USENIX Association

Proceedings of the First Symposium on Networked Systems Design and Implementation

San Francisco, CA, USA
March 29–31, 2004

Operating System Support for Planetary-Scale Network Services

Andy Bavier* Mic Bowman† Brent Chun† David Culler‡ Scott Karlin* Steve Muir*
Larry Peterson* Timothy Roscoe† Tammo Spalink* Mike Wawrzoniak*

*Department of Computer Science, Princeton University
†Intel Research
‡Computer Science Division, University of California, Berkeley

Abstract

PlanetLab is a geographically distributed overlay network designed to support the deployment and evaluation 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 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 platform designed to support the deployment and evaluation of planetary-scale network services [30]. It currently includes over 350 machines spanning 150 sites and 20 countries. It supports over 450 research projects focused on a wide range of services, including file sharing and network-embedded storage [11, 22, 35], content distribution networks [39], routing and multicast overlays [1, 8], QoS overlays [38], scalable object location services [2, 33, 34, 37], anomaly detection mechanisms [9], and network measurement tools [36].

As a distributed system, PlanetLab is characterized by a unique set of relationships between principals—e.g., users, administrators, researchers, service providers—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. Researchers must have access to the platform, which implies a distributed set of machines that must be shared 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 on it. Not only must each of 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 design is largely the product of our experience having hundreds of users stressing PlanetLab since the platform became operational in July 2002.

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). Virtualization and virtual machines are, of course, well-established concepts. 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 provide 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. However, much of a slice's behavior is left unspecified in the architecture. This includes exactly how a slice is created, which we discuss in the context of unbundled management, as well as the programming environment PlanetLab provides. Giving slices as much latitude as possible in defining a suitable environment means, for example, that the PlanetLab OS does not provide tunnels that connect the constituent VMs into any particular overlay configuration, but instead provides an interface that allows each service to define its own topology on top of the fully-connected Internet. Similarly, PlanetLab does not prescribe a single language or runtime system, but instead allows slices to load whatever environments or software packages they need.[1]

2.1.1 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, etc.) to prevent a slice interfering with another, or gaining access to information in another slice. In many cases, this partitioning and contextualizing 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.

Resource scheduling and VM isolation were recognized as important issues from the start, but the expectation was that a "best effort" solution would be sufficient for some time. Our experience, however, is that excessive loads (especially near conference deadlines) and volatile performance behavior (due to insufficient isolation) were the dominant problems in early versions of the system. The lack of isolation has also led to significant management overhead, as human intervention is required to deal with run-away processes, unbounded log files, and so on.

2.1.2 Isolating PlanetLab

The PlanetLab OS must also protect the outside world from slices. PlanetLab nodes are simply machines connected to the Internet, and 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 and its goal of supporting novel network services, 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, both 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

[1] This is not strictly true, as PlanetLab currently provides a Unix API at the lowest level. Our long-term goal, however, is to decouple those aspects of the API that are unique to PlanetLab from the underlying programming environment.

to slices after the fact. This concern about how users (or their services) affect the outside world is a novel requirement for PlanetLab, unlike traditional timesharing systems, where the interactions between users and unsuspecting outside entities is inherently rare.

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 be just as significant an issue as securing the OS to prevent malicious users from hijacking machines. For example, a single PlanetLab user running TCP throughput experiments on U.C. 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; in particular, this includes the services required to manage a global platform such as PlanetLab. 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 (e.g., slice creation, resource and topology discovery, 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 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.

- 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.

Of particular note, *slice creation* is itself implemented as a service running in its own slice, which leads to the following additional requirement on the PlanetLab OS:

- 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 "bootstrapping" slice creation service to create initial slices, including the slices that other slice creation services run in.

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 heavy-weight 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.

The bottom line is that OS 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) that requires extra privilege or access but is not part of the kernel. In addition, there is a third aspect to the problem that 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 bootstrapping slice creation). The PlanetLab OS strives to minimize the latter, but there remains a core of non-kernel functionality that has to be unique on a node.

