Experiences Building PlanetLab

Larry Peterson, Andy Bavier, Marc E. Fiuczynski, Steve Muir
Department of Computer Science
Princeton University

Abstract. This paper reports our experiences building PlanetLab over the last four years. It identifies the requirements that shaped PlanetLab, explains the design decisions that resulted from resolving conflicts among these requirements, and reports our experience implementing and supporting the system. Due in large part to the nature of the "PlanetLab experiment," the discussion focuses on synthesis rather than new techniques, balancing system-wide considerations rather than improving performance along a single dimension, and learning from feedback from a live system rather than controlled experiments using synthetic workloads.

1 Introduction

PlanetLab is a global platform for deploying and evaluating network services [21, 3]. In many ways, it has been an unexpected success. It was launched in mid-2002 with 100 machines distributed to 40 sites, but today includes 700 nodes spanning 336 sites and 35 countries. It currently hosts 2500 researchers affiliated with 600 projects. It has been used to evaluate a diverse set of planetary-scale network services, including content distribution [33, 8, 24], anycast [35, 9], DHTs [26], robust DNS [20, 25], large-file distribution [19, 1], measurement and analysis [30], anomaly and fault diagnosis [36], and event notification [23]. It supports the design and evaluation of dozens of long-running services that transport an aggregate of 3-4TB of data every day, satisfying tens of millions of requests involving roughly one million unique clients and servers.

To deliver this utility, PlanetLab innovates along two main dimensions:

- **Novel management architecture.** PlanetLab administers nodes owned by hundreds of organizations, which agree to allow a worldwide community of researchers—most complete strangers—to access their machines. PlanetLab must manage a complex relationship between node owners and users.

- **Novel usage model.** Each PlanetLab node should gracefully degrade in performance as the number of users grows. This gives the PlanetLab community an incentive to work together to make best use of its shared resources.

In both cases, the contribution is not a new mechanism or algorithm, but rather a synthesis (and full exploitation) of carefully selected ideas to produce a fundamentally new system.

Moreover, the process by which we designed the system is interesting in its own right:

- **Experience-driven design.** PlanetLab's design evolved incrementally based on experience gained from supporting a live user community. This is in contrast to most research systems that are designed and evaluated under controlled conditions, contained within a single organization, and evaluated using synthetic workloads.

- **Conflict-driven design.** The design decisions that shaped PlanetLab were responses to conflicting requirements. The result is a comprehensive architecture based more on balancing global considerations than improving performance along a single dimension, and on real-world requirements that do not always lend themselves to quantifiable metrics.

One could view this as a new model of system design, but of course it isn't [6, 27].

This paper identifies the requirements that shaped the system, explains the design decisions that resulted from resolving conflicts among these requirements, and reports our experience building and supporting the system. A side-effect of the discussion is a fairly complete overview of PlanetLab's current architecture, but the primary goal is to describe the design decisions that went into building PlanetLab, and to report the lessons we have learned in the process. For a comprehensive definition of the PlanetLab architecture, the reader is referred to [22].

2 Background

This section identifies the requirements we understood at the time PlanetLab was first conceived, and sketches the high-level design proposed at that time. The discussion includes a summary of the three main challenges we have faced, all of which can be traced to tensions between the requirements. The section concludes by looking at the relationship between PlanetLab and similar systems.

2.1 Requirements

PlanetLab's design was guided by five major requirements that correspond to objectives we hoped to achieve as well as constraints we had to live with. Although we recognized all of these requirements up-front, the following discussion articulates them with the benefit of hindsight.

(R1) It must provide a global platform that supports both short-term experiments and long-running services. Unlike previous testbeds, a revolutionary goal of PlanetLab was that it support experimental services that could run continuously and support a real client workload. This implied that multiple services be able to run concurrently since a batch-scheduled facility is not conducive to a 24×7 workload. Moreover, these services (experiments) should be isolated from each other so that one service does not unduly interfere with another.

(R2) It must be available immediately, even though no one knows for sure what "it" is. PlanetLab faced a dilemma: it was designed to support research in broad-coverage network services, yet its management (control) plane is itself such a service. It was necessary to deploy PlanetLab and start gaining experience with network services before we fully understood what services would be needed to manage it. As a consequence, PlanetLab had to be designed with explicit support for evolution. Moreover, to get people to use PlanetLab—so we could learn from it—it had to be as familiar as possible; researchers are not likely to change their programming environment to use a new facility.

(R3) We must convince sites to host nodes running code written by unknown researchers from other organizations. PlanetLab takes advantage of nodes contributed by research organizations around the world. These nodes, in turn, host services on behalf of users from other research organizations. The individual *users* are unknown to the node *owners*, and to make matters worse, the services they deploy often send potentially disruptive packets into the Internet. That sites own and host nodes, but trust PlanetLab to administer them, is unprecedented at the scale PlanetLab operates. As a consequence, we must correctly manage the trust relationships so that the risks to each site are less than the benefits they derive.

(R4) Sustaining growth depends on support for autonomy and decentralized control. PlanetLab is a world-wide platform constructed from components owned by many autonomous organizations. Each organization must retain some amount of control over how their resources are used, and PlanetLab as a whole must give geographic regions and other communities as much autonomy as possible in defining and managing the system. Generally, sustaining such a system requires minimizing centralized control.

(R5) It must scale to support many users with minimal resources. While a commercial variant of PlanetLab might have cost recovery mechanisms to provide resource guarantees to each of its users, PlanetLab must operate in an under-provisioned environment. This means conservative allocation strategies are not practical, and it is necessary to promote efficient resource sharing. This includes both physical resources (e.g., cycles, bandwidth, and memory) and logical resources (e.g., IP addresses).

Note that while the rest of this paper discusses the many tensions between these requirements, two of them are quite synergistic. The requirement that we evolve PlanetLab (R2) and the need for decentralized control (R4) both point to the value of factoring PlanetLab's management architecture into a set of building block components with well-defined interfaces. A major challenge of building PlanetLab was to understand exactly what these pieces should be.

To this end, PlanetLab originally adopted an organizing principle called *unbundled management*, which argued that the services used to manage PlanetLab should themselves be deployed like any other service, rather than bundled with the core system. The case for unbundled management has three arguments: (1) to allow the system to more easily evolve; (2) to permit third-party developers to build alternative services, enabling a software bazaar, rather than rely on a single development team with limited resources and creativity; and (3) to permit decentralized control over PlanetLab resources, and ultimately, over its evolution.

2.2 Initial Design

PlanetLab supports the required usage model through *distributed virtualization*—each service runs in a *slice* of PlanetLab's global resources. Multiple slices run concurrently on PlanetLab, where slices act as network-wide containers that isolate services from each other. Slices were expected to enforce two kinds of isolation: *resource isolation* and *security isolation*, the former concerned with minimizing performance interference and the latter concerned with eliminating namespace interference.

