PlanetLab: An Overlay Testbed for Broad-Coverage Services

Brent Chun, David Culler, Timothy Roscoe
Intel Research – Berkeley

Andy Bavier, Larry Peterson, Mike Wawrzoniak
Princeton University

Mic Bowman
Intel Corporation

ABSTRACT

PlanetLab is a global overlay network for developing and accessing broad-coverage network services. Our goal is to grow to 1000 geographically distributed nodes, connected by a diverse collection of links. PlanetLab allows multiple services to run concurrently and continuously, each in its own *slice* of PlanetLab. This paper describes our initial implementation of PlanetLab, including the mechanisms used to implement virtualization, and the collection of core services used to manage PlanetLab.

1. INTRODUCTION

The last few years have seen the emergence of a new class of network services, including network-embedded storage [10], peer-to-peer file sharing [16, 5], content distribution networks [21], robust routing overlays [17, 2], scalable object location [3, 15, 20, 14], and scalable event propagation [6]. Researchers are also beginning to deploy network measurement and monitoring tools [18, 19].

What all these applications have in common is that they benefit from being widely distributed over the Internet. To support the design and evaluation of such applications, we are building a global overlay network called PlanetLab. Our goal is to grow PlanetLab to 1000 geographically distributed nodes connected by a diverse collection of links, including edge sites, co-location and routing centers, and homes (i.e., at the end of DSL lines and cable modems). This paper describes an early version of PlanetLab (Version 1), that has been deployed on 100 nodes distributed across 42 sites, and is currently being used to experiment with several dozen broad-coverage network services.

To appreciate the design decisions we have made, it is important to understand that while we have a short-term goal of supporting experimentation with the types of services mentioned above, our medium-term goal is to support continuously running services that potentially serve a client community. In other words, PlanetLab is designed to support the seamless migration of an application from early prototype, through multiple design iterations, to a popular service that continues to evolve. In the long-term, we envision PlanetLab serving as a microcosm for the next generation Internet [13].

2. ARCHITECTURAL OVERVIEW

The centerpiece of the PlanetLab architecture is a *slice* (a horizontal cut of global PlanetLab resources). Each *service* (a set of distributed and cooperating programs delivering some higher-level functionality) runs in a slice of PlanetLab. A slice encompasses some amount of processing, memory, storage, and network resources across a set of individual PlanetLab nodes distributed over the network. A slice is more than just the sum of the distributed resources, however. It is more accurate to view a slice as a network of *virtual machines*, with a set of local resources bound to each virtual machine.

A virtual machine is the environment where the program that implements some aspect of a service runs. Each virtual machine runs on a single node and is allowed to consume some fraction of that node's resources. In addition to being bound to a set of resources, a virtual machine also defines the programming interface (execution environment) to which programs are written. Multiple virtual machines run on each PlanetLab node, where a virtual machine monitor (VMM) arbitrates the node's resources among them.

This section gives a high-level overview of the PlanetLab architecture, first from the local (per-node) perspective, and then from the global (network-wide) perspective. This overview provides the framework in which the specific mechanisms described in the next two sections are defined.

2.1 Node Perspective

PlanetLab slices are collections of virtual machines, each running on a physical PlanetLab node. Each node, therefore, has to provide a virtual machine abstraction. This virtual machine must be useful (it should be no harder to program than a conventional server machine) and protected from other virtual machines on the same node. In addition, node resources (CPU, local disk space, network bandwidth, etc.) must be shared fairly so that one slice cannot starve another. Virtual machines must also be restricted in some ways, particularly with regard to the volume and nature of network traffic they can generate. This is because unlike many testbeds, PlanetLab is implemented over the "real" Internet, and experimental PlanetLab services must coexist peaceably with existing Internet traffic.

The kind of virtual machine available to a slice, and the operating system VMM that provides it, is one of the central design questions in PlanetLab. The solutions adopted

are also expected to change over time as PlanetLab evolves: planned obsolescence of building blocks is an important aspect of the design.

