Operating System Support for Planetary-Scale Network Services

Andy Bavier, Larry Peterson, Mike Wawrzoniak, Scott Karlin, Tammo Spalink, Timothy Roscoe, David Culler, Brent Chun, Mic Bowman

Abstract

PlanetLab is a geographically distributed overlay network designed to support the evaluation and deployment of planetary-scale network services. Two high-level goals shape its design. First, to enable a large research community to share the infrastructure, PlanetLab provides *distributed virtualization*, whereby each service runs in an isolated slice of PlanetLab's global resources. Second, to support competition among multiple network services, PlanetLab decouples the operating system running on each node from the network-wide services that define PlanetLab, a principle sometimes referred to as *unbundled management*. This paper describes how PlanetLab realizes the goals of distributed virtualization and unbundled management, with a focus on the OS running on each node.

1 Introduction

PlanetLab is a geographically distributed overlay network designed to support the evaluation and deployment of planetary-scale network services [25]. It currently includes nearly 200 machines spanning 80 sites and 15 countries, and peers with 2600 Internet autonomous systems. It supports over 120 active research projects focused on a wide range of services, including file sharing and network-embedded storage [9, 19, 28], content distribution networks [32], routing and multicast overlays [1, 7], QoS overlays [31], scalable object location services [2, 26, 27, 30], anomaly detection mechanisms [8], and network measurement tools [29].

As a distributed system, PlanetLab is characterized by a unique set of relationships between principals—users, administrators, researchers, service providers, and so on—which make the design requirements for its operating system different from traditional hosting services or timesharing systems.

The first relationship is between PlanetLab as an organization, and the institutions that own and host PlanetLab nodes: the former has administrative control over the nodes, but local sites also need to enforce policies about how the nodes are used, and the kinds and quantity of network traffic the nodes can generate. This implies a need to share control of PlanetLab nodes.

The second relationship is between PlanetLab and its users: currently, researchers evaluating and deploying planetary-scale services. Users must have access to the platform. This implies a distributed set of machines that must be shared between users in a way they will find useful. A PlanetLab "account", together with associated resources, must therefore span multiple machines. We call this abstraction a *slice*, and implement it using a technique called *distributed virtualization*.

A third relationship exists between PlanetLab and those researchers contributing to the system by designing and building *infrastructure services*, that is, services that contribute to the running of the platform as opposed to being merely applications of it. Not only must these services run in a slice, but PlanetLab must support multiple, parallel services with similar functions developed by different groups. We call this principle *unbundled management*, and it imposes its own requirements on the system.

Finally, PlanetLab exists in relation to the rest of the Internet. Experience shows that the experimental networking performed on PlanetLab can easily impact many external sites' intrusion detection and vulnerability scanners. This leads to requirements for policies limiting what traffic PlanetLab users can send to the rest of the Internet, and a way for concerned outside individuals to find out exactly why they are seeing unusual traffic from PlanetLab. The rest of the Internet needs to feel safe from PlanetLab.

The contribution of this paper is to describe in more detail the requirements that result from these relationships, and how PlanetLab fulfills them using a synthesis of operating systems techniques. This contribution is partly one of "design" because PlanetLab is a work-in-progress, and only time will tell what infrastructure services will evolve to give it fuller definition. At the same time, however, this is an "experiences" paper because the design is largely a response to having hundreds of users stressing PlanetLab over the last 15 months.

2 Requirements

This section defines distributed virtualization and unbundled management, and identifies the requirements each places on PlanetLab's design.

2.1 Distributed Virtualization

PlanetLab services and applications run in a *slice* of the platform: a set of nodes on which the service receives a fraction of each node's resources, in the form of a virtual machine (VM) on the node. Virtualization and virtual machines are, of course, a well-established concept. What is new in PlanetLab is *distributed virtualization*: the acquisition of a distributed set of VMs that are treated as a single, compound entity by the system.

To support this concept, PlanetLab must have facilities to create a slice, initialize it with sufficient persistent state to boot the service or application in question, and bind the slice to a set of resources on each constituent node. In concrete terms, the minimal persistent state for a service comprises a set of keys permitting `ssh` access to the VM on each node, or a URL from which the newly created slice downloads a boot script. The resources bound to a VM include both physical resources on the node (e.g., cycles, memory, and link bandwidth), and logical resources that the VM controls (e.g., an allotment of TCP and UDP port numbers).

2.1.1 Underspecified Abstraction

While the per-node properties of a slice are well-defined (described in more detail in Section 4), the way that a slice is composed from VMs on different nodes depends on the *slice creation service* employed. Consistent with the principle of unbundled management, PlanetLab needs to support multiple, competing slice creation services, and consequently, must specify as little as possible, *a priori*, what the global properties of a slice should be. Ultimately, what a slice *is*, above and beyond a collection of VMs, will be defined by the service that creates it; that is, slices are underspecified. This in turn implies two requirements on the PlanetLab OS:

1. It must provide a *low-level interface for creating a VM* that can be shared by multiple slice creation services. It must also host a simple, "bootstrapping" slice creation service to create initial slices, including the slices that other slice creation services run in.

2. It must provide an *efficient, low-level programming environment* that gives service developers as much latitude as possible in defining the behavior of a slice.

This second requirement has several implications. For example, it means that the PlanetLab OS should not provide tunnels that connect the constituent VMs into any particular overlay configuration, but should instead provide an interface that allows each service to define its own topology on top of the fully-connected Internet. Similarly, it should not prescribe a single language or runtime system, but instead allow slices to load whatever environments or software packages they need.

An important technical issue that will influence how the slice abstraction evolves is how quickly a network-wide slice can be instantiated. Applications like the ones listed in the Introduction are relatively long-lived (although possibly modified and restarted frequently), and hence the process of creating the slice in which they run can be a heavyweight operation. On the other hand, a facility for rapidly establishing and tearing down a slice (analogous to creating/destroying a network connection) would lead to slices that are relatively short-lived, for example, a slice that corresponds to a communication session with a known set of participants. We evaluate the performance of the current slice creation mechanism in Section 5. It is not yet clear what other slice creation services the user community will provide, or how they will utilize the capability to create and destroy slices.

2.1.2 Isolating Slices

PlanetLab must isolate slices from each other, thereby maintaining the illusion that each slice spans a distributed set of private machines. The same requirement is seen in traditional operating systems, except that in PlanetLab the slice is a distributed set of VMs rather than a single process or image. Per-node resource guarantees are also required: for example, some slices run time-sensitive applications, such as network measurement services, that have soft real-time constraints reminiscent of those provided by multimedia operating systems. This means three things with respect to the PlanetLab OS:

- It must allocate and schedule node resources (cycles, bandwidth, memory, and storage) so that the runtime behavior of one slice on a node does not adversely affect the performance of another on the same node. Moreover, certain slices must be able to request a minimal resource level, and in return, receive (soft) real-time performance guarantees.

- It must either *partition or contextualize the available name spaces* (network addresses, file names) to prevent a slice interfering with another, or gaining access to information in another slice. This partitioning and contextualization must be coordinated over the set of nodes in the system.

- It must *provide a stable programming base* that cannot be manipulated by code running in one slice in a way that negatively affects another slice. In the context of

a Unix- or Windows-like operating system, this means that a slice cannot be given root or system privilege.

2.1.3 Isolating PlanetLab

The PlanetLab OS must also protect the outside world from slices. PlanetLab nodes are simply machines connected to the Internet; as a consequence, buggy or malicious services running in slices have the potential to affect the global communications infrastructure. Due to PlanetLab's widespread nature, this impact goes far beyond the reach of an application running on any single computer. This places two requirements on the PlanetLab OS.