At a high-level, PlanetLab consists of a centralized front-end, called PlanetLab Central (PLC), that remotely manages a set of nodes. Each node runs a *node manager* (NM) that establishes and controls *virtual machines* (VM) on that node. We assume an underlying *virtual machine monitor* (VMM) implements the VMs. Users create slices through operations available on PLC, which results in PLC contacting the NM on each node to create a local VM. A set of such VMs defines the slice.

We initially elected to use a Linux-based VMM due to Linux's high mind-share [3]. Linux is augmented with Vservers [16] to provide security isolation and a set of schedulers to provide resource isolation.

2.3 Design Challenges

Like many real systems, what makes PlanetLab interesting to study—and challenging to build—is how it deals with the constraints of reality and conflicts among requirements. Here, we summarize the three main challenges; subsequent sections address each in more detail.

First, unbundled management is a powerful design principle for evolving a system, but we did not fully understand what it entailed nor how it would be shaped by other aspects of the system. Defining PlanetLab's management architecture—and in particular, deciding how to factor management functionality into a set of independent pieces—involved resolving three main conflicts:

- minimizing centralized components (R4) yet maintaining the necessary trust assumptions (R3);

- balancing the need for slices to acquire the resources they need (R1) yet coping with scarce resources (R5);

- isolating slices from each other (R1) yet allowing some slices to manage other slices (R2).

Section 3 discusses our experiences evolving PlanetLab's management architecture.

Second, resource allocation is a significant challenge for any system, and this is especially true for PlanetLab, where the requirement for isolation (R1) is in conflict with the reality of limited resources (R5). Part of our approach to this situation is embodied in the management structure described in Section 3, but it is also addressed in how scheduling and allocation decisions are made on a per-node basis. Section 4 reports our experience balancing isolation against efficient resource usage.

Third, we must maintain a stable system on behalf of the user community (R1) and yet evolve the platform to provide long-term viability and sustainability (R2). Section 5 reports our operational experiences with PlanetLab, and the lessons we have learned as a result.

2.4 Related Systems

An important question to ask about PlanetLab is whether its specific design requirements make it unique, or if our experiences can apply to other systems. Our response is that PlanetLab shares "points of pain" with three similar systems—ISPs, hosting centers, and the GRID—but pushes the envelope relative to each.

First, PlanetLab is like an ISP in that it has many points-of-presence and carries traffic to/from the rest of the Internet. Like ISPs (but unlike hosting centers and the GRID), PlanetLab has to provide mechanisms that can by used to identify and stop disruptive traffic. PlanetLab goes beyond traditional ISPs, however, in that it has to deal with arbitrary (and experimental) network services, not just packet forwarding.

Second, PlanetLab is like a hosting center in that its nodes support multiple VMs, each on behalf of a different user. Like a hosting center (but unlike the GRID or ISPs), PlanetLab has to provide mechanisms that enforce isolation between VMs. PlanetLab goes beyond hosting centers, however, because it includes third-party services that manage other VMs, and because it must scale to large numbers of VMs with limited resources.

Third, PlanetLab is like the GRID in that its resources are owned by multiple autonomous organizations. Like the GRID (but unlike an ISP or hosting center), PlanetLab has to provide mechanisms that allow one organization to grant users at another organization the right to use its resources. PlanetLab goes far beyond the GRID, however, in that it scales to hundreds of "peering" organizations by avoiding pair-wise agreements.

PlanetLab faces new and unique problems because it is at the intersection of these three domains. For example, combining multiple independent VMs with a single IP address (hosting center) and the need to trace disruptive traffic back to the originating user (ISP) results in a challenging problem. PlanetLab's experiences will be valuable to other systems that may emerge where any of these domains intersect, and may in time influence the direction of hosting centers, ISPs, and the GRID as well.

3 Slice Management

This section describes the slice management architecture that evolved over the past four years. While the discussion includes some details, it primarily focuses on the design decisions and the factors that influenced them.

3.1 Trust Assumptions

Given that PlanetLab sites and users span multiple organizations (R3), the first design issue was to define the underlying trust model. Addressing this issue required that we identify the key principals, explicitly state the trust assumptions among them, and provide mechanisms that are consistent with this trust model.

Over 300 autonomous organizations have contributed nodes to PlanetLab (they each require control over the nodes they own) and over 300 research groups want to deploy their services across PlanetLab (the node owners need assurances that these services will not be disruptive). Clearly, establishing 300×300 pairwise trust relationships is an unmanageable task, but it is well-understood that a trusted intermediary is an effective way to manage such an N×N problem.

PLC is one such trusted intermediary: node owners trust PLC to manage the behavior of VMs that run on their nodes while preserving their autonomy, and researchers trust PLC to provide access to a set of nodes that are capable of hosting their services. Recognizing this role for PLC, and organizing the architecture around
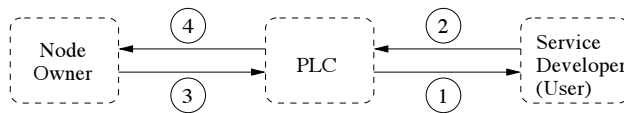
Figure 1: Trust relationships among principals.

it, is the single most important aspect of the design beyond the simple model presented in Section 2.2.

With this backdrop, the PlanetLab architecture recognizes three main principals:

- PLC is a trusted intermediary that manages nodes on behalf a set of owners, and creates slices on those nodes on behalf of a set of users.

- An *owner* is an organization that hosts (owns) PlanetLab nodes. Each owner retains ultimate control over their own nodes, but delegates management of those nodes to the trusted PLC intermediary. PLC provides mechanisms that allow owners to define resource allocation policies on their nodes.

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

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

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

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

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

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

Given this model, the security architecture includes the following mechanisms. First, each node boots from an immutable file system, loading (1) a *boot manager* program, (2) a public key for PLC, and (3) a node-specific secret key. We assume that the node is physically secured by the owner in order to keep the key secret, although a hardware mechanism such as TCPA could also be leveraged. The node then contacts a boot server running at PLC, authenticates the server using the public key, and uses HMAC and the secret key to authenticate itself to PLC. Once authenticated, the boot server ensures that the appropriate VMM and the NM are installed on the node, thus satisfying the fourth trust relationship.

Second, once PLC has vetted an organization through an off-line process, users at the site are allowed to create accounts and upload their private keys. PLC then installs these keys in any VMs (slices) created on behalf of those users, and permits access to those VMs via `ssh`. Currently, PLC requires that new user accounts are authorized by a *principal investigator* associated with each site—this provides some degree of assurance that accounts are only created by legitimate users with a connection to a particular site, thus satisfying the first trust relationship.

