minutes.

5.2.3 Owner-VM

Once a node boots–and both the VMM and NM are running–the NM immediately creates a local *owner-VM* that provides the local owner with access to the node. Technical and administrative contacts for the owning organization can ssh into the owner-VM using the keys they uploaded to the management authority.

The owner-VM, called site_admin, exists on each node, and provides a local account by which the site's technical contact can access information about the state of the node, review network logs, and define various node-specific configuration scripts. Among these is the list of slices (and resource pools) that are to be instantiated on the node at startup time.

Specifically, an initialization script running in site_admin invokes the NM to create a resource pool for each slice in the list. It does this using the create_root_pool operation described earlier in this Section. Typically, this list includes a pool for the slice creation service of each slice authority trusted by the node's owner. For example, a site that wants to ensure that slice my_favorite_cdn receives 10% of the node's capacity, and the remaining resources are distributed via PlanetLab's slice creation service (pl_conf, described below) would run the script:

```
rcap10 = create_root_pool(rspec10, my_favorite_cdn)
rcap85 = create_root_pool(rspec85, pl_conf)
rcap5 = create_root_pool(rspec5, pl_conf)
bind(rcap5, pl_conf)
```

The first line sets aside 10% of the node's capacity for a CDN service that the owner wishes to host; a VM corresponding to that service is started at some later time. The last three lines of the script set aside 90% of the node's capacity for redistribution to other slices: the second line creates a pool corresponding to 85% of the node's capacity and makes this pool available to pl_conf for redistribution, while the last two lines allocate and bind 5% of the node's capacity to pl_conf to ensure that the service has sufficient resources to run. When pl_conf starts, it uses the get_rcap operation to retrieve a handle on the pool assigned to it.

Note that the values in rspec85 effectively limit the resources that the node wishes to make available to VMs. For example, by setting peak_rate = 10Mbps the node owner specifies that slices cannot transmit at a higher rate than 10Mbps. Not specifying an upper bound causes the node manager to use whatever capacity (e.g., disk space or memory size) happens to be available on the node. Also note that with this script owners can disallow certain slices from being instantiated on their nodes.

5.2.4 Audit Service

The PlanetLab management authority includes an auditing service in the software installed on each node. This service, called netflow, runs in its own VM, and uses ulogd [14] to record information about every packet flow transmitted by the node. The service records

(prot, src_ip, src_port, dst_ip, dst_port, byte_cnt, pkt_cnt, vm_id)

tuples for each flow, retrieves a batch of such tuples from the kernel at five minute intervals (hence providing a timestamp for each tuple), and logs the tuples to disk. The auditing service is able to determine the slice associated with each flow because every socket call happens in the context of a VM (vserver context) and the NM records the VM-to-slice mapping.

The auditing service on each node also runs a web server that external users query the auditing information recorded on that node for the last 24 hours. This lets users determine the slice responsible for any suspect network traffic. Older audit logs are uploaded and archived by the management authority.

5.3 Slice Authority

The PlanetLab slice authority defines precisely what constitutes a system-wide slice. It maintains a database that records the persistent state of each registered slice, including information about every principal that has access to the slice.

5.3.1 Database

The slice authority maintains a database of registered principals, and the slices they have created. The database includes:

principal = (name, email, org, addr, key, role)
org = (name, addr, admin)
slice = (name, time, state, rspec, user[ ], nodes[ ])

where

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

Similar to the management authority database, 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).

Note that the main way the current implementation leverages the management and slice authorities being combined is that it runs a single database that includes tuples from both authorities. 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 providers, the same person is typically the admin in practice, and so there has been no reason to distinguish between these two cases.[2]

Both a programmatic and GUI front-end let principals register information about themselves, create slices, bind users to slices, and request that the slice be instantiated on a set of nodes. The current implementation involves a two-level validation process for principals: PlanetLab first enters into an off-line agreement with an organization, which identifies an administrative authority that is responsible for approving additional principals at that organization. The slice authority then lets this admin approve a principal at the organization, create slices (this results in a slice record in the database), and associate principals with slices. Those principals 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.