PlanetLab's design philosophy draws an important distinction between the *Application Programming Interface* used by typical services and the *Protection Interface* implemented by the VMM. The latter is the level at which both inter-VM protection and resource allocation to VMs is performed. Once separated, these two interfaces can evolve relatively independently. To some extent, solutions in the design space for PlanetLab node virtualization mechanisms are characterized by where these two interfaces are drawn.

For example, software runtimes such as the Java Virtual Machine and the Microsoft Common Language Runtime place the API at a very high level of abstraction, and rely on an underlying operating system to provide protection and resource control between applications. This option was rejected for PlanetLab because of the large amount of design policy mandated by these systems—more lightweight solutions can always support such language VMs inside their own VM, but allow much greater flexibility and can often avoid the overhead of the entire runtime.

At the other end of the scale, complete virtual machine monitors like VMware provide a protection interface at a very low level of abstraction: that of conventional hardware. This offers complete flexibility in choice of API (any operating system can be run in a VM, even without recompilation), and excellent protection properties (slices are completely encapsulated along with their associated operating systems), but comes at a high price in CPU and memory resources: even high-end commercial server VMMs typically support 10s of VMs per machine, which is insufficient for PlanetLab.

Mainstream operating systems such as Unix provide protection and resource control at the same level as the API (e.g. at the Unix system call/process interface). For example, a PlanetLab node implementation using such an OS might give each slice a process group on the machine. This has efficiency advantages: since the API and protection interfaces are relatively high-level, considerable sharing of resources (e.g., physical memory for code and data) is possible, and thus, many slices can be supported on a machine. The downside is that protection is problematic, and the number of resources that must be allocated is large, with a high-level protection interface introducing much complexity to the system (e.g., policies may now need to include file descriptors, sockets, network port numbers, loopback interfaces, and so on).

A middle ground between the complete virtualization and mainstream operating systems are recent modifications to Unix systems to provide *virtual kernels* or *virtual servers*. Examples of this approach include Linux VServers [9], The BSD Jail [**?**], User-Mode Linux [**?**], and commercial offerings such as Ensim's Private Server technology [**?**]. Such systems provide the sharing efficiencies of the Unix option, but with better possibilities for protection and resource control. On the downside, supporting alternative, future APIs in an efficient manner remains problematic. Version 1 of PlanetLab, as described in this paper, uses Linux VServers with additional resource control and protection facilities derived from the Scout [12] operating system.

Recently, the notion of an *isolation kernel* has been proposed by at least two groups in the research community [22, 7]. Isolation kernels claim to occupy an attractive point in the design space: they provide a protection interface close to real hardware suitable for running (and consequently evolving) operating systems and their APIs, and a low-level multiplexing point for system resources. Unlike pure VMs, however, the "hardware" presented to a guest operating system is considerably more efficient for the operating system to handle, and more cooperation between VMs and the VMM itself is possible.

2.2 Network Perspective

At the global level, the central challenge is discovering what resources are available, dynamically creating slices that span these resources, and launching services within these slices. This process is rootedd in the individual nodes, which issue *tickets* (credentials) that specify resource amounts (e.g., cycle and link bandwidths) and a time frame for which the resources can be acquired. The bearer of a ticket can redeem it at the node for a *lease* on the resources over the given time frame, subject to the node's admission control policy. In the abstract, this process works as follows; Section 4 described the implementation in Version 1.

First, a *node manager*, implemented as part of the VMM, runs on each node. It takes a set of tickets as input, and according to a local admission control policy, determines if the tickets can be redeemed. The ticket set is signed by an authority that the node trusts to enforce the global allocation policy. If the request can be satisfied, the node manager reserves the specified resources, creates a virtual machine and binds it to those resources, and returns a lease. This lease is later used by the service manager to launch a program within the virtual machine.

Second, a *resource monitor* is an service running on each node. It monitors resource availability on the node and periodically reports the results to one or more agents. Resource monitors depend on an interface exported by the virtual machine that allows them to record certain facts about the state of the node; e.g., the current CPU load and the available link bandwidth.

Third, an *agent* collects resource availability information from a set of resource monitors or other agents, and issues tickets that can be used to acquire resources on the monitored nodes. An agent can respond to two kinds of queries: (1) it can *advertise* the tickets it is holding that meet certain criteria, and (2) it can *grant* tickets themselves to a requester.