Third, PLC runs an auditing service that records information about all packet flows coming out of the node. The auditing service offers a public, web-based interface on each node, through which anyone that has received unwanted network traffic from the node can determine the responsible users. PLC archives this auditing information by periodically downloading the audit log.

3.2 Virtual Machines and Resource Pools

Given the requirement that PlanetLab support long-lived slices (R1) and accommodate scarce resources (R5), the second design decision was to decouple slice creation from resource allocation. In contrast to a hosting center that might create a VM and assign it a fixed set of resources as part of an SLA, PlanetLab creates new VMs without regard for available resources—each such VM is given a fair share of the available resources on that node whenever it runs—and then expects slices to engage one or more *brokerage services* to acquire resources.

To this end, the NM supports two abstract objects: *virtual machines* and *resource pools*. The former is a container that provides a point-of-presence on a node for a slice. The latter is a collection of physical and logical resources that can be bound to a VM. The NM supports operations to create both objects, and to bind a pool to a VM for some fixed period of time. Both types of objects are specified by a *resource specification* (rspec), which is a list of attributes that describe the object. A VM can run as soon as it is created, and by default is given a fair share of the node's unreserved capacity. When a resource pool

is bound to a VM, that VM is allocated the corresponding resources for the duration of the binding.

Global management services use these per-node operations to create PlanetLab-wide slices and assign resources to them. Two such service types exist today: *slice creation services* and *brokerage services*. These services can be separate or combined into a single service that both creates and provisions slices. At the same time, different implementations of brokerage services are possible (e.g., market-based services that provide mechanisms for buying and selling resources [10, 14], and batch scheduling services that simply enforce admission control for use of a finite resource pool [7]).

As part of the resource allocation architecture, it was also necessary to define a policy that governs how resources are allocated. On this point, owner autonomy (R4) comes into play: only owners are allowed to invoke the "create resource pool" operation on the NM that runs on their nodes. This effectively defines the one or more "root" pools, which can subsequently be split into subpools and reassigned. An owner can also directly allocate a certain fraction of its node's resources to the VM of a specific slice, thereby explicitly supporting any services the owner wishes to host.

3.3 Delegation

PlanetLab's management architecture was expected to evolve through the introduction of third-party services (R2). We viewed the NM interface as the key feature, since it would support the many third-party creation and brokerage services that would emerge. We regarded PLC as merely a "bootstrap" mechanism that could be used to deploy such new global management services, and thus, we expected PLC to play a reduced role over time.

However, experience showed this approach to be flawed. This is for two reasons, one fundamental and one pragmatic. First, it failed to account for PLC's central role in the trust model of Section 3.1. Maintaining trust relationships among participants is a critical role played by PLC, and one not easily passed along to other services. Second, researchers building new management services on PlanetLab were not interested in replicating all of PLCs functionality. Instead of using PLC to bootstrap a comprehensive suite of management services, researchers wanted to leverage some aspects of PLC and replace others.

To accommodate this situation, PLC is today structured as follows. First, each owner implicitly assigns all of its resources to PLC for redistribution. The owner can override this allocation by granting a set of resources to a specific slice, or divide resources among multiple brokerage services, but by default all resources are allocated to PLC.

Second, PLC runs a slice creation service—called pl_conf—on each node. This service runs in a standard VM and invokes the NM interface without any additional privilege. It also exports an XML-RPC interface by which anyone can invoke its services. This is important because it means other brokerage and slice creation services can use pl_conf as their point-of-presence on each node rather than have to first deploy their own slice. Originally, the PLC/pl_conf interface was private as we expected management services to interact directly with the node manager. However, making this a well-defined, public interface has been a key to supporting delegation.

Third, PLC provides a front-end—available either as a GUI or as a programmatic interface at www.planet-lab.org—through which users create slices. The PLC front-end interacts with pl_conf on each node with the same XML-RPC interface that other services use.

Finally, PLC supports two methods by which slices are actually instantiated on a set of nodes: *direct* and *delegated*. Using the direct method, the PLC front-end contacts pl_conf on each node to create the corresponding VM and assign resources to it. Using delegation, a slice creation service running on behalf of a user contacts PLC for a *ticket* that encapsulates the right to create a VM or redistribute a pool of resources. A ticket is a signed rspec; in this case, it is signed by PLC. The agent then contacts pl_conf on each node to redeem this ticket, at which time pl_conf validates it and calls the NM to create a VM or bind a pool of resources to an existing VM. The mechanisms just described currently support two slice creation services (PLC and Emulab [34], the latter uses tickets granted by the former), and two brokerage services (Sirius [7] and Bellagio [2], the first of which is granted capacity as part of a root resource allocation decision).

Note that the delegated method of slice creation is push-based, while the direct method is pull-based. With delegation, a slice creation service contacts PLC to retrieve a ticket granting it the right to create a slice, and then performs an XML-RPC call to pl_conf on each node. For a slice spanning a significant fraction of PlanetLab's nodes, an implementation would likely launch multiple such calls in parallel. In contrast, PLC uses a polling approach: each pl_conf contacts PLC periodically to retrieve a set of tickets for the slices it should run.

While the push-based approach can create a slice in less time, the advantage of pull-based approach is that it enables slices to persist across node reinstalls. Nodes cannot be trusted to have persistent state since they are completely reinstalled from time to time due to unrecoverable errors such as corrupt local file systems. The pull-based strategy views all nodes as maintaining only soft state, and gets the definitive list of slices for that node from PLC. Therefore, if a node is reinstalled, all of its slices are automatically recreated. Delegation makes it

possible for others to develop alternative slice creation semantics—for example, a "best effort" system that ignores such problems—but PLC takes the conservative approach because it is used to create slices for essential management services.

3.4 Federation

Given our desire to minimize the centralized elements of PlanetLab (R4), our next design decision was to make it possible for multiple independent PlanetLab-like systems to co-exist and federate with each other. Note that this issue is distinct from delegation, which allows multiple management services to co-exist withing a single Planet-Lab.

There are three keys to enabling federation. First, there must be well-defined interfaces by which independent instances of PLC invoke operations on each other. To this end, we observe that our implementation of PLC naturally divides into two halves: one that creates slices on behalf of users and one that manages nodes on behalf of owners, and we say that PLC embodies a *slice authority* and a *management authority*, respectively. Corresponding to these two roles, PLC supports two distinct interfaces: one that is used to create and control slices, and one that is used to boot and manage nodes. We claim that these interfaces are minimal, and hence, define the "narrow waist" of the PlanetLab hourglass.