The rspec maintained by the PlanetLab 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

[2] All PlanetLab documentation apart from this report refers to the admin role as the *principal investigator*, but we elect to use the former as it is a more generic term for the corresponding user.

information to its point-of-presence on each node (i.e., its slice creation service). For example, the PlanetLab slice authority maintains a database identifier for each slice that can be used to determine when a slice is deleted and recreated. This is a critical idea: the slice authority defines exactly what constitutes a slice, and its database gives a slice permanence; node managers simply instantiate local VMs.

For each rspec attribute type, the database maintains a record of the privilege level required to change the associated value, thereby letting users (who have the least privilege) modify certain resource allocations. The database also lets default resource values be specified for each type by either the principal creating the slice or by the PlanetLab slice authority.

For each slice, the database 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 the PlanetLab slice authority, 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.

5.3.2 Slice Creation Service

The slice authority has a point-of-presence on each node–in the form of a slice creation service–that creates slices on that node. The PlanetLab slice creation service (pl_conf) supports slice creation in two different ways: by contacting the PlanetLab database periodically to get a list of currently active slices, and by acting as a local (per-node) agent for existing slices to redeem signed *tickets* that represent the ability to create a specific slice. Both of these mechanisms use a common back-end to interact with the node manager to create slices.

Figure 3 illustrates the two methods to instantiate a slice, but before either can be invoked, the slice must be created in the slice authority database. This is shown in the figure as the initial *create slice* arrow, and corresponds to multiple operations that we do not describe here. The slice is then instantiated on a set of nodes in one of two ways.

In the first case, corresponding to arrow *1a*, the slice authority causes the slice to be instantiated on each node (and sets the slice's state = central) by calling a

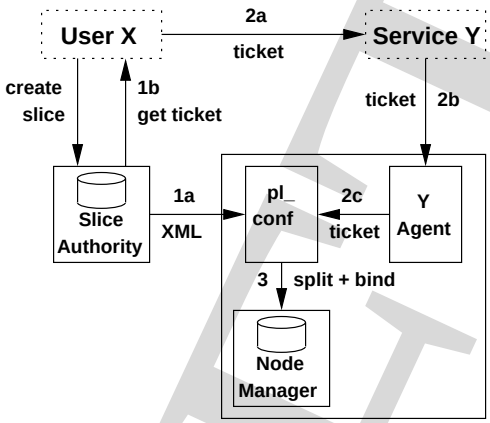

Figure 3: Mechanisms by which slices are created.

```
slice_instantiate(slice_name)
```

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

```
ticket = slice_get_ticket(slice_name)
```

operation (arrow *1b*). The user is then responsible for contacting the individual nodes and redeeming the ticket. In the example shown, the slice user passes the signed ticket to a third-party service, *Y*, which assumes responsibility for slice creation. This lets services such as *Y* provide an enhanced slice creation facility, perhaps providing lower latency or improved error reporting, on top of the basic functionality provided by the default slice creation service.

To complete the first mode of slice creation, pl_conf contacts the PlanetLab slice authority every 10 minutes and downloads a compressed XML representation of the relevant parts of the slice database, as shown by arrow *1a*. For each slice that is not already instantiated on the node, pl_conf pushes the corresponding XML element onto the back-end queue.

To complete the second mode of slice creation, pl_conf accepts XML-RPC requests (arrow *2c*) to create new slices from an existing local slice (e.g., *Y*'s agent), where these requests take the form:

```
rcap = create_slice(ticket, slice_name)
```

The ticket passed as an argument to create_slice was earlier obtained from the slice authority using the programmatic interface—it contains a complete slice

description, equivalent to one entry in the slice database, and is signed by the slice authority using a private key. pl_conf uses the corresponding public key to verify the authenticity of the ticket; if successful, the XML element contained in the ticket is pushed onto the back-end queue. We use various measures to try to prevent tickets being misused by malicious users, including expiry times on each ticket, publishing a ticket-revocation list as part of the XML file downloaded periodically by the service front-end, and frequently changing the key-pair used to sign the tickets.

The back-end of pl_conf takes requests from its queue and translates them into the corresponding calls to the node manager (arrow *3*). It first interprets service-specific fields in the rspec. For example, a unique database id for the slice is used to detect when a slice was deleted and recreated without the node ever observing the intermediate state, perhaps due to network connectivity problems. Similarly, a reference to an initialization script, also stored in the slice authority database, is translated into an encoded script that is passed directly to node manager to initialize the slice. The back-end uses the split_pool operation to obtain an rcap for the desired rspec, and then uses that rcap in a bind call to associate the rspec with the appropriate slice name, thus causing the slice to be instantiated on that node.