2.3 Evolving Architecture

While unbundled management addresses the challenge of evolving PlanetLab as a whole, there remains the very practi-

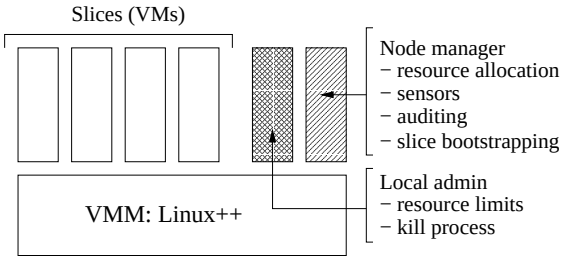

Figure 1: PlanetLab Node Architecture

cal issue of evolving the underlying OS that supports unbundled management.

Simply stated, 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, and in any case without first gaining some experience, no one could fully understand what such a system 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 system rather than a minimalist API tuned for someone else's OS research agenda.

This suggested the strategy of starting with a full-featured OS—we elected to use Linux due to its widespread use in the research community—and incrementally transforming it based on experience. This evolution is guided by the "meta" architecture depicted in Figure 1.

At the lowest level, each PlanetLab node runs a *virtual machine monitor* (VMM) that implements and isolates VMs. The VMM also defines the API to which services are implemented. PlanetLab currently implements the VMM as a combination of the Linux kernel and a set of kernel extensions, as outlined in Section 4.

A privileged, "root" VM running on top of the VMM, called the *node manager*, monitors and manages all the VMs on the node. Generally speaking, the node manager enforces policies on creating VMs and allocating resources to them, with services interacting with the node manager to create new VMs rather than directly calling the VMM. Moreover, all interactions with the node manager are local: only services running in some other VM on the node are allowed to call the node manager, meaning that remote access to a specific node manager is always indirect through one of the services running on the node. Today, most policy is hard-coded into the node manager, but we expect that local administrators will eventually be able to configure the policies on their own nodes. (This is the purpose of the local administrator VM shown in Figure 1.)

A subset of the services (slices) running on top of the VMM can be characterized as *privileged* in some way: they are allowed to make privileged calls to the node manager (e.g., to allocate local resources to a VM). We expect all slices that provide a service to end-users to be unprivileged, while some infrastructure services may need to run in a privileged slice. To date, three types of infrastructure services are emerging: (1) brokerage services that are used to acquire resources and create slices that are bound to them, (2) environment services that are used to initialize and maintain a slice's code base, and (3) monitoring services that are used to both discover the available resources and monitor the health of running services.

Because we expect new facilities to be incorporated into the architecture over time, the key question is where any new functionality should be implemented: in an unprivileged slice, in a privileged slice, in the node manager, or in the VMM? Such decisions are guided by the following two principles:

- Each function should be implemented at the "highest" level possible, that is, running a service in a slice with limited privileged capabilities is preferred to a slice with widespread privileges, which in turn is preferred to augmenting the node manager, all of which are preferable to adding the function to the VMM.

- Privileged slices should be granted the minimal privileges necessary to support the desired behavior. They should not be granted blanket superuser privileges.

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 discussed in the previous section. This section discusses how PlanetLab's requirements recommend certain approaches over others, and in the process, discusses related work.

3.1 Node Virtualization

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 as provide stronger 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 [41]. 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 [14], and projects such as User Mode Linux [12], BSD's Jail [21], and Linux vservers [25]. Such high-level virtualization adequately supports PlanetLab's goals of supporting large numbers of overlay services, while providing reasonable assurances of isolation.

3.2 Isolation and Resource Allocation

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 [18], 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 as in Processor Capacity Reserves [27], scheduling along data paths as in Scout [28], or entirely restructuring the OS to eliminate server processes in the data path as in Nemesis [23]. 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.

The problem of distributed coordination of resources, in turn, has been explored in the context of Condor [26] and more recently the Open Grid Services Architecture [16]. 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 [42] have been implemented above PlanetLab, as have more service-oriented frameworks like SHARP [17].