Second, supporting multiple independent PLCs implies the need to name each instance. It is PLC in its slice authority role that names slices, and its name space must be extended to also name slice authorities. For example, the slice cornell.cobweb is implicitly plc.cornell.cobweb, where plc is the top-level slice authority that approved the slice. (As we generalize the slice name space, we adopt "." instead of "_" as the delimiter.) Note that this model enables a hierarchy of slice authorities, which is in fact already the case with plc.cornell, since PLC trusts Cornell to approve local slices (and the users bound to them).

This generalization of the slice naming scheme leads to several possibilities:

- PLC delegates the ability to create slices to regional slice authorities (e.g., plc.japan.utokyo.ubiq);

- organizations create "private" PlanetLab's (e.g., epfl.chawla) that possibly peer with each other, or with the "public" PlanetLab; and

- alternative "root" naming authorities come into existence, such as one that is responsible for commercial (for-profit) slices (e.g., com.startup.voip).

The third of these is speculative, but the first two scenarios have already happened or are in progress, with five private PlanetLabs running today and two regional slice

Service	Lines of Code	Language
Node Manager	2027	Python
Proper	5752	C
pl_conf	1975	Python
Sirius	850	Python
Stork	12803	Python
CoStat + CoMon	1155	C
PlanetFlow	5932	C

Table 1: Source lines of code for various management services

authorities planned for the near future. Note that there must be a single global naming authority that ensures all top-level slice authority names are unique. Today, PLC plays that role.

The third key to federation is to design pl_conf so that it is able to create slices on behalf of many different slice authorities. Node owners allocate resources to the slice authorities they want to support, and configure pl_conf to accept tickets signed by slice authorities that they trust. Note that being part of the "public" PlanetLab carries the stipulation that a certain minimal amount of capacity be set aside for slices created by the PLC slice authority, but owners can reserve additional capacity for other slice authorities and for individual slices.

3.5 Least Privilege

We conclude our description of PlanetLab's management architecture by focusing on the node-centric issue of how management functionality has been factored into self-contained services, moved out of the NM and isolated in their own VMs, and granted minimal privileges.

When PlanetLab was first deployed, all management services requiring special privilege ran in a single root VM as part of a monolithic node manager. Over time, stand-alone services have been carved off of the NM and placed in their own VMs, multiple versions of some services have come and gone, and new services have emerged. Today, there are five broad classes of management services. The following summarizes one particular "suite" of services that a user might engage; we also identify alternative services that are available.

Slice Creation Service: pl_conf is the default slice creation service. It requires no special privilege: the node owner creates a resource pool and assigns it to pl_conf when the node boots. Emulab [34] offers an alternative slice creation service that uses tickets granted by PLC and redeemed by pl_conf.

Brokerage Service: Sirius [7] is the most widely used brokerage service. It performs admission control on a resource pool set aside for one-hour experiments.

Sirius requires no special privilege: pl_conf allocates a sub-pool of resources to Sirius. Bellagio [2] and Tycoon [14] are alternative market-based brokerage services that are initialized in the same way.

Monitoring Service: CoStat is a low-level instrumentation program that gathers data about the state of the local node. It is granted the ability to read `/proc` files that report data about the underlying VMM, as well as the right to execute scripts (e.g., `ps` and `top`) in the root context. Multiple additional services—e.g., CoMon [31], PsEPR [5], SWORD [18]—then collect and process this information on behalf of users. These services require no additional privilege.

Environment Service: Stork [12] deploys, updates, and configures services and experiments. Stork is granted the right to mount the file system of a client slice, which Stork then uses to install software packages required by the slice. It is also granted the right to mark a file as immutable, so that it can safely be shared among slices without any slice being able to modify the file. Emulab and AppManager [28] provide alternative environment services without extra privilege; they simply provide tools for uploading software packages.

Auditing Service: PlanetFlow [11] is an auditing service that logs information about packet flows, and is able to map externally visible network activity to the responsible slice. PlanetFlow is granted the right to run `ulogd` in the root context to retrieve log information from the VMM.

The need to grant narrowly-defined privileges to certain management services has led us to define a mechanism called Proper (PRivileged OPERation) [17]. Proper uses an ACL to specify the particular operations that can be performed by a VM that hosts a management service, possibly including argument constraints for each operation. For example, the CoStat monitoring service gathers various statistics by reading `/proc` files in the root context, so Proper constrains the set of files that can be opened by CoStat to only the necessary directories. For operations that affect other slices directly, such as mounting the slice's file system or executing a process in that slice, Proper also allows the target slice to place additional constraints on the operations that can be performed e.g., only a particular directory may be mounted by Stork. In this way we are able to operate each management service with a small set of additional privileges above a normal slice, rather than giving out coarse-grained capabilities such as those provided by the standard Linux kernel, or co-locating the service in the root context.

Finally, Table 1 quantifies the impact of moving functionality out of the NM in terms of lines-of-code. The LOC data is generated using David A. Wheeler's 'SLOC-Count'. Note that we show separate data for Proper and the rest of the node manager; Proper's size is in part a function of its implementation in C.

One could argue that these numbers are conservative, as there are additional services that this list of management services employ. For example, CoBlitz is a large file transfer mechanism that is used by Stork and Emulab to disseminate large files across a set of nodes. Similarly, a number of these services provide a web interface that must run on each node, which would greatly increase the size of the TCB if the web server itself had to be included in the root context.

4 Resource Allocation

One of the most significant challenges for PlanetLab has been to maximize the platform's utility for a large user community while dealing with the reality of limited resources. This challenge has led us to a model of weak resource isolation between slices. We implement this model through fair sharing of CPU and network bandwidth, simple mechanisms to avoid the worst kinds of interference on other resources like memory and disk, and tools to give users information about resource availability on specific nodes. This section reports our experiences with this model in practice, and describes some of the techniques we've adopted to make the system as effective and stable as possible.

4.1 Workload

PlanetLab supports a workload mixing one-off experiments with long-running services. A complete characterization of this workload is beyond the scope of this paper, but we highlight some important aspects below.

CoMon—one of the performance-monitoring services running on PlanetLab—classifies a slice as *active* on a node if it contains a process, and *live* if, in the last five minutes, it used at least 0.1% (300ms) of the CPU. Figure 2 shows, by quartile, the number of active and live slices across PlanetLab during the past year. Each graph shows five lines; 25% of PlanetLab nodes have values that fall between the first and second lines, 25% between the second and third, and so on.