- It must *thoroughly account resource usage*, and make it possible to place *limits* on resource consumption so as to mitigate the damage a service can inflict on the Internet. Proper accounting is also required to isolate slices from each other. Here, we are concerned both with the node's impact on the hosting site (e.g., how much network bandwidth it consumes) and remote sites completely unaffiliated with PlanetLab (e.g., sites that might be probed from a PlanetLab node). Furthermore, the local administrators of a PlanetLab site and PlanetLab as an organization need to collectively set these policies for a given node.

- It must make it easy to *audit* resource usage, so that *actions* (rather than just resources) can be accounted to slices after the fact. Unlike traditional timesharing systems, where the interactions between users and unsuspecting outside entities is inherently rare, this concern about how users (or their services) affect the outside world is a novel requirement for PlanetLab.

Security was recognized from the start as a critical issue in the design of PlanetLab. However, effectively limiting and auditing legitimate users has turned out to much more significant an issue than securing the OS to prevent malicious users from hijacking machines. To date, there have been no security breaches of note, but a single PlanetLab user running TCP throughput experiments on UC Berkeley nodes managed to consume over half of the available bandwidth on the campus gateway over a span of days. Also, many experiments (e.g., Internet mapping) have triggered IDS mechanisms, resulting in complaints that have caused local administrators to pull the plug on nodes. The Internet has turned out to be unexpectedly sensitive to the kinds of traffic that experimental planetary-scale services tend to generate.

2.2 Unbundled Management

Planetary-scale services are a relatively recent and ongoing subject of research, and in particular, this includes the services required to manage PlanetLab as a global platform. Moreover, it is an explicit goal of PlanetLab to allow independent organizations (in this case, research groups) to deploy alternative services in parallel, allowing users to pick which ones to use. This applies to application-level services targeted at end-users, as well as *infrastructure services* used to manage and control PlanetLab itself. Slice creation is one example of an infrastructure service. Others include resource discovery, topology discovery and routing support, performance monitoring, and software distribution. The key to unbundled management is to allow parallel infrastructure services to run in their own slices of PlanetLab, and to evolve over time.

This is a new twist on the traditional problem of how to evolve a system, where one generally wants to try a new version of some service in parallel with an existing version, and roll back and forth between the two versions. In our case, multiple competing services are simultaneously evolving. The desire to support unbundled management leads to two requirements for the PlanetLab OS.

- To minimize the functionality subsumed by the PlanetLab OS—and maximize the functionality running as services on top of the OS—only local (per-node) abstractions should be directly supported by the OS, allowing all global (network-wide) abstractions to be implemented by infrastructure services. We have already discussed this in relation to slice creation, but it extends to almost all aspects of PlanetLab's infrastructure.

- To maximize the opportunity for services to compete with each other on a level playing field, the interface between the OS and these infrastructure services must be *sharable*, and hence, without special privilege. In other words, rather than have a single privileged application controlling a particular aspect of the OS, the PlanetLab OS potentially supports many such management services. One implication of this interface being sharable is that it must be well-defined, explicitly exposing the state of the underlying OS. In contrast, the interface between an OS and a privileged control program running in user space is often ad hoc since the control program is, in effect, an extension of the OS that happens to run in user space.

Operating system design often faces a tension between implementing functionality in the kernel and running it in user space, the objective often being to minimize kernel code. Like many VMM architectures, the PlanetLab OS faces an additional, but analogous, tension between what can run in a slice or VM, and functionality (such as slice user authentication) which requires extra privilege or access.

In addition to these, there is a third aspect to the problem which is peculiar to PlanetLab: functionality that can be implemented by parallel, competing subsystems, versus

mechanisms which by their very nature can only be implemented once (such as slice creation). The PlanetLab OS strives to minimize the latter, but there remains a core of non-kernel functionality which has to be unique on a node.

2.3 Practical Concerns

In addition to the requirements imposed by slices and unbundled management, practical concerns have also influenced the PlanetLab OS. The research community was ready to use PlanetLab the moment the first machines were deployed. Waiting for a new OS tailored for broad-coverage services was not an option; besides, without first gaining some experience, no one could fully understand what such an OS should look like. Moreover, experience with previous testbeds strongly suggested two biases of application writers: (1) they are seldom willing to port their applications to a new API, and (2) they expect a full-featured OS rather than a minimalist API tuned for OS designer's research agenda.

This suggested the strategy of starting with a popular full-featured OS—we elected to use Linux—and incrementally transforming it based on experience. Finding the right way to balance the requirements outlined above with the desire to support a full-featured OS like Linux has been the greatest source of tension (both technically and socially) in the design of the PlanetLab OS.

3 Design Alternatives

The PlanetLab OS is a synthesis of existing operating systems abstractions and techniques, applied to the new context of a distributed platform, and motivated by the requirements derived in the previous section. This section discusses how PlanetLab's requirements recommend certain approaches over others, and in the process, discusses related work.

The first challenge of the PlanetLab OS is to provide a virtual machine abstraction for slices; the question is at what level. At one end of the spectrum, full hypervisors like VMware completely virtualize the physical hardware and thus support multiple, unmodified operating system binaries. If PlanetLab were to supply this low level of virtualization, each slice could run its own copy of an OS and have access to all of the devices and resources made available to it by the hypervisor. This would allow PlanetLab to support OS kernel research as well, and provide better isolation by removing contention for OS resources. The cost of this approach is performance: VMware cannot support the number of simultaneous slices required by PlanetLab due to the large amount of memory consumed by each machine image. Thus far, the PlanetLab community has not required the ability to run multiple operating systems, and so PlanetLab is able to take advantage of the efficiency of supporting a single OS API.

A slightly higher-level approach is to use *paravirtualization*, proposed by so-called isolation kernels like Xen [4] and Denali [33]. Short of full virtualization of the hardware, a subset of the processor's instruction set and some specialized virtual devices form the virtual machine exported to users. Because the virtual machine is no longer a replica of a physical machine, operating systems must be ported to the new "architecture", but this architecture can support virtualization far more efficiently. Paravirtualizing systems are not yet mature, but if they can be shown to scale, they represent a promising technology for PlanetLab.

The approach we adopted is to virtualize at the system-call level, similar to commercial offerings like Ensim [12], and projects such as User Mode Linux [10], BSD's Jail [18], and Linux vservers [17]. Such high-level virtualization adequately supports PlanetLab's goals of supporting large numbers of overlay services, while providing reasonable assurances of isolation.

A second, orthogonal challenge is to isolate virtual machines. Operating systems with the explicit goal of isolating application performance go back at least as far as the KeyKOS system [16], which provided strict resource accounting between mutually antagonistic users. More recently, isolation mechanisms have been explored for multimedia support, where many applications require soft real-time guarantees. Here the central problem is *crosstalk*, where contention for a shared resource (often a server process) prevents the OS from correctly scheduling tasks. This has variously been addressed by sophisticated accounting across control transfers [23], scheduling along data paths in Scout [24], or entirely restructuring the OS to eliminate server processes in the data path [20]. The PlanetLab OS borrows isolation mechanisms from Scout, but the key difference is in how these mechanisms are controlled, since each node runs multiple competing tasks that belong to a global slice, rather than a purely local set of cooperating tasks.