Fourth, a *resource broker* responds to queries from service managers trying to discover a slice to run in. Each query describes the resources needed by the service, and specifies the principal on whose behalf the request is being made. Given a query, the broker first contacts one or more agents to discover what tickets it is possible to obtain. It then matches this information with the service's resource requirements to

produce a slice specification, contacts the relevant agents to obtain the tickets, and returns them to the service manager.

Finally, a *service manager* is associated with each service. It contacts a resource broker to discover a slice and obtain the tickets needed to instantiate it. It then submits the tickets to the admission control mechanism on each node to create a network of virtual machines. Should virtual machine creation succeed on each node, the service manager then launches the service, that is, loads and starts a program in each virtual machine. Admission control returns a lease on the slice. The manager must periodically renew this lease.

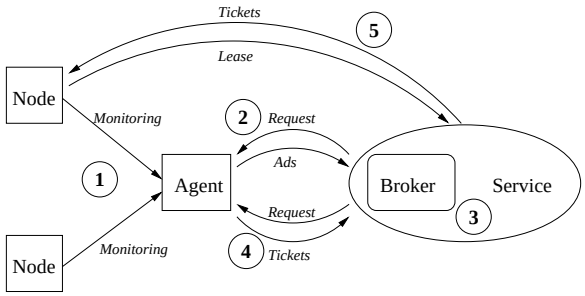

Figure 1: Acquiring a Slice

Figure 1 shows one picture of how these components might interact. Step 1 shows resource monitoring information flowing from a set of nodes to their trusted agent. At step 2, a resource broker running within the service manager requests a description of tickets held by the agent, and the agent responds with a set of advertisements. At step 3, the broker combines the advertisements with the known service requirements to produce a slice specification. The broker requests the tickets for the slice from the agent at step 4 and the agent replies with the tickets. Finally, at step 5 the service presents the tickets to the node on which they are good; the node instantiates a virtual machine bound to the resources and returns a lease to this VM to the service.

3. VIRTUAL MACHINES

This section describes how PlanetLab implements virtual machines on each node. The plan is to evolve the PlanetLab VMM, by initially supporting a single, well-known API (the Linux system call interface), and then changing the underlying implementation over time. As described in this section, Version 1 adopts the virtualized kernel strategy. We are exploring the use of isolation kernels as the basis for node virtualization in Version 2 of PlanetLab.

3.1 Vservers

Vservers is a patch to the Linux 2.4 kernel that provides the illusion of multiple, independently managed virtual servers (vservers) running on a single machine [9]. It implements a level of virtualization at the system call interface 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.

3.1.1 Virtualization

Virtualization 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 by 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. A special security context (context 1) is given a complete view of the entire machine and, as described in Section 4.3, is used to create administrative slices.

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 3.1.3). Achieving comparable amounts of sharing in a virtual machine monitor or isolation kernel approach is strictly harder, albeit the isolation guarantees are different.

Virtualizing above the kernel, however, 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 in a VMM or isolation kernel. Another 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, even in the presence of proportional-share schedulers.

3.1.2 Vserver Root

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.

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 vserver-specific `/etc/passwd` and running an `sshd` daemon a different TCP port), which provides the basis for potential integration with other wide-area testbeds such as NetBed [24] 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.

3.1.3 Scalability

Scalability in the current implementation is determined primarily 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, hundreds of vservers can easily co-exist on a given 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 [4].

Operating system resource limits are a secondary factor which ultimately determine 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.

3.2 Protected Raw Sockets

One key decision made very early in design process was that users of PlanetLab should not have root access to the machines. This is because we expect PlanetLab to support a large number of users (researchers) that cannot all be trusted to not misuse root priviledge. On the other hand, we recognize that many users will need access to services that normally require root priviledge. Access to raw sockets is one such example.

Our resolution of this dilemma is to provide a "protected" version of the priviledged service. For example, in the case of raw sockets, rather than allow a service to gain access to all incoming packets and write arbitrary packets to the network, services are forced create sockets that are bound to specific UDP or TCP ports: incoming packets are classified and delivered only the to the service that created the socket, and outgoing packets are filtered to ensure that they are properly formed (e.g., the process does not spoof the source IP address or UDP/TCP port numbers). The protected raw socket facility consults a policy database that indicates what port numbers are allocated to each service.