3.3 Network Virtualization

Having settled on node virtualization at the system call level, the third challenge is how to virtualize 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 [6, 13]. The PlanetLab OS uses a similar approach, providing shared network access using a "safe" version of the raw socket interface.

Exokernels have traditionally either provided the raw (physical) device to a single library OS to manage, or controlled sharing of the raw device between library OSes based on connections. However, in PlanetLab the kernel must take responsibility for sharing raw access (both reception and transmission of potentially arbitrary packets) among multiple competing services in a controlled manner according to some administrative policy. Additionally, it must protect the surrounding network from buggy and malicious services, an issue typically ignored by existing systems.

An alternative to sharing and partitioning a single network address 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 different IP address to each VM and allow each to use the entire port space and manage its own routing table. The problem is that we simply do not have enough IPv4 addresses available to assign on the order of 1000 to each node.

3.4 Monitoring

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 upon 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 [5, 32] 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 the network, and system monitoring.

4.1 Node Virtualization

A slice corresponds to a distributed set of virtual machines. Each VM, in turn, is implemented as a *vserver* [25]. The vserver mechanism is a patch to the Linux 2.4 kernel that provides multiple, independently managed virtual servers running on a single machine. Vservers are the principal mechanism in PlanetLab for providing virtualization on a single node, and contextualization of name spaces; e.g., user identifiers and files.

As well as providing security between slices sharing a node, vservers provide a limited root privilege that allows a slice to customize its 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.

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 the files, processes, SysV IPC, network interfaces, and accounts that can be named in their containing vserver, and are otherwise denied access to system-wide operating system resources. Each vserver is given its own UID/GID namespace, along with a weaker form of `root` that provides a local superuser without 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 can modify the vserver's root filesystem, allowing users to customize the installed software packages for their vserver. Combined with per-vserver UID/GID namespaces, it allows vservers to implement their own account management schemes (e.g., by maintaining a per-vserver `/etc/passwd` and running `sshd` on a different TCP port), thereby providing the basis for integration with other wide-area testbeds such as NetBed [42] and RON [1].

A vserver is initialized with two pieces of persistent state: a set of SSH keys and a vserver-specific `rc.vinit` file. The former allow the owners of the slice to SSH into the vserver, while the latter serves as a boot script that gets executed each time the vserver starts running.

Vservers communicate with one another via IP, and not local sockets or other IPC functions. This strong separation between slices simplifies resource management and isolation between vservers, since the interaction between two vservers is independent of their locations. However, the namespace of network addresses (IP address and port numbers) is not contextualized: this would imply either an IP address for each vserver, or hiding each vserver behind a per-node NAT. We rejected both these options in favor of slices sharing port numbers and addresses on a single node.

4.1.2 Implementation

Each vserver on a machine is assigned a unique *security context*, and each process 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 the process's descendants. Isolation between vservers is enforced at the system call interface by using a combination of security context and UID/GID to check access control privileges and decide what information should be exposed to a given process. All of these mechanisms are implemented in the baseline vserver patch to the kernel. We have implemented several utilities to simplify creating and destroying vservers, and to transparently redirect a user into the vserver for his or her specific slice using SSH.

On PlanetLab, our utilities initialize a vserver by creating a mirror of a reference root filesystem inside the vserver using hard links and the "immutable" and "immutable invert" filesystem bits. Next we create two Linux accounts with login name equal to the slice name, one in the node's primary vserver and one in the vserver just created, and sharing a single UID. The default shell for the account in the main vserver is set to `/bin/vsh`, a modified `bash` shell that performs the following four actions upon login: switching to the slice's vserver security context, `chroot`ing to the vserver's root filesystem, relinquishing a subset of the true superuser's capabilities, and redirecting into the other account inside the vserver. The 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 substantial sharing of physical memory and disk space, with no active state needed for idle vservers. For physical memory, savings are accrued by having single copies of the kernel and daemons, and shared read-only and copy-

on-write memory segments across unrelated vservers. Disk space sharing is achieved using the filesystem immutable invert bit which allows for a primitive form of copy-on-write (COW). Using COW on chrooted vserver root filesystems, vserver disk footprints are just 5.7% of that required with full copies (Section 5.1). Comparable sharing in a virtual machine monitor or isolation kernel is strictly harder, although with different isolation guarantees.