5.4 Infrastructure Services

As outlined in Section 4, the architecture explicitly decomposes management into separate functions rather than combining them in a single management entity. This is of value when trying to understand the design space for distributed virtualization, but it is also a design principle of PlanetLab, known as *unbundled management* [11, 3]. The consequence of unbundled management is a collection of infrastructure services—of which the slice authority's slice creation service is one example—which we contend has several benefits: (1) it keeps the node manager as minimal as possible; (2) it maximizes owner and provider choice, and hence, autonomy; and (3) it makes the system as a whole easier to evolve over time.

Several infrastructure services currently run on PlanetLab and play an active role in the management of the system. They include resource brokerage services used to acquire resources [2, 5], environment services that keep a slice's software packages up-to-date [6], monitoring services that track the health of nodes and slices [13, 4], and discovery services used to learn what resources are available [10].

6 Evolution of PlanetLab

This section identifies several ways in which current implementation of Planetlab already takes advantage of different degrees-of-freedom offered by the architecture, as well as ways in which we see PlanetLab evolving in the future.

6.1 Chain of Responsibility

The critical need to preserve the chain of responsibility in creating, sustaining, and scaling a distributed system spanning many node owners and service providers is tenent of the architecture. The requirement was not fully appreciated at the inception of PlanetLab, but has become clear as the system has grown far beyond a small, known set of researchers. New organizations will not join PlanetLab unless they can map network activity onto responsible principals, and many of the owners involved in the 231 security incidents reported in the last year would have taken their nodes offline had the complaints not been rapidly traced to the responsible party.

Moreover, while some of these incidents are due to the fact that experimental and sometimes buggy services run on PlanetLab, user mistakes and naive users are seldom the explanation; in our estimation, mistakes account for less than 10% of the reports. More common causes include IDS alarms at remote sites reacting to probing by services measuring the network, unexpected bandwidth usage by popular services, and violations of local Acceptable Use Policies by caching or other content services.

In every case, using the auditing service and the slice authority database to contact the responsible principals has been important in quickly resolving the complaint. Crucially, this process does not require pairwise agreements between owners and providers, an essential factor in PlanetLab's being being able to scale to 200 autonomous organizations.

6.2 Federation

Although PlanetLab currently combines the single management and slice authorities, decoupling the two offers an important degree of freedom as the system evolves. Specifically, allowing for a federation of such authorities has become a necessity as PlanetLab's reach becomes increasingly international (regional centers are planned in China and Europe) and as it crosses research/commercial boundaries (some companies are building "intranet" PlanetLabs). The architecture supports federation in two distinct stages.

In the first stage, multiple autonomous regions run their own management authority, while a single global slice authority grants service providers access to nodes across management boundaries. This distributes the node management problem without balkanizing slices. The second stage sees multiple slice authorities, but distinct from management domains. There may remain a single "research" slice authority, for example, and it is easy to imagine a "public good" slice authority that approves only those service providers that offer network reports or SETI@HOME style services.

We expect each slice authority to establish peering relationships with others, letting it create slices on nodes that do not run its own slice creation service. In the limit, slice authorities simply become "naming" authorities, largely decoupled from the slice creation services on nodes. Although not mandated by the architecture, we expect slice authorities to communicate with each other via pl_conf-style tickets (signed rspecs).

6.3 Decoupling Slices and Pools

The architecture decouples the process of creating a slice from the act of acquiring resources that can be bound to the slice. This makes it possible to first use the slice authority's slice creation service to create VMs on a set of nodes, and only later acquire resources that can be bound to each VM, perhaps assisted by a brokerage service that implements a market for node resources. Two scenarios illustrate the benefit of this decoupling.

Today, users interface with the PlanetLab slice authority to create slices that span one or more PlanetLab nodes, relying on the standard slice creation service to instantiate a VM on each node. Once a slice has been created, with a default set of resources specified by the slice authority, users can then go to brokerage services to request additional resources; e.g., by bidding in an auction, or queueing up for additional capacity on a set of nodes during some time slot. These brokerage services have a point-of-presence on each node, which lets them call the local node manager to assign some fraction of their pool to a client slice. These allocations happen at a much finer lifetimes (measured in hours) than do slices (measured in weeks). Brokerage services currently acquire their pool indirectly through the slice authority: each node grants all their resources to pl_conf, which has a policy of granting some fraction of this capacity to brokerage services for redistribution to existing slices.