3.2.1 UDP and TCP Sockets

Protected raw sockets use the standard Linux socket API with minor semantic differences. Just as in standard Linux, first the socket must be created with the

```
socket(int domain, int type, int protocol);
```

system call. To create a protected raw socket, the `domain` argument must be set to `PF_INET`, `type` to `SOCK_RAW` and `protocol` can be either `IPPROTO_TCP` or `IPPROTOT_UDP` for TCP or UDP sockets, respectively. Return values are the same as in Linux.

Once the socket is created, it is necessary to bind it to a particular local port of the specified protocol. The standard Linux `bind` system call is used. For example, the following code fragment

```
bzero((char *)& sin, sizeof(sin));
sin.sin_port = htons(9090);

if((bind(sock, (struct sockaddr *)& sin, sizeof(sin)))
< 0) {
  perror("bind");
  exit(1);
}
```

binds the previously created socket to local TCP port 9090.

After the socket is created and bound to a local port, it is ready to be used to send and receive data. The usual `send, sendto, sendmsg, recv, recvfrom, recvmsg` and `select` calls can 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. The IP and UDP/TCP checksum are performed by the kernel.

3.2.2 ICMP Sockets

ICMP packets can be sent and received through protected raw ICMP sockets. To protect users from interfering with each other, each ICMP socket is allowed to send and receive only packets of the registered type bound to the socket. Similar to standard sockets, the `bind` system call is used to specify the packets that are to be received and sent through a socket. To receive and send 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. For example, the following code fratment

```
if((sock = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP_UDP))
< 0) {
  perror("socket");
  exit(1);
}

bzero((char *)& sin, sizeof(sin));
sin.sin_port = htons(9090);

if((bind(sock, (struct sockaddr *)& sin, sizeof(sin)))
< 0) {
  perror("bind");
  exit(1);
}
```

creates and binds the ICMP socket to local UDP port 9090. Only ICMP messages associated with the local UDP port 9090 can be received and sent through this socket.

To exchange ICMP messages that are not associated with a specific TCP/UDP port number—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 that is used to demultiplex/match packets. Only messages containing the right identifier are received and sent through a protected raw ICMP socket. For example, the following code fragment

```
if((sock = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP))
    < 0) {
  perror("socket");
  exit(1);
}

bzero((char *)& sin, sizeof(sin));
sin.sin_port = htons(23456);

if((bind(sock, (struct sockaddr *)& sin, sizeof(sin)))
    < 0) {
  perror("bind");
  exit(1);
 }
```

creates and binds the ICMP socket to identifier `23456`.

3.3 Resource Limits and Isolation

PlanetLab provides resource limits per vserver as well as resource isolation between the vservers running on a node. More specifically, resource limits on outgoing traffic protect the rest of the world from PlanetLab, while resource isolation between vservers protect PlanetLab services from each other. This section describes the resource management mechanisms now in place on PlanetLab and outlines future work in this area.

PlanetLab currently enforces a cap on the total outgoing bandwidth of a node while providing fair service between vservers. This is done using the hierarchical token bucket (`htb`) queueing discipline of the Linux Traffic Control facility (`tc`) [11] as follows. First, the node administrator configures the root token bucket with the maximum rate at which he is willing to allow traffic to leave the node. Next, for each vserver, a token bucket is automatically created that is a child of the root token bucket. The `htb` queueing 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. For example, if the node administrator sets the root token bucket rate to 5Mbps, and the rate of each vserver token bucket is set to 1Kbps, the 5Mbps will be fairly divided among all vservers (in this case we expect vservers to use more than 1Kbps, but we assign each the same small rate so that they fairly share the total).

In addition to this general rate-limiting facility, PlanetLab also limits the outgoing rate for certain classes of packets that may raise alarms within the network. For instance, we may choose 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 using `htb` and classifying outgoing packets so that they end up in the correct bucket. Figuring out reasonable output rates for potentially troublesome packets is ongoing work.