PlanetLab's application of vservers makes extensive use of the Linux capability mechanism. Capabilities determine whether privileged operations such as pinning physical memory or rebooting are allowed. In general, the vserver `root` account is denied all capabilities that could undermine the security of the machine (e.g., accessing raw devices) and granted all other capabilities. However, as discussed in Section 2.3, each PlanetLab node supports two special contexts with additional capabilities: the *node manager* and the *admin slice*.

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 admin slice provides weaker privileges to site administrators, giving them a set of tools to manage the nodes without providing full `root` access. The admin context has a complete view of the machine and can, for example, cap the node's total outgoing bandwidth rate, kill arbitrary processes, and run `tcpdump` to monitor traffic on the local network.

4.1.3 Discussion

Virtualizing above the kernel has a cost: weaker guarantees on isolation and 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 a Linux kernel vulnerability might gain control of the operating system, and hence compromise security of the machine. We have observed such an incident, in which a subset of PlanetLab nodes were compromised in this way. This would have been less likely using a lower-level VM monitor. Another potential cost incurred by virtualizing above the kernel is QoS crosstalk. Eliminating all QoS crosstalk (e.g., interactions through the buffer cache) is strictly harder with vservers. As described in the next section, however, fairly deep isolation can be achieved.

The combination of vservers and capabilities provides more flexibility in access control than we currently use in PlanetLab. For example, sensor slices (see section 4.4) could be granted access to information sources that cannot otherwise easily be shared among clients. As we gain experience on the privileges services actually require, extending the set of Linux capabilities is a natural path toward exposing privileged operations in a controlled way.

4.2 Isolation and Resource Allocation

A key feature of slices is the isolation they provide between services. Early experience with PlanetLab illustrates the need for slice isolation. For example, we have seen slices acquire all available file descriptors on several nodes, preventing other slices from using the disk or network; routinely fill all available disk capacity with unbounded event logging; and consume 100% of the CPU on 50 nodes by running an infinite loop. Isolating slices is necessary to make the platform useful.

The node manager provides a low-level interface for obtaining resources on a node and binding them to a local VM that belongs to some slice. The node manager does not make any policy decisions regarding how resources are allocated, nor is it remotely accessible. Instead, a bootstrap brokerage service running in a privileged slice implements the resource allocation policy. This policy includes how many resources to allocate to slices that run other brokerage services; such services are then free to redistribute those resources to still other slices. Note that resource allocation is largely controlled by a central policy in the current system, although we expect it to eventually be defined by the node owner's local administrator. Today, the only resource-related policy set by the local node administrator is an upper bound on the total outgoing bandwidth that may be consumed by the node.

4.2.1 Interface

The node manager denotes the right to use a set of node resources as a *resource capability* (rcap)—a 128-bit opaque value, the knowledge of which provides access to the associated resources. The node manager provides privileged slices with the following operation to create a resource capability:

$$\text{rcap} \leftarrow \text{acquire}(\text{rspec})$$

This operation takes a *resource specification* (rspec) as an argument, and returns an rcap should there by sufficient resources on the node to satisfy the rspec. Each node manager tracks both the set of resources available on the node, and the mapping between committed resources and the corresponding rcaps; i.e., the rcap serves as an index into a table of rspecs.

The rspec describes a slice's privileges and resource reservations over time. Each rspec consists of a list of reservations for physical resources (e.g., CPU cycles, link bandwidth, disk capacity), limits on logical resource usage (e.g., file descriptors), assignments of shared name spaces (e.g., TCP and UDP port numbers), and other slice privileges (e.g., the right to create a virtual machine on the node). The rspec also specifies the start and end times of the interval over which these values apply.