Looking at each graph in more detail, Figure 2(a) illustrates that the number of active slices on most PlanetLab nodes has grown steadily. The median active slice count has increased from 40 slices in March 2005 to the mid-50s in April 2006, and the maximum number of active slices has increased from 60 to 90. PlanetLab nodes can support large numbers of mostly idle slices because each VM is very lightweight. Additionally, the data shows that 75% of PlanetLab nodes have consistently had at least 40

(a) Active slices, by quartile
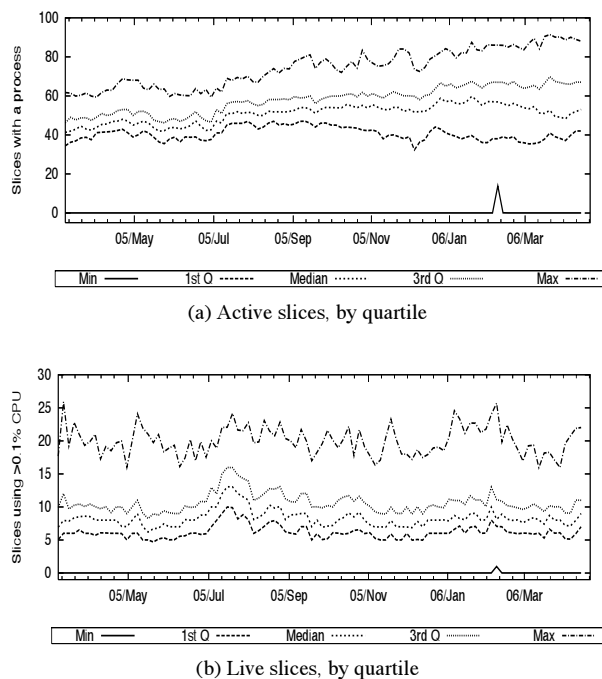

(b) Live slices, by quartile

Figure 2: Active and live slices on PlanetLab

active slices during the past year.

Figure 2(b) shows the distribution of live slices. Note that at least 50% of PlanetLab nodes consistently have a live slice count within two of the median. Additional data indicates that this is a result of three factors. First, some monitoring slices (like CoMon and PlanetFlow) are live everywhere, and so create a lower bound on the number of live slices. Second, most researchers do not appear to greedily use more nodes than they need; for example, only 10% of slices are deployed on all nodes, and 60% are deployed on less than 50 nodes. We presume researchers are self-organizing their services and experiments onto disjoint sets of nodes so as to distribute load, although there are a small number of popular nodes that support over 25 live slices. Third, the slices that *are* deployed on all nodes are not live on all of them at once. For instance, in April 2006 we observed that CoDeeN was active on 436 nodes but live on only 269. Robust (and adaptive) long-running services are architected to dynamically balance load to less utilized nodes [33, 26].

Of course we did not know what PlanetLab's workload would look like when we made many early design decisions. As reported in Section 2.2, one such decision was to use Linux+Vservers as the VMM, primarily because of the maturity of the technology. Since this time, alternatives like Xen have advanced considerably, but we have not felt compelled to reconsider this decision. A key reason is that PlanetLab nodes run up to 25 live VMs,

and up to 90 active VMs, at at time. This is possible because we could build a system that supports resource overbooking and graceful degradation on a framework of Vserver-based VMs. In contrast, Xen allocates specific amounts of resources, such as physical memory and disk, to each VM. For example, on a typical PlanetLab node with 1GB memory, Xen can support only 10 VMs with 100MB memory each, or 16 with 64MB memory. Therefore, it's not clear how a PlanetLab based on Xen could support our current user base. Note that the management architecture presented in the previous section is general enough to support multiple VM types (and a Xen prototype is running in the lab), but resource constraints make it likely that most PlanetLab slices will continue to use Vservers for the foreseeable future.

4.2 Graceful Degradation

PlanetLab's usage model is to allow as many users on a node as want to use it, enable resource brokers that are able to secure guaranteed resources, and gracefully degrade the node's performance as resources become over utilized. This section describes the mechanisms that support such behavior and evaluates how well they work.

4.2.1 CPU

The earliest version of PlanetLab used the standard Linux CPU scheduler, which provided no CPU isolation between slices: a slice with 400 Java threads would get 400 times the CPU of a slice with one thread. This situation occasionally led to collapse of the system and revealed the need for a slice-aware CPU scheduler.

Fair share scheduling [32] does not collapse under load, but rather supports graceful degradation by giving each scheduling container proportionally fewer cycles. Since mid-2004, PlanetLab's CPU scheduler has performed fair sharing among slices. During that time, however, PlanetLab has run three distinct CPU schedulers: v2 used the SILK scheduler [3], v3.0 introduced CKRM (a community project in its early stages), and v3.2 (the current version) uses a modification of Vserver's CPU rate limiter to implement fair sharing and reservations. The question arises, why so many CPU schedulers?

The answer is that, for the most part, we switched CPU schedulers for reasons other than scheduling behavior. We switched from SILK to CKRM to leverage a community effort and reduce our code maintenance burden. However, at the time we adopted it, CKRM was far from production quality and the stability of PlanetLab suffered as a result. We then dropped CKRM and wrote another CPU scheduler, this time based on small modifications to the Vservers code that we had already incorporated into the PlanetLab kernel. This CPU scheduler gave us the capability to provide slices with CPU reservations as well as shares, which we lacked with SILK and CKRM. Per-
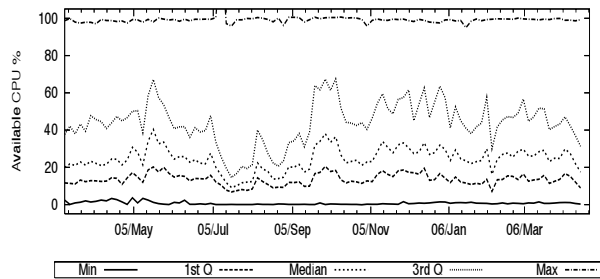
Figure 3: CPU % Available on PlanetLab

Figure 4: CDF of memory consumed when slice reset

haps more importantly, the scheduler was more robust, so PlanetLab's stability dramatically improved, as shown in Section 5. We are solving the code maintenance problem by working with the Vservers developers to incorporate our modifications into their main distribution.

The current (v3.2) CPU scheduler implements fair sharing and work-conserving CPU reservations by overlaying a token bucket filter on top of the standard Linux CPU scheduler. Each slice has a token bucket that accumulates tokens at a specified rate; every millisecond, the slice that owns the running process is charged one token. A slice that runs out of tokens has its processes removed from the runqueue until its bucket accumulates a minimum amount of tokens. This filter was already present in Vservers, which used it to put an upper bound on the amount of CPU that any one VM could receive; we simply modified it to provide a richer set of behaviors.

The rate that tokens accumulate depends on whether the slice has a *reservation* or a *share*. A slice with a reservation accumulates tokens at its reserved rate: for example, a slice with a 10% reservation gets 100 tokens per second, since a token entitles it to run a process for one millisecond. The default share is actually a small reservation, providing the slice with 32 tokens every second, or 3% of the total capacity.