Isolation between vservers is desirable to minimize crosstalk among slices as they contend for resources. Two possible approaches to providing resource 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; for instance, if a slice is truly guaranteed 10Mcps on the CPU, then this guarantee protects it from excessive CPU usage by other slices. PlanetLab will ultimately provide CPU and bandwidth guarantees for slices that request them, and "fair best effort" service for the rest.

The hierarchical token bucket rate limiter described above fairly distributes bandwidth between outgoing packet flows, and can also serve as a first step towards granting bandwidth guarantees for slices. For instance, in order to provide a vserver with 1Mbps of bandwidth on a node, the rate of the vserver's token bucket can be set to this amount. The token bucket can be configured so that the vserver can share the excess capacity over 1Mbps, or its usage can be capped to 1Mbps. In either case, the admission controller must ensure that the sum of all vserver token bucket rates is no greater

than the root bucket rate in order to provide a reasonable guarantee of service. We require more experience to determine how well the `htb` mechanism is able to provide services with true bandwidth guarantees.

The Linux CPU scheduler provides approximate fairness between processes, but this raises two problems for PlanetLab. First, a vserver with many processes could use up more than its fair share of the CPU. We want to enforce isolation primarily on the level of vservers rather than processes. Second, the standard Linux scheduler cannot provide CPU guarantees in the form of reservations. We are currently planning on leveraging Scout's [12] CPU scheduling infrastructure to provide vservers with fairness and resource guarantees, but this feature has not yet been deployed in PlanetLab.

4. MANAGEMENT SERVICES

Rather than view PlanetLab management as a single, fixed service, our approach is to unbundled management of the overlay into a set of largely independent services, each running in their own slice of PlanetLab. Some aspects of management are service-specific (e.g., monitoring the health of a running service), while others are part of the shared infrastructure (e.g., discovering the set of available nodes). PlanetLab provides an initial version of the latter services, as described in this section, although our expectation is that they will be replaced by better alternatives over time.

4.1 System Installation and Update

PlanetLab poses challenges for the maintenance of infrastructure software on PlanetLab nodes, in particular because the system software is expected to evolve over time, while supporting continuously running services. It is a requirement to be able to upgrade pretty much all of PlanetLab's core software, including the operating system kernel, under remote control. Fortunately, the nature of the applications running over PlanetLab reduces the burden of maintaining continuous operation: it is safe to take a single node down for a software upgrade since applications should assume that a small number of nodes will always be going down anyway.

Our original approach to installing software on machines, adopted for reasons of rapid initial deployment, was to use the U.C. Berkeley RootStock [**?**] system. This had the advantage that it included an update daemon on each node to allow software to be updated after installation, but suffered a number of drawbacks in the PlanetLab case, the most serious being that a human operator is required to insert and remove a floppy disk when the machine is installed.

The solution currently adopted consists of three components: a powerful boot monitor (a Linux kernel booted from a CD) that allows almost arbitrary remote manipulation of the machine, a boot server that securely downloads instructions to a newly-booted machine (including complete installs of operating system software), and a process for runtime update of non-kernel packages (which we retain from RootStock). The system is centralized, but still appropriate for a federated model where multiple PlanetLabs exist run by different organisations.

4.1.1 Boot CD

The boot environment for PlanetLab was developped from the University of Cambridge XenoBoot disk [25] and shares many features with the BSD-based NetBed CD [23]; indeed the two projects have exchanged several ideas over the design. The CD supplied for PlanetLab machines is a minimal, though complete, Linux system that boots from the CD and runs from the CD and dynamic RAMdisk. The PlanetLab node *always* boots from the CD, but is capable of then booting another kernel from a harddisk if required.

At boot time, the node brings up the main network interface using DHCP, or using static address information on a floppy disk if present. It then connects to a web server (the *boot server*) using SSL, verifying the server's identity with a certificate burned onto the CD. The node posts to the boot server a variety of information about itself, including hardware specification, current network configuration, and ethernet MAC address. It recieves in return a file encrypted and signed with an offline private key held at Intel Research. The node decrypts this file and verifies its authenticity using a public key also burned on the CD, and finally executes the file as a script. What happens next is entirely up to the script, and consequently is under the control of the boot server.