Having settled on virtualization at the system call level, the third challenge is how to provide low-level access to some virtual devices, particularly the network. Vertically-structured operating systems like Exokernel and Nemesis have explored allowing access to raw network devices by using filters on send and receive [5, 11]. The PlanetLab OS does something similar by providing shared network access using a "safe" version of the raw socket interface. The primary difference between the PlanetLab OS and exokernels is the degree to which devices must be shared. It is not enough to provide "raw" access to a device so that a user-level OS can manage it. Instead, the kernel must take responsibility for fairly sharing access among multiple competing services. It is also the case that the services running on PlanetLab are more interested in building network-wide functionality than in having low-level access to devices. Thus, there is little penalty in providing a full-featured file system interface, for example, rather than sup-

port for low-level disk access. Similarly, providing access to the network at the IP level is sufficient.

A fourth challenge is the distributed coordination of resources. This problem has been explored in the context of Condor [22] and more recently the Open Grid Services Architecture [14]. However, both these systems are aimed at the execution of batch computations, rather than the support of long-running network services. They also seek to define complete architectures within which such computations run. In PlanetLab the requirements are rather different: the platform must support multiple approaches to creating and binding resources to slices. To illustrate this distinction, we point out that both the Globus grid toolkit and the account management system of the Emulab testbed [34] have been implemented above PlanetLab, as have more service-oriented frameworks like SHARP [15].

A final, and somewhat new challenge is to support the monitoring and management of a large distributed infrastructure. On the network side, commercial management systems such as HP OpenView and Micromuse Netcool provide simplified interfaces to routing functionality, service provisioning, and equipment status checks. On the host management side, systems such as IBM's Tivoli and Computer Associates' UniCenter address the corresponding problems of managing large numbers of desktop and server machines in an enterprise. Both kinds of systems are aimed at single organizations with well-defined applications and goals, seeking to manage and control the equipment they own. Managing a wide-area, evolving, federated system like PlanetLab (or the Internet as a whole) poses different challenges. Here, we are pretty much on our own.

4 Planetlab OS

This section defines the PlanetLab OS, the per-node software base on top of which the global slice abstraction is built. The PlanetLab OS consists of a Linux 2.4-series kernel with patches for vservers and hierarchical token bucket packet scheduling; the SILK (Scout in Linux Kernel) module that provides CPU scheduling, network accounting, and safe raw sockets; and the node manager, a trusted domain that contains slice bootstrapping machinery and node monitoring and management facilities. We describe the functionality provided by these components and discuss how it is used to implement slices, focusing on four main areas: the VM abstraction, resource allocation, controlled access to raw network sockets, and system monitoring.

4.1 Virtual Machine

A slice corresponds to a set of virtual machines. Each VM, in turn, is implemented as a *vserver* [17]. The vserver mechanism is a patch to the Linux 2.4 kernel that provides the illusion of multiple, independently managed virtual servers running on a single machine; each slice maps onto a unique vserver on each of its constituent physical machines.

Vservers are the principal mechanism in PlanetLab for providing virtualization on a single node, and contextualization of name spaces—user identifiers, files, etc. As well as providing security between slices sharing a node, they provide a limited root privilege which allows slices to customize their VM as if it was a dedicated machine. Vservers also correspond to the resource containers used for isolation, which we discuss in section 4.2. This section describes the virtualization provided by vservers.

4.1.1 Interface

Vservers provide virtualization at the system call level by extending the non-reversible isolation provided by `chroot` for filesystems to other operating system resources such as processes and SysV IPC. Processes within a vserver are given full access to files, processes, SysV IPC, network interfaces, and accounts which can be named in their containing vserver and are denied access to all other operating system resources otherwise. Each vserver is also given a weaker form of `root` along with its own UID/GID namespace which allows each vserver to have its own superuser while at the same time not compromising the security of the underlying machine.

Despite having only a subset of the true superuser's capabilities, vserver `root` is still useful in practice. It allows for modification of the vserver's root filesystem which, for example, allows users to customize what software packages are installed in a particular vserver. Combined with per-vserver UID/GID namespaces, it allows vservers to implement their own internal account management schemes (e.g., by maintaining a per-vserver `/etc/passwd` and running an `sshd` daemon a different TCP port), which provides the basis for integration with other wide-area testbeds such as NetBed [34] and RON [1]. Finally, as we gain additional experience on what privileges services actually require, adding additional extensions to the existing set of Linux capabilities provides a natural path towards exposing privileged operations in a controlled manner.

Vservers communicate with one another via local IP, and not local sockets or other system IPC functions. This strong separation between slices simplifies resource management and isolation between vservers, since the interaction between two vservers is independent of whether they exist on the same node, and slices must therefore be written accordingly.

One namespace that is not contextualized to slices is that of network addresses (IP address and port numbers): achieving this would entail either giving each slice its own IP address on a physical node, or else hiding each vserver behind a per-machine Network Address Translator. We rejected both these options in favor of slices sharing port numbers and addresses on a single node.

4.1.2 Implementation

Virtualization in vservers is implemented at the system call interface and isolation is enforced based on the idea of a security context. Each vserver on a machine is assigned a unique security context, and each process running on that machine is associated with a specific vserver through its security context. A process's security context is assigned via a new system call and inherited by all of the process's descendants. Isolation between different vservers is enforced through the system call interface using a combination of a process's security context and UID/GID when checking access control privileges and deciding what information should be exposed to a given process. All of the aforementioned mechanisms are implemented as part of the baseline vserver patch to the kernel. We have also implemented a number of utility programs that simplify the creation and destruction of vservers and allow users to be transparently redirected into vservers for their specific slices using the SSH protocol.

On PlanetLab, a vserver is created by first choosing a unique security context and creating a mirror of a reference root filesystem for the vserver using hard links and the immutable and immutable invert filesystem bits. Next, two Linux accounts are created using our utility programs, one in the node's primary vserver and one in the vserver just created. Both accounts use a login name identical to that of the slice. The account in the main vserver is specified to use a special shell, `/bin/vsh`. This shell is a modified `bash` shell that we have written which performs the following four actions upon login: a switch to the slice's vserver security context, a `chroot` to the vserver's root filesystem, relinquishing of a subset of the true superuser's capabilities, and redirection into an account in the vserver with an identical login name. The end result of this two account arrangement is that users accessing their virtual machines remotely via SSH/SCP are transparently redirected into the appropriate vserver and need not modify any of their existing service management scripts.

By virtualizing above a standard Linux kernel, vservers achieve scalability through large amounts of resource sharing and no active state for idle vservers. Sharing of physical memory and disk space is substantial. For physical memory, savings are accrued by having a single copy of the kernel, a single copy of all kernel and user-level daemons, and, perhaps most importantly, sharing of read-only and copy-on-write memory segments across unrelated vservers. Disk space sharing is also significant due to the introduction of the filesystem immutable invert bit which allows for a primitive form of filesystem copy-on-write (COW). By using COW on chrooted vserver root filesystems, vserver disk footprints are reduced to just 5.7% of what would be requiring with copying (Section 5.1). Achieving comparable amounts of sharing in a virtual machine monitor or isolation kernel approach is strictly harder, albeit the isolation guarantees are different.