The main difference between reservations and shares occurs when there are runnable processes but no slice has enough tokens to run: in this case, slices with shares are given priority over slices with reservations. First, if there is a runnable slice with shares, tokens are given out fairly to all slices with shares (i.e., in proportion to the number of shares each slice has) until one can run. If there are no runnable slices with shares, then tokens are given out fairly to slices with reservations. The end result is that the CPU capacity is effectively partitioned between the two classes of slices: slices with reservations get what they've reserved, and slices with shares split the unreserved capacity of the machine proportionally.

CoMon indicates that the average PlanetLab node has its CPU usage pegged at 100% all the time. However, fair sharing means that an individual slice can still ob-
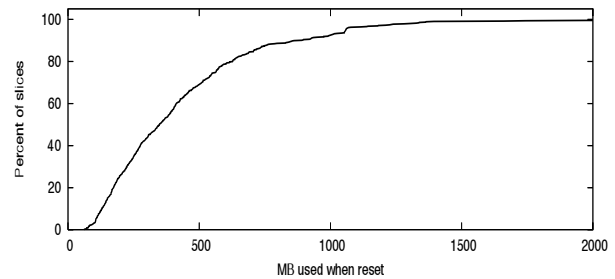
tain a significant percentage of the CPU. Figure 3 shows, by quartile, the CPU availability across PlanetLab, obtained by periodically running a spinloop in the CoMon slice and observing how much CPU it receives. The data shows large amounts of CPU available on PlanetLab: at least 10% of the CPU is available on 75% of nodes, at least 20% CPU on 50% of nodes, and at least 40% CPU on 25% of nodes.

4.2.2 Memory

Memory is a particularly scarce resource on PlanetLab, and we were faced with with chosing between four designs. One is the default Linux behavior, which either kernel panics or randomly kills a process when memory becomes scarce. This clearly does not result in graceful degradation. A second is to statically allocate a fixed amount of memory to each slice. Given that there are up to 90 active VMs on a node, this would imply an impractically small 10MB allocation for each VM on the typical node with 1GB of memory. A third option is to explicitly allocate memory to live VMs, and reclaim memory from inactive VMs. This implies the need for a control mechanism, but globally synchronizing such a mechanism across PlanetLab (i.e., to suspend a slice) is problematic at fine-grained time scales. The fourth option is to dynamically allocate memory to VMs on demand, and react in a more predictable way when memory is scarce.

We elected the fourth option, implementing a simple watchdog daemon, called pl_mom, that resets the slice consuming the most physical memory when swap has almost filled. This penalizes the memory hog while keeping the system running for everyone else.

Although pl_mom was noticably active when first deployed—as users learned to not keep log files in memory and to avoid default heap sizes—it now typically resets an average of 3-4 VMs per day, with higher rates during heavy usage (e.g., major conference deadlines). For example, 200 VMs were reset during the two week runup to the OSDI deadline. We note, however, that roughly one-third of these resets were on under-provisioned nodes (i.e., nodes with less than 1GB of memory).

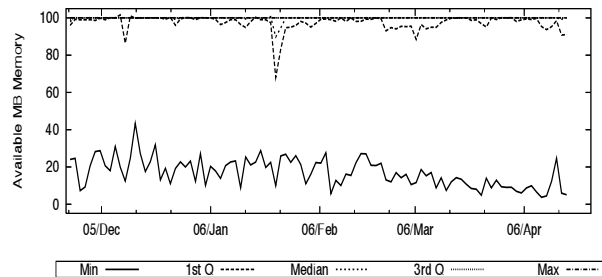Figure 4 shows the cumulative distribution function of

Figure 5: Memory availability on PlanetLab

(a) Transmit bandwidth in Kb/s, by quartile

(b) Receive bandwidth in Kb/s, by quartile

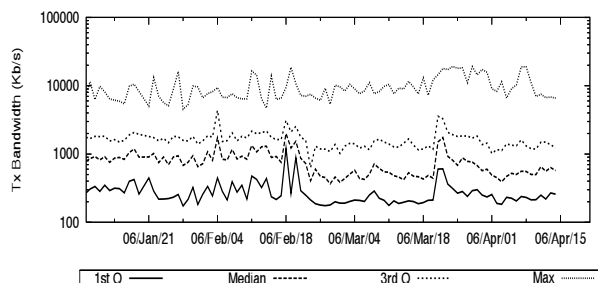Figure 6: Sustained network rates on PlanetLab

how much physical memory individual VMs were consuming when they were reset between November 2004 and April 2006. We note that about 10% of the resets (corresponding largely to the first 10% of the distribution) occurred on nodes with less than 1GB memory, where memory pressure was tighter. Over 80% of all resets had been allocated at least 128MB. Half of all resets occurred when the slice was using more than 400MB of memory, which on a shared platform like PlanetLab indicates either a memory leak or poor experiment design (e.g., a large in-memory logfile).

Figure 5 shows CoMon's estimate of how many MB of memory are available on each PlanetLab node. CoMon estimates available memory by allocating 100MB, touching random pages periodically, and then observing the size of the in-memory working set over time. This serves as a gauge of memory pressure, since if physical memory is exhausted and another slice allocates memory, these pages would be swapped out. The CoMon data shows that a slice can keep a 100MB working set in memory on at least 75% of the nodes (since only the minimum and first quartile line are really visible), so it appears that there is not as much memory pressure on PlanetLab as we expected. This also reinforces our intuition that pl_mom resets slices mainly on nodes with too little memory or when the slice's application has a memory leak.
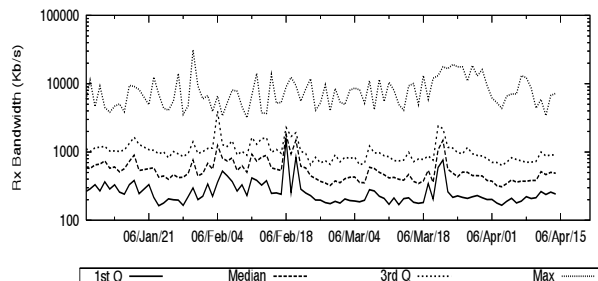
4.2.3 Bandwidth

Hosting sites can cap the maximum rate at which the local PlanetLab nodes can send data. PlanetLab fairly shares the bandwidth under the cap among slices, using Linux's Hierarchical Token Bucket traffic filter [15]. The node bandwidth cap allows sites to limit the *peak* rate at which nodes send data so that PlanetLab slices cannot completely saturate the site's outgoing links.