The motivation for both authenticating the server and the script is to prevent replay attacks from a "fake" web server and also protect PlanetLab in the event that the "real" web server is compromised: adversaries still cannot execute arbitrary scripts on booting PlanetLab nodes since they still cannot sign a new script. During a normal install process (described below), all files downloaded from the web server are signed in this way.

4.1.2 Boot Server and Software Maintenance

The task of the boot server is to respond to this request from a booting machine with an appropriate script. Typically, one of three scripts is sent to the machine: one to perform a reinstall of the current PlanetLab software distribution on the node, one to simply boot a second kernel from the node's hard disk, and a third to bring up a heavily firewalled `ssh` server to allow remote diagnostics and login by a PlanetLab administrator. A database on the boot server holds the *boot state* for each node, including a state machine that keeps track of whether the machine needs a reformat and reinstall, or should simply boot from the hard disk as normal.

This arrangement confers great flexibility in both bringing up nodes, and installing software. In principle, nearly any filing system and kernel can be installed on the bare hardware of the node using this technique, provided that code can be written for the Linux distribution on the CD to do the job. Most importantly, this can all be written or modified *after* the CD has been distributed and the node installed. In combination with the ability the remotely power-cycle a machine, the result is a powerful remote management facility.

Since the current PlanetLab system software environment is a derivative of RedHat Linux, we can leverage RedHat's package management and installation tools to maintain Plan-

etLab nodes. The installation boot script mimics the conventional RedHat network install behavior, and we retain the use of RootStock to regularly update software packages installed on the nodes while they are running. The only changes we have made to this process is to ensure that all files downloaded from the boot server are signed by the offline key, for the reasons detailed above.

4.2 Dynamic Slice Creation

Dynamic slice creation is currently implemented as a set of daemons and a command-line program that communicate via secure RPC protocols. Each node runs a node manager daemon that implements the node's admission control policy and handles lease and virtual machine management requests. Each node manager delegates authority to issue tickets for its resources to an agent daemon that runs on a well-known machine (`www.planet-lab.org`). The agent keeps track of which node managers are available for dynamic slice creation and issues tickets in response to requests from brokers. The service manager is currently a command-line program and uses an integrated broker that speaks to a single agent to obtain tickets to create slices. Both node managers and agents also run a tiny embedded web server to allow for remote inspection of their state.

4.2.1 Resource Monitoring

Each node runs a resource monitor that periodically reports the status of its resources to an agent. The resource monitor currently uses the Ganglia cluster monitoring toolkit [8], and reports to a centralized agent running at `www.planet-lab.org`. The monitoring facility currently reports overall CPU and memory utilization, as well as per-slice network usage. This information is aggregated at the agent and made available as an XML document.

4.2.2 Agents

We currently implement a single, centralized agent, running as a daemon process on `www.planet-lab.org`. It uses use secure XML-RPC protocols to communicate with brokers, issues tickets for all nodes, and discovers such nodes by periodically polling a Ganglia resource monitor. The agent handles two main types of XML-RPCs from brokers: ticket advertisement requests (`getads`) and new ticket requests (`gettickets`). Ticket advertisement requests are handled by returning the current set of available nodes as a set of advertisements, each of which includes a node's IP address. New ticket requests are handled by performing a two-way authentication via SSL, verifying the broker is authorized to request tickets, creating a set of signed tickets, storing the tickets in a local DB file (for crash recovery), and finally returning them to the broker. Requests for tickets are specified using an XML slice description document. In the current implementation, slice description files consist of a slice name (e.g., `oceanstore`), the number of nodes requested, and a desired lease length in seconds.

4.2.3 Node Managers

A *node manager* is that part of each node's VMM that implements admission control and handles all the mechanics of creating and deleting vservers. Node managers are implemented as daemon processes that use secure XML-RPC protocols to communicate with service managers. They make themselves available for dynamic slice creation through the central agent by periodically sending an existence packet (including a heartbeat for debugging) to a local Ganglia resource monitor, and by accepting signed tickets from the trusted agent.