Once acquired, rcaps can be passed from one service to another. The resources associated with the rcap are bound

to a virtual machine (vserver) at slice creation time using the following operation:

bind(slice_name, rcap)

This operation takes a slice identifier and an rcap as arguments, and assuming the rspec associated with the rcap includes the right to create a virtual machine, creates the vserver and binds the resources described by the rcap to it. Note that the node manager supports other operations to manipulate rcaps, as described more fully elsewhere [10].

4.2.2 Implementation

Non-renewable resources, such as memory pages, disk space, and file descriptors, are 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.

For renewable resources such as CPU cycles and link bandwidth, the OS supports two approaches to providing isolation: *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 (e.g., 1Mbps of link bandwidth). PlanetLab provides CPU and bandwidth guarantees for slices that request them, and "fair best effort" service for the rest. In addition to isolating slices from each other, 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) [24] is used to cap the total outgoing bandwidth of a node, cap per-vserver output, and to provide bandwidth guarantees and fair service among 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 (as well as 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 determining reasonable output rates for them is a subject of ongoing work.

CPU scheduling is implemented by the SILK kernel module, which leverages Scout [28] to provide vservers with CPU guarantees and fairness. Replacing Linux's CPU scheduler was necessary because, while Linux provides approximate fairness between individual processes, it cannot enforce fairness between vservers; nor can it 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 the vserver's set of processes receives 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 vserver with 10 shares gets $10/50 = 20\%$ of the CPU.

The PS scheduling policy is also used to provide minimum cycle guarantees by capping the number of shares and using an admission controller to ensure that the cap is not exceeded. The current policy limits the number of outstanding CPU shares to 1000, meaning 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 [23]). 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

We have implemented a centrally-controlled brokerage service, called PlanetLab Central (PLC), that is responsible for globally creating slices [31]. PLC maintains a database of principals, slices, resource allocations, and policies on a central server. It exports an interface that includes operations to create and delete slices; specify a boot script, set of user keys, and resources to be associated with the slice; and instantiate a slice on a set of nodes. A per-node component of PLC, called a *resource manager*, runs in a privileged slice; it is allowed to call the acquire operation on each node. The resource

manager on each node periodically communicates with the central PLC server to obtain policy about what slices can be created and how many resources are to be bound to each.

PLC allocates resources to participating institutions and their projects, but it also allocates resources to other brokerage services for redistribution. In this way, PLC serves as a bootstrap brokerage service, where we expect more sophisticated (and more decentralized) services to evolve over time. We envision this evolution proceeding along two dimensions.

First, PLC currently allows the resources bound to a slice to be specified by a simple (share, duration) pair, rather than exposing the more detailed rspec accepted by the node manager. The share specifies a relative share of each node's CPU and link capacity that the slice may consume, while duration indicates the period of time for which this allocation is valid. PLC policy specifies how to translate the share value given by a user into a valid rspec presented to the node manager. Over time we expect PLC to expose more of the rspec structure directly to users, imposing less policy on resource allocation decisions. We decided to initially hide the node manager interface because its semantics are not well-defined at this point: how to divide resources into allocatable units is an open problem, and to compensate for this difficulty, the fields of the rspec are meaningful only to the individual schedulers on each node.

Second, there is significant interest in developing market-based brokerage services that establish resource value based on demand, and a site's purchasing capacity based on the resources it contributes. PLC currently has a simple model in which each site receives an equal number of shares that it redistributes to projects at that site. PLC also allocates shares directly to certain infrastructure services, including experimental brokerage services that are allowed to redistribute them to other services. In the long term, we envision each site administrator deciding to employ different, or possibly even multiple, brokerage services, giving each some fraction of its total capacity. (In effect, site administrators currently allocate 100% of their resources to PLC by default.)

To date, PLC supports two additional brokerage services: Emulab and SHARP. Emulab supports short-lasting network experiments by pooling a set of PlanetLab resources and establishing a batch queue that slices are able to use to serialize their access. This is useful during times of heavy demand, such as before conference deadlines. SHARP [17] is 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.