The *sustained* rate of each slice is limited by the pl_mom watchdog daemon. The daemon allows each slice to send a quota of bytes each day at the node's cap rate, and if the slice exceeds its quota, it imposes a much smaller cap for the rest of the day. For example, if the slice's quota is 16GB/day, then this corresponds to a sustained rate of 1.5Mbps; once the slice sends more than

16GB, it is capped at 1.5Mbps until midnight GMT. The goal is to allow most slices to burst data at the node's cap rate, but prevents slices that are sending large amounts of data from badly abusing the site's local resources.

There are two weaknesses of PlanetLab's bandwidth capping approach. First, some sites pay for bandwidth based on the total amount of traffic they generate per month, and so they need to control the node's sustained bandwidth rather than the peak. As mentioned, pl_mom limits sustained bandwidth, but it operates on a per-slice (rather than per-node) basis and cannot currently be controlled by the sites. Second, PlanetLab does not currently cap *incoming* bandwidth. Therefore, PlanetLab nodes can still saturate a bottleneck link by downloading large amounts of data. We are currently investigating ways to fix both of these limitations.

Figure 6 shows, by quartile, the sustained rates at which traffic is sent and received on PlanetLab nodes since January 2006. These are calculated as the sums of the average transmit and receive rates for all the slices of the machine over the last 15 minutes. Note that the y axis is logarithmic, and the Minimum line is omitted from the graph. The typical PlanetLab node transmits about 1Mb/s and receives 500Kb/s, corresponding to about 10.8GB/day sent and 5.4GB/day received. These numbers are well below the typical node bandwidth cap of 10Mb/s. On the other hand, some PlanetLab nodes do actually have sustained rates of 10Mb/s both ways.
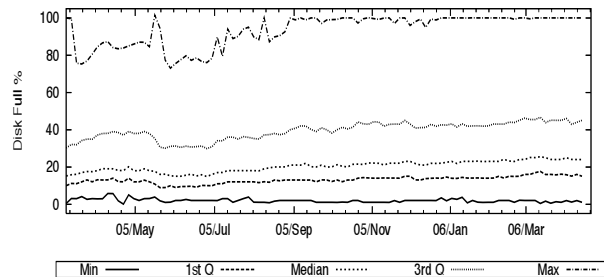
Figure 7: Disk usage, by quartile, on PlanetLab

Figure 8: RTT CDF on network, overlay with and without SCHED_RR

4.2.4 Disk

PlanetLab nodes do not provide permanent storage: data is not backed up, and any node may be reinstalled without warning. Services adapt to this environment by treating disk storage as a cache and storing permanent data elsewhere, or else replicating data on multiple nodes. Still, a PlanetLab node that runs out of disk space is essentially useless. In our experience, disk space is usually exhausted by runaway log files written by poorly-designed experiments. This problem was mitigated, but not entirely solved, by the introduction of per-slice disk quotas in June 2005. The default quota is 5GB, with larger quotas granted on a case-by-case basis.

Figure 7 shows, by quartile, the disk utilization on PlanetLab. The visible dip shortly after May 2005 is when quotas were introduced. We note that, though disk utilization grows steadily over time, 75% of Planetlab nodes still have at least 50% of the disks free. Some PlanetLab nodes do occasionally experience full disks, but most are old nodes that do not meet the current system requirements.

4.2.5 Jitter

CPU scheduling latency can be a serious problem for some PlanetLab applications. For example, in a packet forwarding overlay, the time between when a packet arrives and when the packet forwarding process runs will appear as added network latency to the overlay clients. Likewise, many network measurement applications assume low scheduling latency in order to produce precisely spaced packet trains. Many measurement applications can cope with latency by knowing which samples to trust and which must be discarded, as described in [29]. Scheduling latency is more problematic for routing overlays, which may have to drop incoming packets.

A simple experiment indicates how scheduling latency can affect applications on PlanetLab. We deploy a packet forwarding overlay, constructed using the Click modular software router [13], on six Planetlab nodes co-located at Abilene PoPs between Washington, D.C. and Seattle. O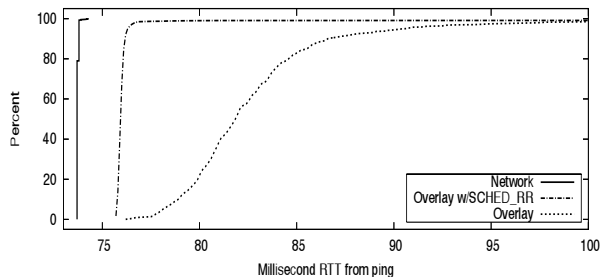ur experiment then uses `ping` packets to compare the RTT between the Seattle and D.C. nodes on the network and on the six-hop overlay. Each of the six PlanetLab nodes running our overlay had load averages between 2 and 5, and between 5 and 8 live slices, during the experiment. We observe that the network RTT between the two nodes is a constant 74ms over 1000 pings, while the overlay RTT varies between 76ms and 135ms. Figure 8 shows the CDF of RTTs for the network (leftmost curve) and the overlay (rightmost curve). The overlay CDF has a long tail that is chopped off at 100ms in the graph.

There are several reasons why the overlay could have its CPU scheduling latency increased, including: (1) if another task is running when a packet arrives, Click must wait to forward the packet until the running task blocks or exhausts its timeslice; (2) if Click is trying to use more than its "fair share", or exceeds its CPU guarantee, then its token bucket CPU filter will run out of tokens and it will be removed from the runqueue until it acquires enough tokens to run; (3) even though the Click process may be runnable, the Linux CPU scheduler may still decide to schedule a different process; and (4) interrupts and other kernel activity may preempt the Click process or otherwise prevent it from running.

We can attack the first three sources of latency using existing scheduling mechanisms in PlanetLab. First, we give the overlay slice a CPU reservation to ensure that it will never run out of tokens during our experiment. Second, we use `chrt` to run the Click process on each machine with the SCHED_RR scheduling policy, so that it will immediately jump to the head of the runqueue and preempt any running task. The Proper service described in Section 3.5 enables our slice to run the privileged `chrt` command on each PlanetLab node.

The middle curve in Figure 8 shows the results of rerunning our experiment with these new CPU scheduling parameters. The overhead of the Click overlay, around 3ms, is clearly visible as the difference between the two left-most curves. In the new experiment, about 98% of overlay RTTs are within 3ms of the underlying network RTT, and 99% are within 6ms. These CPU scheduling mechanisms are employed by *PL-VINI*, the VINI (VIr-

tual Network Infrastructure) prototype implemented on PlanetLab, to reduce latency in an overlay network as an artifact of CPU scheduling delay [4].