Using XML-RPCs over SSL, node managers handle a number of different types of requests from service managers related to lease management and access control to the virtual machines underlying the leases. The most important requests are the following: new lease requests (`newlease`), lease cancellations (`deletelease`), lease renewals (`rewewlease`), and adding (`addkey`) and removing (`delkey`) SSH keys from a lease's virtual machine. In addition, node managers also support a number of passive requests to allow probing of a given virtual machine's state (e.g., the `getsshkeys` RPC returns a list of SSH public keys in the virtual machine's SSH `.authorized_keys` file).

The most important request handled by a node manager is the lease creation request. Lease creation requests from service managers are handled first through an authentication phase: a two-way authentication via SSL, ascertaining the service manager's identity through the SSL handshake, verifying the ticket presented as part of the request is signed by a trusted agent and has not expired, and finally by verifying that the service manager's identity matches the identity of the principal named in ticket. Assuming service manager authentication succeeds, the node manager creates a new lease by creating a new virtual machine, creating a signed lease that specifies the term of the lease, storing the lease in a local DB file (for crash recovery), and returning the lease to the service manager. Creating a new virtual machine entails creating a new vserver and creating a pair of accounts, one in the main vserver and one in the vserver just created, to allow for transparent redirection using SSH/SCP into the vserver created for the virtual machine.

Vserver creation is done 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. Two Linux accounts are then created, 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 essentially a modified `bash` shell 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.

4.2.4 Service Managers and Brokers

Service managers are implemented as executions of as a command-line program that implements the secure XML-RPC protocols required to communicate with both agents and node managers. They use a simple integrated broker that is capable of probing and requesting tickets from a

single agent in the creation of a slice. They authenticate themselves to agents and node managers using an RSA key pair and an X.509 certificate signed by a trusted certificate authority, both of which are stored locally in the user's `.planetlab` directory. Service managers store information about the slices they are currently managing, including the slice's XML slice description file, unredeemed tickets, and valid leases. This information is stored on disk in the user's `.planetlab` directory and used to recover from partial completion of slice management operations. For example, if a node was unreachable when the service manager attempted to redeem a ticket for a lease, it can retry the request later by reading the unredeemed ticket from disk. Service managers perform all node manager RPCs in parallel to all nodes to reduce execution time and return either 0 or a positive error code to allow service managers to be called programmatically from other programs.

4.2.5 *Trust Relationships*

Trust relationships and delegations of trust are expressed using a combination of X.509 certificates and signed XML files. Agents accept ticket requests from brokers that present X.509 certificates signed by a trusted certificate authority and prove they possess the private keys associated with the public keys contained in the X.509 certificates. Node managers accept lease creation requests from service managers that present tickets signed by a trusted agent and prove they are the principle named in the tickets. The latter is done by authenticating the service manager as part of the SSL handshake protocol and comparing the SHA1 hash of service manager's public key to the principle named in the ticket. Both tickets and leases are expressed as XML files which include a principle, an IP address, a slice name, and an interval of UTC time where the ticket or lease is valid. Also included is the authorizing principle's RSA signature on the SHA1 hash of either the ticket or lease's XML (Figure 2).

4.3 Administrative Slices

Administrative slices provide management services with a complete view of node state along with additional capabilities to perform privileged operations. Unlike conventional virtual machines, which are confined to particular vservers, virtual machines in administrative slices run in a special security context (context 1) that exposes the entire state of the underlying physical machine. In addition, they also carry a set of Linux capabilities that specify the set of privileges that `root` should have within a specific administrative slice. The former allows management services to perform tasks such as conventional virtual machine management and distributed process monitoring, tasks which would be impossible within a conventional slice due to the isolation enforced by vservers. The latter allows management services to perform tasks such as passive monitoring of all outgoing network packets, a task that requires elevated privileges, in this case the `CAP_NET_RAW` Linux capability.