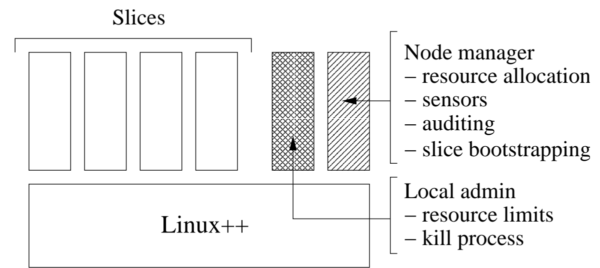

Figure 1: PlanetLab vserver contexts

A weaker version of `root` allows each vserver to have its own superuser while at the same time not compromising the security of the underlying machine. Superuser privileges are granted safely to vservers by having each vserver `root` relinquish a subset of the true superuser's capabilities and by leveraging the isolation already provided by vservers to limit the scope of a vserver `root`'s activities to the containing vserver. In Linux, access control for privileged operations is based on a capabilities system. Capabilities determine whether privileged operations such as pinning physical memory or rebooting the machine are allowed or disallowed. Vserver `root` is denied all capabilities that could undermine the security of the machine (e.g., accessing raw devices) and granted all other capabilities.

4.1.3 Discussion

Sections 4.1.1 and 4.1.2 describe the capabilities of the vserver contexts in which every slice runs. However, each PlanetLab node supports two special security contexts with additional capabilities, as illustrated in Figure 1. The first is a privileged context called the *node manager*. The node manager comprises components that should be considered part of the PlanetLab OS but that reside outside of the kernel, along with pieces of the slice bootstrapping facility. Essentially, the node manager context runs with standard `root` capabilities and includes the machinery to create a slice, initialize its state, and assign resources to it; sensors that export information about the node; and a traffic auditing service. The second context provides weaker system privileges to site administrators, with the goal of giving them a set of tools to manage the nodes without providing them with full `root` access. The administrative context has a complete view of the entire machine and can cap the node's total outgoing bandwidth rate, kill arbitrary processes, run `tcpdump` to monitor traffic on his network, and so on. Of course, the site administrator always has recourse to flipping the node's power switch if he suspects things have gone very wrong.

Virtualizing above the kernel comes at a cost: weaker guarantees on isolation and additional challenges for eliminating QoS crosstalk. Unlike virtual machine monitors and isolation kernels that provide isolation at a low-level,

vservers implement isolation at the system call interface. Hence, a malicious vserver that exploits some obscure bug in the Linux operating system could potentially gain control of the underlying operating system, and hence compromise security of the machine. (In practice, we have yet to observe such an incident. However, in principle, such an attack is still possible.) Such an attack would not be possible using a lower-level VM monitor. Another potential cost incurred by virtualizing above the kernel is QoS crosstalk. Eliminating all forms of QoS crosstalk (e.g., interactions through the Linux buffer cache) is strictly harder in a vserver-based approach. As described in the next section, however, isolation can be accomplished at a fairly deep level.

4.2 Resource Allocation

A key feature of slices is the isolation they provide between services. A small sampling of experiences on PlanetLab serves to motivate the isolation mechanisms described in this section. For example:

- Just before the SOSP deadline, one experiment consumed all available file descriptors on several nodes, effectively preventing other slices from using the disk or network.
- Several times a day a PlanetLab node somewhere runs out of disk space, often as a result of out-of-control event logging within a slice.
- Buggy code in one slice recently went into an infinite loop and consumed 100% of the CPU on 50 nodes at once.

The PlanetLab OS's node manager supports slice bootstrapping by providing a low-level, uniform interface for obtaining resources on a node. The node manager generates resource "tickets" called *rcaps* (resource capability) that can be redeemed for specific node resources. A service that wants to run on the node acquires the appropriate rcaps and presents them to the node manager. At this point, the node manager can perform admission control and either accept or reject an rcap; assuming the rcap is accepted, the node manager binds the resources associated with the rcaps to the service's virtual machine.

Note that the node manager does not make any policy decisions regarding how resources are used; policy is imposed by infrastructure services running on top of the node manager and by local site administrators. As an example of the latter, the site administrator may set an upper bound on the total outgoing bandwidth of a node, and the node manager will heed this limit when issuing rcaps. The rest of this section describes the node manager interface and the mechanisms used to allocate resources and enforce limits.

4.2.1 Interface

An rcap represents rights to use specific resources on a node. An rcap is a 128-bit opaque value, knowledge of which provides access to the associated resources. Each node manager tracks both the set of resources available and a mapping between dedicated resources and rcaps. The node manager provides trusted services with the following operation to create an rcap:

```
acquire(rspec) -> rcap
```

The `acquire` operation takes an *rspec* (resource specification) as an argument. The node manager creates and returns a new rcap and remembers the mapping between the two; i.e., the rcap serves as an index into a table of rspecs.

The rspec is used to describe a slice's access privileges to resources over time. Each rspec consists of a list of reservations, proportional share values, or upper bound limits for specific resources. It also specifies the start and end times of the interval over which these values apply. The rspec allows reservation of a virtual machine, rates for CPU and network bandwidth consumption, memory pages, disk space, specific TCP and UDP port numbers, and a pool of file descriptors. Proportional sharing, rather than hard reservation, is also supported for CPU and bandwidth. Finally, upper bound limits can be expressed for the rates of incoming and outgoing network bandwidth, rate at which CPU cycles are received, number of distinct IP flows, number of UDP and TCP ports opened, number of file descriptors held, and memory pages and disk space consumed. When used for slice resource provisioning, rspecs generally include only reservation and proportional share privileges. Upper bound privileges are used to leverage resource schedulers to control slice behavior. Reservation privileges provide guaranteed resource bounds, meaning that exactly the specified share of resources will be available during the time window. For proportional share best effort tickets, the resources available to the virtual machine during the time interval depends on the share value relative to other active privileges.

Once acquired, rcaps can be passed from one service to another, where the resources associated with the rcap are bound to a virtual machine (vserver) at some later time (slice creation time) using the `bind` operation:

```
bind(slice_id, rcap)
```

The `bind` operation takes a slice identifier and an rcap as arguments. To incorporate node resources, a slice first needs a virtual machine rcap; without this, other rcaps are useless. Once the virtual machine is created, the node manager associates the resources described by additional rcaps with it.

The initial node manager implementation is visible as an XML-RPC server listening on a well known TCP port. The node manager interface is not remotely network accessible,

that is, a specific node manager's operations are only available to services running on that node. Infrastructure services provide communication authentication and encryption operations (SSL, SSH, etc) that are too complex to support in the node manager, allowing for competing implementations to evolve over time. All nodes are bootstrapped with an infrastructure service that assists other services in contacting remote node managers by acting as a proxy.

4.2.2 Implementation

Static resources, such as memory pages, disk space, and file descriptors, can be isolated using per-slice reservations and limits. These are implemented by wrapping the appropriate system calls or kernel functions to intercept allocation requests. Each request is either accepted or denied based on the slice's current overall usage, and if it is accepted, the slice's counter is incremented by the appropriate amount. Currently the ability to reserve and limit these resources within a virtual machine is being added to the vservers implementation.

For dynamic resources such as the CPU and network link, two possible approaches to providing isolation are *fairness* and *guarantees*. Fairness ensures that each of the N slices running on a node receives no less than $1/N$ of the available resources during periods of contention, while guarantees provide a slice with a reserved amount of the resource. In the latter case, the scheduling mechanism that provides guarantees as a consequence provides isolation. PlanetLab provides CPU and bandwidth guarantees for slices that request them, and "fair best effort" service for the rest. Additionally, resource limits on outgoing traffic and CPU usage can protect the rest of the world from PlanetLab.