4.3 Network Virtualization

The PlanetLab OS supports network virtualization by providing 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 respects. 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 within a slice. A "sniffer" socket can be bound to any port that is already opened by the same VM, and this socket receives copies of all packet headers sent and received on that port. Additionally, sufficiently privileged slices 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 well-formed and that 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. It must then be bound to a particular local port of the specified protocol using the Linux `bind` system call. At this point the socket can send and receive data using the usual `sendto`, `sendmsg`, `recvfrom`, `recvmsg`, and `select` calls. 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 slice 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 (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. Most slices lack the necessary capability 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 in the same vserver; the socket receives copies of all packets sent or received on the port but cannot send packets. A 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 or her connections.

4.3.2 Implementation

Safe raw sockets are implemented by the SILK kernel module, which 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 [36] 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). Safe raw sockets have also been used to implement user-level protocol stacks, such as variants of TCP tuned for high-bandwidth pipes [20], or packet re-ordering when striping across multiple overlay paths [43]. A BSD-based TCP library currently 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 some slices want to customize the routing table so as to control IP tunneling for their own packets. Yet another example is the need to share access to well-known ports; e.g., multiple services want to run DNS servers. In the first case, 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. In the second case, we plan to implement a user-level demultiplexor. In neither case is a single service granted privileged and exclusive access to the address space.

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 hosts 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 pro-

vide a shared interface to alternative monitoring services.

4.4.1 Interface

A sensor provides pieces of information that are available (or can be derived) 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 some subset of it) when queried. 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 [15] compliant server implementing a 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 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 straightforward 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 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.

In contrast, a more complex sensor server will shortly become available that 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 that some sensors may be expensive to invoke, possibly sending and receiving messages over the Internet before they can respond, and as a result may 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 [19], Sophia [40], and IrisNet [29].

The long-term goal is for these monitoring services to detect, reason about, and react to anomalous behavior before it becomes disruptive. However, 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 naïve service design and analysis, programmer errors, or hyper-sensitive 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.

Specifically, a traffic auditing service runs on every PlanetLab node, snooping all outgoing traffic using the administrative raw sniffer socket provided by the SILK module that tags each packet with the ID of the sending vserver. 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 or her 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 or she can make queries about the origin of traffic sent earlier than one hour ago.

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 him. 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 do not 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, so 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; it 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 [7].

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. It remains to be seen how rapidly a slice can be deployed on a large number of nodes.

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 an rcap 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 calls vadduser to instantiate the vserver and then calls vserver-init to 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 and 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 [40], to demonstrate how long it takes to initialize a service once a slice exists. Sophia's slice is managed by a 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 script inside each vserver. This script downloads and installs the `apt-get` tools and a root Sophia slice RPM, and starts an update process. Using `apt-get`, the update process periodically downloads the tree of current packages specific for the Sophia slice. If a newer package based on the RPM hierarchy is found, it and its dependencies are download and installed. With this mechanism, the new versions 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, it takes on average 11.2 seconds to perform an empty update on a node; i.e., to download the package tree, but not find anything new to upgrade. When a new Sophia "core" package is found and 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 180 nodes, the average update time (corresponding to the slowest node) is 228.0 seconds. The performance could be much improved, for example, by using a better distribution mechanism. Also, a faster alternative to the RPM package dependencies system could improve the locally performed dependency checks.

6 Conclusions

Based on experience providing the network research community with a platform 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 and 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, we expect a set of alternative infrastructure services to emerge and supplant these bootstrap services.

References

[1] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th SOSP*, pages 131–145, Banff, Alberta, Canada, Oct 2001.

[2] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proc. Pervasive 2002*, pages 195–210, Zurich, Switzerland, Aug 2002.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. 3rd OSDI*, pages 45–58, New Orleans, LA, Feb 1999.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfi eld. Xen and the Art of Virtualization. In *Proc. 19th SOSP*, Lake George, NY, Oct 2003.

[5] A. Bavier, T. Voigt, M. Wawrzoniak, and L. Peterson. SILK: Scout Paths in the Linux Kernel. Technical Report 2002–009, Department of Information Technology, Uppsala University, Uppsala, Sweden, Feb 2002.