An alternative gets pl_conf out of the loop, with node owners interacting directly with brokerage services. In such a scenario, a node owner runs one or more slice creation services. However, instead of pre-allocating some pool of resources to each of these slice creation services, the node owner elects to grant all of the

node's resources to a single brokerage service, expecting it to sell those resources on a market.

6.4 Delegation

The principle of unbundled management requires that one service is permitted to manipulate another slice and/or the resources associated with that slice. Delegation is one mechanism for supporting such operations.

PlanetLab currently supports delegation at two levels. At the node level, an rcap acquired for an object represented by an rspec can easily be transferred from the object owner to a service that the owner trusts to manipulate the object. Transferal of an rcap requires no intervention by the local node manager; just the owner making the 128-bit value known to the service. For example, a slice may request that a local brokerage service temporarily grant it a boost in CPU share by contacting the brokerage service and passing an rcap for its own VM.

PlanetLab also supports delegation at the network level. While nothing in the architecture directly mandates this form of delegation, we have implemented just such a scheme in the current slice authority, as previously described in Section 5.3.2. To recap, PlanetLab lets users authenticate themselves to the authority, perform a sequence of operations on the database tuples, then request that the slice authority generate a cryptographically-signed *ticket* that represents the operations that the authenticated user is allowed to perform directly. The user can then delegate responsibility for performing those operations–e.g., slice creation or resource allocation–to another service by giving the signed ticket to that service.

6.5 Owner Autonomy

In discussing the possibilities for putting excess resources on a market, it is important to not lose sight of the fact that owners retain full control over their nodes. They select a management authority, they determine which slice authorities they trust (and by implication, which service providers they will allow), and they make the root resource allocation decision. This latter point means that an owner can first ensure that sufficient resources are set aside for any services that provide value to local users, and then make the remaining resources available on a market, perhaps with some fraction of the remainder earmarked for research or public-good purposes.

References

[1] Linux Advanced Routing and Traffic Control. `http://lartc.org`.

[2] A. AuYoung, B. Chun, C. Ng, D. Parkes, J. Shneidman, A. Snoeren, and A. Vahdat. Bellagio: An Economic-Based Resource Allocation System for PlanetLab. http://bellagio.ucsd.edu/about.php.

[3] A. Bavier, M. Bowman, D. Culler, B. Chun, S. Karlin, S. Muir, L. L. Peterson, T. Roscoe, T. Spalink, and M. H. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proc. 1st NSDI*, San Francisco, CA, Mar. 2004.

[4] P. Brett, R. Knauerhase, M. Bowman, R. Adams, A. Nataraj, J. Sedayao, and M. Spindel. A Shared Global Event Propagation System to Enable Next Generation Distributed Services. In *Proc. 4th WORLDS*, San Francisco, CA, Dec. 2004.

[5] David Lowenthal. Sirius: A Calendar Service for PlanetLab. http://snowball.cs.uga.edu/~dkl/pslogin.php.

[6] Juston Cappos and John Hartman. Stork: A Software Packagement Management Service for PlanetLab. http://www.cs.arizona.edu/stork.

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

[8] Linux VServers Project. http://linux-vserver.org.

[9] S. Nabah, H. Franke, J. Choi, C. Seetharaman, S. Kaplan, N. Singhi, V. Kashyap, and M. Kravetz. Class-based prioritized resource control in Linux. In *Proc. OLS 2003*, Ottawa, Ontario, Canada, Jul 2003.

[10] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed Resource Discovery on PlanetLab with SWORD. In *Proc. 4th WORLDS*, San Francisco, CA, Dec. 2004.

[11] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. HotNets–I*, Princeton, NJ, Oct 2002.

[12] L. Peterson and T. Roscoe. The Design Principles of PlanetLab. Technical Report 04-021, PlanetLab, June 2004.

[13] Vivek Pai and KyoungSoo Park. CoMon: A Monitoring Infrastructure for PlanetLab. http://comon.cs.princeton.edu.

[14] H. Wilte. ulogd: Userspace Packet Logging for netfilter. http://gnumonks.org/projects.