The Hierarchical Token Bucket (htb) queuing discipline of the Linux Traffic Control facility (tc) [21] is used to cap the total outgoing bandwidth of a node, cap per-vserver output, and provide bandwidth guarantees and fair service between vservers. The node administrator configures the root token bucket with the maximum rate at which the site is willing to allow traffic to leave the node. At vserver startup, a token bucket is created that is a child of the root token bucket; if the service requests a guaranteed bandwidth rate, the token bucket is configured with this rate, otherwise it is given a minimal rate (5Kbps) for "fair best effort" service. Packets sent by a vserver are tagged in the kernel and subsequently classified to the vserver's token bucket. The htb queuing discipline then provides each child token bucket with its configured rate, and fairly distributes the excess capacity from the root to the children that can use it in proportion to their rates. A bandwidth cap can be placed on each vserver limiting the amount of excess capacity that it is able to use. By default, the rate of the root token bucket is set at 100Mbps; each vserver is capped at 10Mbps and given a rate of 5Kbps for "fair best effort" service.

In addition to this general rate-limiting facility, htb can also be used to limit the outgoing rate for certain classes of packets that may raise alarms within the network. For instance, we are able to limit the rate of outgoing pings, or of packets containing IP options to a small number per second; this simply involves creating additional child token buckets and classifying outgoing packets so that they end up in the correct bucket. Identifying potentially troublesome packets and figuring out reasonable output rates for them is a subject of ongoing work.

CPU scheduling is implemented by the SILK kernel module running on all PlanetLab nodes. SILK's CPU scheduling infrastructure leverages Scout [24] to provide vservers with CPU guarantees and fairness. Replacing Linux's CPU scheduler was necessary because, while Linux provides approximate fairness between processes, it cannot enforce fairness between vservers or provide guarantees. PlanetLab's CPU scheduler uses a proportional sharing (PS) scheduling policy to fairly share the CPU. It incorporates the resource container [3] abstraction and maps each vserver onto a resource container that possesses some number of shares. Individual processes spawned by the vserver are all placed within the vserver's resource container. The result is that, together, the vserver's processes receive a CPU rate proportional to the vserver's shares divided by the sum of shares of all active vservers. For example, if a vserver is assigned 10 shares and the sum of shares of all active vservers (i.e., vservers that contain a runnable process) is 50, then the the vserver with 10 shares gets $10/50 = 20\%$ of the CPU.

The PS scheduling policy can provide minimum resource guarantees by capping the number of shares and using an admission controller to ensure that the cap is not exceeded. For example, limiting the number of outstanding CPU shares to 1000 means that each share is a guarantee for at least 0.1% of the CPU. Additionally, the PlanetLab CPU scheduler provides a switch to allow a vserver to proportionally share the excess capacity, or to limit it to its guaranteed rate (similar to the Nemesis scheduler []). In the previous example, the vserver with 10 shares received 20% of the CPU because it was allowed to proportionally share the excess; with this bit turned off, it would be rate-capped at $10/1000 = 1\%$ of the CPU.

4.2.3 Discussion

A trusted infrastructure service, called PlanetLab Central (PLC), is responsible for globally bootstrapping slices. PLC maintains a database of principals, slices, resource allocations, and policies on a central server. A PLC component called the *resource manager* also runs in the node manager context on each node.[1] The PLC resource man-

[1] Ideally, PLC's resource manager would run in a separate, but privileged VM on each node, since it is a trusted global infrastructure service. It is currently co-located with the node manager as an implementation expediency.

ager communicates with the central PLC server to obtain slice information and invokes the `acquire` and `bind` operations to create, initialize, and associate resources with a slice. Currently we have implemented only a web-based interface to PLC, which is described below; a programmatic interface is in the works.

PLC maintains a list of Principal Investigators (PIs) who are empowered to create slices. A PI sponsors each user of PlanetLab and assumes ultimate responsibility for his actions. To create a new slice, the PI logs into the PlanetLab Central web site and follows these steps:

- Choose a descriptive name for the slice. Slices with a common prefix can be managed together, so slices for students in the networking class at Princeton might all begin with "princeton_cs461".
- Assign users to the slice. From the standpoint of PlanetLab, a user maps to an SSH public key.
- Choose a set of machines on which the slice will run. It is assumed that a separate "resource discovery" services exist outside of PLC to aid the PI in locating machines with the desired levels of available resources and network location to include in the set.
- Choose the per-node resource reservations, proportional shares, and limits for this slice. Each PI has some number of resource "credits" available, and the resources allocated to the slice are charged against the PI's account. By default there are no limits placed on the slice and all resources are obtained "best effort".

The PI's actions do not create the slice, they only cause the PLC slice database to be updated. Every ten minutes, the PLC resource manager on each node polls the PLC server to obtain a signed, timestamped file that contains all necessary information for all slices running on that node. This information includes a list of slices that should be present on the node, resource allocation information for each slice, and policy information at the node, slice, and principal level. For each slice, the resource manager then applies relevant policies to the slice's resource allocation to obtain an appropriate rcap, which is then bound to the slice's VM. In other words, the rcaps never leave the machine; the PLC resource manager calls `acquire` and then immediately calls `bind`. The duration of each rcap obtained by the resource manager is only twenty minutes, so a currently running slice that does not appear in the most recent slice update will be reclaimed after a short time.

We also envision infrastructure services that support rich resource management policies to emerge and provide competing alternatives to PLC. One example of such a service is SHARP [15], a secure distributed resource management framework that allows agents, acting on behalf of sites, to exchange computational resources in a secure, fully decentralized fashion. In SHARP, agents peer to trade resources with peering partners using cryptographically signed statements about resources. An implementation of SHARP currently runs in its own slice (bootstrapped using PLC), and interacts with the node manager to create, bootstrap, and bind resources to virtual machines on each node. A user creates a slice using SHARP by first obtaining promises from agents at target sites, redeeming those promises for rcaps on specific nodes at those sites, and finally by asking the node manager on the specific nodes to create new virtual machines and the bind resources to them. The ease of building SHARP over the node manager interface serves as a proof of concept demonstrating the simplicity and generality of the interface.

The bottom line is that creating and handing out rcaps must be a privileged operation. The unanswered long-term question is whether a centralized service like PLC will retain this privilege, thereby having the opportunity to implement a global policy about how resources are allocated, or if the local node administrator will subsume responsibility for distributing rcaps, perhaps by entrusting a service like SHARP to distribute and acquire resources. If one views PlanetLab as analogous to an ISP, then the former seems likely. Alternatively, a fully decentralized vision of how PlanetLab will evolve suggests the latter.

4.3 Safe Raw Sockets

The PlanetLab OS provides a "safe" version of Linux raw sockets that services can use to send and receive IP packets without root privileges. These sockets are safe in two senses. First, each raw socket is bound to a particular TCP or UDP port and receives traffic only on that port; conflicts are avoided by ensuring that only one socket of any type (i.e., standard TCP/UDP or raw) is sending on a particular port. Second, outgoing packets are filtered to make sure that the local addresses in the headers match the binding of the socket. Safe raw sockets support network measurement experiments and protocol development on PlanetLab.

Safe raw sockets can also be used to monitor traffic on the node. A raw "sniffer" socket can be bound to any port that is already opened by the same VM, and this socket receives copies of all packets sent and received on that port. Additionally, privileged users can open a special administrative sniffer socket that receives copies of all outgoing packets on the machine tagged with the context ID of the sending vserver; this administrative socket is used to implement the traffic monitoring facility described in Section 4.4.

4.3.1 Interface

A standard Linux raw socket captures all incoming IP packets and allows writing of arbitrary packets to the network. In contrast, a safe raw socket is bound to a specific UDP or TCP port and receives only packets matching the protocol and port to which it is bound. Outgoing packets are filtered

to ensure that they are properly formed (e.g., the source IP and UDP/TCP port numbers are not spoofed).