[6] R. Black, P. Barham, A. Donnelly, and N. Stratford. Protocol Implementation in a Vertically Structured Operating System. In *Proc. IEEE Conf. Comp. Networks*, Nov 1997.

[7] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single Instance Storage in Windows 2000. In *Proc. 4th USENIX Windows Sys. Symp.*, pages 13–24, Seattle, WA, Aug 2000.

[8] Y. Chu, S. Rao, and H. Zhang. A Case For End System Multicast. In *Proc. SIGMETRICS 2000*, pages 1–12, Santa Clara, CA, Jun 2000.

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

[10] B. Chun and T. Spalink. Slice Creation and Management, Jun 2003. `http://www.planet-lab.org/`.

[11] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th SOSP*, pages 202–215, Banff, Alberta, Canada, Oct 2001.

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

[13] D. R. Engler and M. F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *Proc. SIGCOMM '96*, pages 53–59, Stanford, CA, Aug 1996.

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

[15] 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 Req. for Cmts.*, Jun 1999.

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

[17] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proc. 19th SOSP*, Lake George, NY, Oct 2003.

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

[19] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *Proc. 1st Int. Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, MA, Mar 2002.

[20] C. Jin, D. Wei, S. H. Low, G. Buhrmaster, J. Bunn, D. H. Choe, R. L. A. Cottrell, J. C. Doyle, W. Feng, O. Martin, H. Newman, F. Paganini, S. Ravot, and S. Singh. FAST TCP: From Theory to Experiments, Apr 2003. Submitted for publication.

[21] P.-H. Kamp and R. N. M. Watson. Jails: Confining the Omnipotent Root. In *Proc. 2nd Int. SANE Conf.*, Maastricht, The Netherlands, May 2000.

[22] 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 *Proc. 9th ASPLOS*, pages 190–201, Cambridge, MA, Nov 2000.

[23] 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 J. Sel. Areas Comm.*, 14(7):1280–1297, 1996.

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

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

[26] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. 8th ICDCS*, pages 104–111, San Jose, CA, Jun 1988.

[27] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Int. Conf. Multimedia Computing & Systems*, pages 90–99, Boston, MA, May 1994.

[28] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proc. 2nd OSDI*, pages 153–167, Seattle, WA, Oct 1996.

[29] S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. IrisNet: An Architecture for Enabling Sensor-Enriched Internet Service. Technical Report IRP–TR–03–04, Intel Research Pittsburgh, Jun 2003.

[30] 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.

[31] L. Peterson, J. Hartman, S. Muir, T. Roscoe, and M. Bowman. Evolving the Slice Abstraction, Jan 2004. `http://www.planet-lab.org/`.

[32] Plkmod: SILK in PlanetLab. `http://www.cs.princeton.edu/~acb/plkmod`.

[33] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proc. INFOCOM 2002*, pages 1190–1199, New York City, Jun 2002.

[34] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. 18th Middleware*, pages 329–350, Heidelberg, Germany, Nov 2001.

[35] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-Scale Persistent Peer-to-Peer Storage Utility. In *Proc. 18th SOSP*, pages 188–201, Banff, Alberta, Canada, Oct 2001.

[36] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A facility for distributed internet measurement. In *Proc. 4th USITS*, Seattle, WA, Mar 2003.

[37] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Peer-to-Peer Lookup Service for Internet Applications. In *Proc. SIGCOMM 2001*, pages 149–160, San Diego, CA, Aug 2001.

[38] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: Offering Internet QoS Using Overlays. In *Proc. HotNets–I*, Princeton, NJ, Oct 2002.

[39] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *Proc. 5th OSDI*, pages 345–360, Boston, MA, Dec 2002.

[40] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An Information Plane for Networked Systems. In *Proc. HotNets–II*, Cambridge, MA, Nov 2003.

[41] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proc. 5th OSDI*, pages 195–209, Boston, MA, December 2002.

[42] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. 5th OSDI*, pages 255–270, Boston, MA, Dec 2002.

[43] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. In *Proc. 11th ICNP*, Atlanta, GA, Nov 2003.