Administrative slices are created in a similar manner as that of conventional slices but require additional details to be specified and require additional user privileges in order for tickets to be obtained from an agent. To request tickets for an administrative slice, users augment their XML slice

```
sig-rsa-sha1-base64: m/P0ttuTL2NadGyWhKZKdSC0ul01R6
jhZ4J4C7zmlhs+cKdBUnxVHNX0ma9RE2xS3L+Ns4nYatunqXdih
khwY/Gp4PVKWPsHlxd0y2gugbdC3sTps6v3NDqIaJz1Aoxx6fIK
rEMR6SZcY5l+30ujnoLqGMTWz6tIc6IXSVRnDpw=

<?xml version="1.0" ?>
<lease>
    <principle_sha1>
        ec6c223a8a2a8be1caf90f8b51e8c121f805c4a4
    </principle_sha1>
    <ip>
        12.155.161.149
    </ip>
    <slice>
        oceanstore
    </slice>
    <start_time>
        2002-12-07 01:52:26
    </start_time>
    <end_time>
        2003-06-07 01:52:26
    </end_time>
</lease>
```

Figure 2: An example lease.

description to specify a request for an administrative slice and include a set of Linux capabilities for `root` within that slice. Unlike conventional slices, which may be created by any PlanetLab principle investigator, administrative slices may only be created by PlanetLab administrators. Restrictions on which users may create which types of slices are enforced by the agent running on `www.planet-lab.org` by controlling the issuing of tickets for specific slice types based on user privileges as stored in a PostgreSQL database.

5. CONCLUDING REMARKS

At the time of this writing, PlanetLab is being used by over 140 researchers around the world, with as many as a dozen services on the verge of running continuously. We are currently testing a beta release of the Boot CD, which we expect to facilitate substantial growth over the next several months.

This paper describes the key elements of Version 1 of the PlanetLab software, including both support for virtual machines running on each node, and the global management facilities needed to keep the software up-to-date and allocate resources to services. By no means, however, do we believe that Version 1 represents the final system. As outlined in Section 2, for example, the design space for both virtual machines and global resource allocation are quite rich, and we expect both to be a focus of continuing research in the next few years. In addition, once PlanetLab begins to host continuously running overlay services, we expect the issue of service composition to come to the forefront. Recognizing and codifying the common sub-services is the key to evolving the next generation Internet.

6. 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, October 2001.

[2] D. Andersen, H. Balakrishnan, M. 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.

[3] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proceedings of Pervasive 2002 - International Conference on Pervasive Computing*, Zurich, Switzerland, August 2002.

[4] 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*, August 2000.

[5] 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)*, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[6] P. Druschel, M. Castro, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20, 2002.

[7] K. Fraser, S. Hand, T. Harris, I. Leslie, and I. Pratt. The Xenoserver Computing Infrastructure, 2002. http://www.cl.cam.ac.uk/Research/SRG/netos/xeno/xeno-general.pdf.

[8] Ganglia. http://ganglia.sourceforge.net.

[9] J. Gelinas. Virtual private servers and security contexts. http://www.solucorp.qc.ca/miscprj/s_context.hc.

[10] 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 Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Nov. 2000.

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

[12] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 153–167, Berkeley, CA, USA, Oct. 1996. USENIX.

[13] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the 1st ACM Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[14] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proceedings of the IEEE INFOCOM Conference*, New York, NY, June 2002.

[15] 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)*, Heidelberg, Germany, November 2001.

[16] 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.

[17] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The End-to-end Effects of Internet Path Selection. In *Proceedings of the ACM SIGCOMM Conference*, Cambridge, MA, September 1999.

[18] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proceedings of the ACM SIGCOMM Conference*, pages 133–146, Pittsburgh, PA, August 2002.

[19] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A facility for distributed internet measurement. In *Proceedings of the 4th USITS Symposium*, Seattle, WA, March 2003.

[20] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM Conference*, San Diego, CA, September 2001.

[21] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *Proceedings of the 5th Symposium on Operating System Design and Implementatio (OSDI)*, Boston, MA, December 2002.

[22] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating System Design and Implementatio (OSDI)*, Boston, MA, December 2002.

[23] 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 5th Symposium on Operating System Design and Implementatio (OSDI)*, Boston, MA, December 2002.

[24] 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 5th USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

[25] Xenoboot. http://www.cl.cam.ac.uk/Research/SRG/netos/xeno/boot/.