Safe raw sockets use the standard Linux socket API with minor semantic differences. Just as in standard Linux, first a raw socket must be created with the `socket` system call, with the difference that it is necessary to specify `IPPROTO_TCP` or `IPPROTO_UDP` in the protocol field. Once the socket is created, it must be bound to a particular local port of the specified protocol using the standard Linux `bind` system call. At this point the socket can send and receive data. The usual `sendto, sendmsg, recvfrom, recvmsg` and `select` calls can all be used. The data received includes the IP and TCP/UDP headers, but not the link layer header. The data sent, by default, does not need to include the IP header; a service that wants to include the IP header sets the `IP_HDRINCL` socket option on the socket.

ICMP packets can also be sent and received through safe raw sockets. Each safe raw ICMP socket is bound to either a local TCP/UDP port or an ICMP identifier, depending on the type of ICMP messages the socket will receive and send. To get ICMP packets associated with a specific local TCP/UDP port (e.g., Destination Unreachable, Source Quench, Redirect, Time Exceeded, Parameter Problem), the ICMP socket needs to be bound to the specific port. To exchange ICMP messages that are not associated with a specific TCP/UDP port—e.g., Echo, Echo Reply, Timestamp, Timestamp Reply, Information Request, and Information Reply—the socket has to be bound to a specific ICMP identifier. The ICMP identifier is a 16-bit field present in the ICMP header. Only messages containing the bound identifier can be received and sent through a safe raw ICMP socket.

PlanetLab users can debug protocol implementations or applications using "sniffer" raw sockets. Vservers lack the necessary permissions to put the network card into promiscuous mode and so cannot run `tcpdump` in the standard way. A sniffer raw socket can be bound to a TCP or UDP port that was previously opened by the same user; the socket receives copies of all packets sent or received on the port but cannot send packets. A small utility called `plabdump` opens a sniffer socket and pipes the packets to `tcpdump` for parsing, so that a user can get full tcpdump-style output for any of his connections.

4.3.2 Implementation

Safe raw sockets are implemented by the SILK kernel module. SILK intercepts all incoming IP packets using Linux's `netfilter` interface and demultiplexes each to a Linux socket or to a safe raw socket. Those packets that demultiplex to a Linux socket are returned to Linux's protocol stack for further processing; those that demultiplex to a safe raw socket are placed directly in the per-socket queue maintained by SILK. When a packet is sent on a safe raw socket, SILK intercepts it by wrapping the socket's `sendmsg` function in the kernel and verifies that the addresses, protocol, and port numbers in the packet headers are correct. If the packet passes these checks, it is handed off to the Linux protocol stack via the standard raw socket `sendmsg` routine.

SILK's port manager maintains a mapping of port assignments to vservers that serves three purposes. First, it ensures that the same port is not opened simultaneously by a TCP/UDP socket and a safe raw socket (sniffer sockets excluded). To implement this, SILK must wrap the `bind`, `connect`, and `sendmsg` functions of standard TCP/UDP sockets in the kernel, so that an error can be returned if an attempt is made to bind to a local TCP or UDP port already in use by a safe raw socket. In other words, SILK's port manager must approve or deny *all* requests to bind to a port, not just those of safe raw sockets. Second, when `bind` is called on a sniffer socket, the port manager can verify that the port is either free or already opened by the vserver attempting the bind. If the port was free, then after the sniffer socket is bound to it the port is owned by that vserver and only that vserver can open a socket on that port. Third, SILK allows the node manager described in Section 4.2.1 to reserve specific ports for the use of a particular vserver. The port manager stores a mapping for the reserved port so that it is considered owned by that vserver, and all attempts by other vservers to bind to that port will fail.

4.3.3 Discussion

The driving application for safe raw sockets has been the Scriptroute [29] network measurement service. Scriptroute provides users with the ability to execute measurement scripts that send arbitrary IP packets, and was originally written to use privileged raw sockets. For example, Scriptroute implements its own versions of `ping` and `traceroute`, and so needs to send ICMP packets and UDP packets with the IP TTL field set. Scriptroute also requires the ability to generate TCP SYN packets containing data to perform `sprobe`-style bottleneck bandwidth estimation. Safe raw sockets allowed Scriptroute to be quickly ported to PlanetLab by simply adding a few calls to `bind`. Other users of safe raw sockets are the modified versions of `traceroute` and `ping` that run in a vserver (on Linux, these utilities typically run with root privileges in order to open a raw socket). Additionally, several groups have already implemented TCP in user-space using safe raw sockets, or are in the process of doing so.

Safe raw sockets also make it facilitate user-level protocol stacks. This allows a slice to utilize, for example, a variant of TCP that is tuned for high-bandwidth pipes [**?**] or packet re-ordering that might occur when data is stripped across multiple overlay paths [**?**]. A BSD-based TCP library currently exists and runs on PlanetLab.

Safe raw sockets are just one example of how PlanetLab services need to be able to share certain address spaces.

Another emerging example is that slices that want to control IP tunneling want to customize the routing table so as to override default routes for its own packets. Yet another example is the need to share access to well-known ports; e.g., multiple services want to run a DNS server. In both cases, we are adopting an approach similar to that used for raw sockets: partition the address space by doing early demultiplexing at a low level in the kernel. No single service is granted privileged and exclusive access to the address space.

Note that an alternative to sharing and partitioning a single space among all virtual machines is to *contextualize* it—that is, we could present each VM with its own local version of the space by moving the demultiplexing to another level. For instance, we could assign a separate IP address to each VM and allow each to use the entire port space and manage its own routing table. In fact, vservers already support this capability. The problem is that we simply do not have enough IPv4 addresses available to assign on the order of 1000 to each node.

4.4 Monitoring

Good monitoring tools are clearly required to support a distributed infrastructure such as PlanetLab, which runs on hundreds of machines worldwide, and host dozens of network services that use and interact with each other and the Internet in complex and unpredictable ways. Managing this infrastructure—collecting, storing, propagating, aggregating, discovering, and reacting to observations about the system's current conditions—is one of the most difficult challenges facing PlanetLab.

Consistent with the principle of unbundled management, we have defined a low-level *sensor interface* for uniformly exporting data from the underlying OS and network, as well as from individual services. Data exported from a sensor can be as simple as the process load average on a node or as complex as a peering map of autonomous systems obtained from the local BGP tables. That is, sensors encapsulate raw observations that already exist in many different forms, and provide a shared interface to alternative monitoring services.

Although the long-term goal is for these monitoring services to detect, reason about, and react to anomalous behavior before it becomes disruptive, PlanetLab has an immediate need of responding to disruptions after the fact. Frequently within the past year, traffic generated by PlanetLab researchers has caught the attention of ISPs, academic institutions, Web sites, and sometimes even home users. In nearly all cases, the problems have stemmed from naive service design and analysis, programmer errors, or hypersensitive intrusion detection systems. Examples include network mapping experiments that probe large numbers of random IP addresses (looks like a scanning worm), services aggressively `traceroute`'ing to certain target sites on different ports (looks like a portscan), services performing distributed measurement to a target site (looks like a DDoS attack), services sending large numbers of ICMP packets (not a bandwidth problem, but renders low-end routers unusable), and so on. Addressing such complaints requires an *auditing* tool that can map an incident onto a responsible party.

4.4.1 Interface

A sensor provides a particular kind of information that is available (or can be derived) on on a local node. A *sensor server* aggregates several sensors at a single access point, thereby providing controlled sharing of sensors among many clients (e.g., monitoring services). To obtain a sensor reading, a client makes a request to a sensor server. Each sensor outputs one or more tuples of untyped data values. Every tuple from a sensor conforms to the same schema. Thus, a sensor can be thought of as providing access to a (potentially infinite) database table.

Sensor semantics are divided into two types: *snapshot* and *streaming*. Snapshot sensors maintain a finite size table of tuples, and immediately return the table (or, conceivably, some subset of it) when queried for it. This can range from a single tuple which rarely varies (e.g. "number of processors on this machine") to a circular buffer that is constantly updated, of which a snapshot is available to clients (for instance, "the times of 100 most recent connect system calls, together with the associated slices"). Streaming sensors follow an event model, and deliver their data asynchronously, a tuple at a time, as it becomes available. A client connects to a streaming sensor and receives tuples until either it or the sensor server closes the connection.

More precisely, a sensor server is an HTTP [13] compliant server implementing a compliant subset of the specification (GET and HEAD methods only) listening to requests from `localhost` on a particular port. Requests come in the form of uniform resource identifiers (URIs) in GET methods. For example, the URI:

```
http://localhost:33080/nodes/ip/name
```

is a request to the sensor named "`nodes`" at the sensor server listening on port 33080. The portion of the URI after the sensor name (i.e., `ip/name`) is interpreted by the sensor. In this case, the nodes sensor returns a comma-separated lines containing the IP address and DNS name of each registered PlanetLab node. We selected HTTP as the sensor server protocol because it is a straight-forward, well understood, and well supported protocol. The primary format for the data returned by the sensor is a text file containing easy-to-parse comma separated values.

4.4.2 Implementation

An assortment of sensor servers have been implemented to date, all of which consist of a stripped-down HTTP server

encapsulating an existing source of information. For example, one sensor server reports various information about kernel activities. The various sensors exported by this server are essentially wrappers around the `/proc` file system. Example sensors include `meminfo` (returns information about current memory usage), `load` (returns 1-minute load average), `load5` (returns 5-minute load average), `uptime` (returns uptime of the node in seconds), and `bandwidth(slice)` (returns the bandwidth consumed by a `slice`, given by a slice id).

These examples are simple in at least two respects. First, they require virtually no processing; they simply parse and filter values already available in `/proc`. Second, they neither stream information nor do they maintain any history. One could easily imagine a variant of `bandwidth`, for example, that both streams the bandwidth consumed by the slice over that last 5 minute period, updated once every five minutes, or returns a table of the last *n* readings it had made.

Another example sensor server reports information about how the local host is connected to the Internet, including path information returned by `traceroute`, peering relationships determined by a local BGP feed, and latency information returned by `ping`. This sensor server illustrates how sensors sometimes require more complex and expensive implementations; some send and receive messages over the Internet before they can respond, and some cache the results of earlier invocations.

4.4.3 Discussion

Using the sensor abstraction, and an emerging collection of sensor implementations, an assortment of monitoring services are being deployed. Many of these services are modeled as distributed query processors, including PIER [**?**], Sophia [**?**], and IrisNet [**?**]. For example, Sophia allows users to make declarative statements about PlanetLab's state in a Prolog-like programming language. To illustrate, the following shows a rule declaration followed by its evaluation. The rule defines a `slice_bandwidth` expression to be the sum of slice bandwidths on all the nodes within the last 30 seconds. Evaluating this rule determines the total bandwidth consumed (`SliceBandwidth`) by a particular slice (`slice47`) across all PlanetLab nodes.

```
slice_bandwidth(Slice, Bandwidth) :-
     forall( [Node, BwVar],
             (node(Node),
              bandwidth(env(node(Node),
                            time(Time),
                            (Time>now-30)),
                        slice(Slice),
                        BwVar) ),
              Result),
     sumlist(Result, Bandwidth).

eval( slice_bandwidth( slice47,
                       SliceBandwidth) ).
```

The terms in this expression—e.g., `bandwidth()`—are Prolog wrappers for one of the available sensors on each node. In effect, the sensors server as I/O for Sophia.

Again turning from longer term efforts to short-term problems, it has been necessary to build an auditing service on top of the available sensor information. The main goal of this service is to minimize the amount of time the PlanetLab support staff has to spend responding to each incident report. Ultimately, the hope is to remove the support team entirely from the process, enabling the victim to directly complain to the responsible research group.

Specifically, a traffic auditing service runs on every PlanetLab node, snooping all outgoing traffic using a special raw socket provided by the SILK module that tags each packet with the ID of the vserver that sent it. From each packet, the auditing service logs the time it was sent, the IP source and destination, protocol, port numbers, and TCP flags if applicable. It then generates Web pages on each node that summarize the traffic sent in the last hour by IP destination and slice name. The hope is that an administrator at a site that receives questionable packets from a PlanetLab machine will type the machine's name or IP address into his browser, find the audit-generated pages, and use them to contact the experimenters about the traffic. For example, an admin who clicks on an IP address in the destination summary page is shown all of the PlanetLab accounts that sent a packet to that address within the last hour, and provided with links to send email to the researchers associated with these accounts. Another link directs the admin to the network traffic database at `www.planet-lab.org` where back logs are archived, so that he can make queries about the origin of traffic sent earlier than one hour ago.

So far, our experience with the traffic auditing service has been mixed. On the one hand, the PlanetLab support team has found it very useful for responding to traffic queries; after receiving a complaint, they use the Web interface to identify the responsible party and forward the complaint on to them. As a result, there has been a reduction in overall incident response time and the time invested by support staff per incident. On the other hand, many external site administrators either don't find the web page or choose not to use it. For example, when receiving a strange packet from `planetlab-1.cs.princeton.edu`, most sites respond by sending email to `abuse@princeton.edu`; by the time the support team receives the complaint, it has bounced through several levels of university administration. We may be able to avoid this indirection by providing reverse DNS mappings for all nodes to *nodename*`.planet-lab.org`, but this requires effort from each site that sponsors PlanetLab nodes. Finding mechanisms that further decentralize the problem-response process is ongoing work.

Finally, although our experience to date has involved implementing and querying read-only sensors that can

be safely accessed by untrusted monitoring services, one could easily imagine PlanetLab also supporting a set of *actuators* that only trusted management services could use to control PlanetLab. For example, there might be an actuator that terminates a slice, such that a Sophia expression can be written to kill a slice that has violated global bandwidth consumption limits. Today, slice termination is not exposed as an actuator, but is implemented in the node manager and can be invoked only by the trusted PLC service, or an authenticated network operator that remotely logs into the node manager.

5 Evaluation

This section evaluates three aspects of slice creation and initialization.

5.1 Vserver Scalability

The scalability of vservers is primarily determined by disk space for vserver root filesystems and service-specific storage. On PlanetLab, each vserver is created with a root filesystem that points back to a trimmed-down reference root filesystem which comprises 1408 directories and 28003 files covering 508 MB of disk. Using vserver's primitive COW on all files, excluding those in `/etc` and `/var`, each vserver root filesystem mirrors the reference root filesystem while only requiring 29 MB of disk space, 5.7% of the original root filesystem size. This 29 MB consists of 17.5 MB for a copy of `/var`, 5.6 MB for a copy of `/etc`, and 5.9 MB to create 1408 directories (4 KB per directory). Given the reduction in vserver disk footprints afforded by COW, we have been able to create 1,000 vservers on a single PlanetLab node. In the future, we would like to push disk space sharing even further by using a true filesystem COW and applying techniques from systems such as the Windows Single Instance Store [6].

Kernel resource limits are a secondary factor in the scalability of vservers. While each vserver is provided with the illusion of its virtual execution environment, there still remains a single copy of the underlying operating system and associated kernel resources. Under heavy degrees of concurrent vserver activity, it is possible that limits on kernel resources may become exposed and consequently limit system scalability. (We have already observed this with file descriptors.) The nature of such limits, however, are no different from that of large degrees of concurrency or resource usage within a single vserver or even on an unmodified Linux kernel. In both cases, one solution is to simply extend kernel resource limits by recompiling the kernel. Of course, simple scaling up of kernel resources may be insufficient if inefficient algorithms are employed within the kernel (e.g., $O(n)$ searches on linked lists). Thus far, we have yet to run into these types of algorithmic bottlenecks.

5.2 Slice Creation

This section reports how long it takes the node manager to create a vserver on a single node. The current implementation of PLC has each node poll for slice creation instructions every 10 minutes, but this is an artifact of piggybacking the slice creation mechanism on existing software update machinery. The more interesting question is how heavy-weight the per-node aspect of slice creation is, assuming an efficient event propagation service.

To create a new slice on a specific node, a slice creation service must complete the following steps at that node:

1. the slice creation service contacts a port mapping service to find the port where the node manager's XML-RPC server is listening for requests;

2. the slice creation service performs a node manager `acquire` RPC to obtain a ticket for immediate rights to a vserver and best-effort resource usage;

3. the slice creation service performs a node manager `bind` RPC to bind the ticket to a new slice name;

4. the node manager, after completing the RPCs, creates the new vserver and notifies the necessary resource schedulers to effect the newly added resource bindings for the new slice; and

5. the node manager first calls `vadduser` to instantiate the vserver and then calls `vserver-init` start execution of software within the new vserver.

Running on a 1.2GHz Pentium, the first three steps complete in 0.15 seconds, on average. How long the fourth and fifth steps takes depends on how the user wants to initialize the slice. At a minimum, the vserver creation and initialization takes an additional 9.66 seconds on average. However, this does not include the time to load an initialize any service software such as SSHD or other packages. It also assumes a hit in a warm cache of vservers. Creating a new vserver from scratch takes over a minute.

5.3 Service Initialization

This section uses an example service—Sophia—to demonstrate how long it takes a service to initialize a service in a slice once the slice exists.

Sophia's basic slice is managed by combination of RPM, apt-get and custom slice tools. When a Sophia slice is created, it must be loaded with the appropriate environment. This is accomplished by executing a boot URL script inside each vserver. The boot URL script downloads and installs apt-get tools and a root Sophia slice RPM, and starts an update process. The update process periodically, using the apt-get tool, downloads the tree of current packages specific for the Sophia slice. If a newer package based on

RPM hierarchy is found, it and it's dependencies are download and installed. With this mechanism, the new version of packages, are not directly pushed to all the nodes, but are published in the Sophia packages tree. The slice's update mechanism then polls (potentially followed with an action request push) the package tree and performs the upgrade actions.

In the current setting, on average it takes 11.2 seconds to perform an empty update on a node, that is, to download the package tree, but not find anything new to upgrade. When a new Sophia "core" package is found and it needs to be upgraded, the time increases to 25.9 seconds per node. These operations occur in parallel, so the slice upgrade time is not bound by the sum of node update times. However, the slice is to be considered upgraded only when all of its active nodes are finished upgrading. When run on the 180 nodes in the current system, the average update time (corresponding to the slowest node) is 228.0 seconds.

The performance could be much improved, for example, by use of a better distribution mechanism such as a CDN, Bullet, or Bittorent. Also using a faster alternative to the RPM package dependencies system could improve the locally performed dependencies checks.

From these sample slice boot/update times, it can be concluded that the current slice architecture is better suited for services of medium to long lifetime and not for short lifetime services on the order of a network connection.

6 Conclusions

Based on experience providing the network research community with a testbed for planetary-scale services, the PlanetLab OS has evolved a set of mechanisms to support distributed virtualization and unbundled management. The design allows network services to run in a slice of PlanetLab's global resources, with the PlanetLab OS providing only local (per-node) abstractions. As much global (network-wide) functionality as possible pushed onto infrastructure services running in their own slices. Only slice creation (coupled with resource allocation) and slice termination run as a global privileged service, but in the long-term.

References

[1] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[2] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proceedings of the 1st International Conference on Pervasive Computing (Pervasive 2002)*, pages 195–210, August 2002.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 45–58, New Orleans, Louisiana, 1999.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, Lake George, New York, October 2003.

[5] R. Black, P. Barham, A. Donnelly, and N. Stratford. Protocol Implementation in a Vertically Structured Operating System. In *Proceedings of IEEE Conference on Computer Networks*, November 1997.

[6] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24, Seattle, Washington, August 2000.

[7] Y. Chu, S. Rao, and H. Zhang. A Case For End System Multicast. In *Proceedings of the ACM SIGMETRICS 2000 Conference*, pages 1–12, Santa Clara, California, June 2000.

[8] B. Chun, J. Lee, and H. Weatherspoon. Netbait: a Distributed Worm Detection Service, 2002. `http://netbait.planet-lab.org/`.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–215, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[10] J. Dike. User-mode Linux. In *Proceedings of the 5th Annual Linux Showcase and Conference*, Oakland, CA, November 2001.

[11] D. R. Engler and M. F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of the ACM SIGCOMM '96 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 53–59, Stanford, California, August 1996.

[12] Ensim Corp. Ensim Virtual Private Server. `http://www.ensim.com/products/materials/datasheet_vps_051003.pdf`, 2000.

[13] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1; RFC 2616. *Internet Request for Comments*, June 1999.

[14] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. `http://www.gridforum.org/ogsi-wg/drafts/ogsa_draft2.9_2002-06-22.pdf`, June 2002. draft 2.9.

[15] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. Sharp: An architecture for secure resource peering. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[16] N. Hardy. The KeyKOS Architecture. *Operating Systems Review*, 19(4):8–25, October 1985.

[17] Jacques Gelinas. Linux vservers Project. `http://www.solucorp.qc.ca/miscprj/s_context.hc`.

[18] P.-H. Kamp and R. N. M. Watson. Jails: Confining the Omnipotent Root. In *Proceedings of the 2nd International SANE Conference*, May 2000.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, Cambridge, Massachusetts, November 2000.

[20] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

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

[22] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[23] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *International Conference on Multimedia Computing and Systems*, pages 90–99, 1994.

[24] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 153–167, Seattle, Washington, October 1996.

[25] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of HotNets–I*, Princeton, New Jersey, October 2002.

[26] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proceedings of the IEEE INFOCOM 2002 Conference*, pages 1190–1199, New York City, June 2002.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pages 329–350, Heidelberg, Germany, November 2001.

[28] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-Scale Persistent Peer-to-Peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 188–201, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[29] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A facility for distributed internet measurement. In *Proceedings of the 4th USENIX Symposium on Internet Technologies (USITS '03)*, Seattle, Washington, March 2003.

[30] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Peer-to-Peer Lookup Service for Internet Applications. pages 149–160, August 2001.

[31] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: Offering Internet QoS Using Overlays. In *Proceedings of HotNets–I*, Princeton, New Jersey, October 2002.

[32] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *Proceedings of the Fifth USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 345–360, Boston, Massachusetts, December 2002.

[33] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the Fifth USENIX Symposium on Operating System Design and Implementation (OSDI)*, Boston, Massachusetts, December 2002.

[34] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Fifth USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 255–270, Boston, Massachusetts, December 2002.