We note two things. First, the obstacle to making this solution available on PlanetLab is primarily one of policy—choosing which slices should get CPU reservations and bumps to the head of the runqueue, since it is not possible to reduce everyone's latency on a heavily loaded system. We plan to offer this option to short-term experiments via the Sirius brokerage service, but long-running routing overlays will need to be handled on a case-by-case basis. Second, while our approach can provide low latency to the Click forwarder in our experiment 99% of the time, it does not completely solve the latency problem. We hypothesize that the remaining CPU scheduling jitter is due to the fourth source of latency identified earlier, i.e., kernel activity. If so, we may be able to further reduce it by enabling kernel preemption, a feature already available in the Linux 2.6 kernel.

4.2.6 Remarks

Note that only limited conclusions can be drawn from the fact that there is unused capacity available on PlanetLab nodes. Users are adapting to the behavior of the system (including electing to not use it) and they are writing services that adapt to the available resources. It is impossible to know how many resources would have been used, even by the same workload, had more been available. However, the data does document that PlanetLab's fair share approach is behaving as expected.

5 Operational Stability

The need to maintain a stable system, while at the same time evolving it based on user experience, has been a major complication in designing PlanetLab. This section outlines the general strategies we adopted, and presents data that documents our successes and failures.

5.1 Strategies

There is no easy way to continually evolve a system that is experiencing heavy use. Upgrades are potentially disruptive for at least two reasons: (1) new features introduce new bugs, and (2) interface changes force users to upgrade their applications. To deal with this situation, we adopted three general strategies.

First, we kept PlanetLab's control plane (i.e., the services outlined in Section 3) orthogonal from the OS. This meant that nearly all of the interface changes to the system affected only those slices running management services; the vast majority of users were able to program to a relatively stable Linux API. In retrospect this is an obvious design principle, but when the project began, we believed our main task was to define a new OS interface tailored for wide-area services. In fact, the one example where we deviated from this principle—by changing the

socket API to support *safe raw sockets* [3]—proved to be an operational diaster because the PlanetLab OS looked enough like Linux that any small deviation caused disproportionate confusion.

Second, we leveraged existing software whereever possible. This was for three reasons: (1) to improve the stability of the system; (2) to lower the barrier-to-entry for the user community; and (3) to reduce the amount of new code we had to implement and maintain. This last point cannot be stressed enough. Even modest changes to existing software packages have to be tracked as those packages are updated over time. In our eagerness to reuse rather than invent, we made some mistakes, the most notable of which is documented in the next subsection.

Third, we adopted a well-established practice of rolling new releases out incrementally. This is for the obvious reason—to build confidence that the new release actually worked under realistic loads before updating all nodes—but also for a reason that's somewhat unique to PlanetLab: some long-running services maintain persistent data repositories, but doing so depends on a minimal number of copies being available at any time. Updates that reboot nodes must happen incrementally if long-running storage services are to survive.

Note that while few would argue with these principles—and it is undoubtedly the case that we would have struggled had we not adhered to them—our experience is that many other factors (some unexpected) had a significant impact on the stability of the system. The rest of this section reports on these operational experiences.

5.2 Node Stability

We now chronicle our experience operating and evolving PlanetLab. Figure 9 illustrates the availability of PlanetLab nodes from September 2004 through April 2006, as inferred from CoMon. The bottom line indicates the PlanetLab nodes that have been up continuously for the last 30 days (stable nodes), the middle line is the count of nodes that came online within the last 30 days, and the top line is all registered PlanetLab nodes. Note that the difference between the bottom and middle lines represents the "churn" of PlanetLab over a month's time; and the difference between the middle and top lines indciates the number of nodes that are offline. The vertical lines in Figure 9 are important dates, and the letters at the top of the graph let us refer to the intervals between the dates.

There have clearly been problems providing the community with a stable system. Figure 9 illustrates several reasons for this:

- Sometimes instability stems from the community stressing the system in new ways. In Figure 9, interval **A** is the run-up to the NSDI'05 deadline. During this time, heavy use combined with memory leaks in
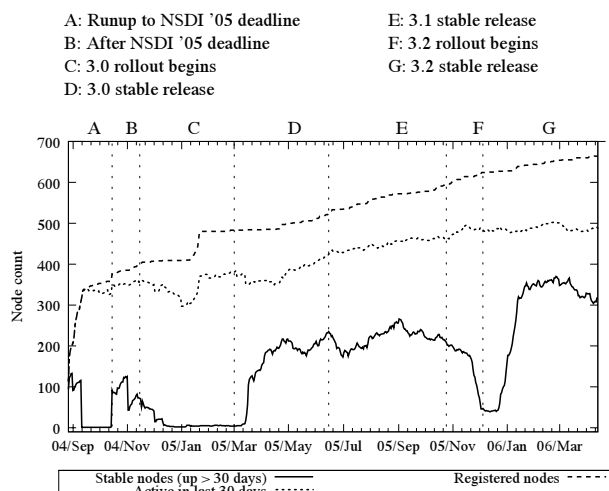
Figure 9: Node Availability

some experiments caused kernel panics due to Out-of-Memory errors. This is the common behavior of Linux when the system runs out of memory and swap space. pl_mom (Section 4.2.2) was introduced in response to this experience.

- Software upgrades that require a reboot obviously effect the set of stable nodes (e.g., intervals **C** and **D**), but installing buggy software has a longer-term affect on stability. Interval **C** shows the release of PlanetLab 3.0. Although the release had undergone extensive off-line testing, it had bugs and a relatively long period of instability followed. PlanetLab was still usable during this period but nodes rebooted at least once per month.

- The pl_mom watchdog is not perfect. There is a slight dip in the number of stable core nodes (bottom line) in interval **D**, when about 50 nodes were rebooted because of a slice with a fast memory leak; as memory pressure was already high on those nodes, pl_mom could not detect and reset the slice before the nodes ran out of memory.

We note, however, that the 3.2 software release from late December 2005 is the best so far in terms of stability: as of February 2006, about two-thirds of active Planet-Lab nodes have been up for at least a month. We attribute most of this to abandoning CKRM in favor of the Vservers native resource management framework and a new CPU scheduler.

One surprising fact to emerge from Figure 9 is that a lot of PlanetLab nodes are dead (denoted by the difference between the top and middle lines). Research organizations gain access to PlanetLab simply by hooking up two machines at their local site. This formula for growth

has worked quite well: a low barrier-to-entry provided the right incentive to grow PlanetLab. However, there have never been well-defined incentives for sites to keep nodes online. Providing such incentives is obviously the right thing to do, but we note that that the majority of the off-line nodes are at sites that no longer have active slices—and at the time of this writing only 12 sites had slices but no functioning nodes—so it's not clear what incentive will work.

Now that we have reached a fairly stable system, it becomes interesting to study the "churn" of nodes that are active yet are not included in the stable core. We find it useful to differentiate between three categories of nodes: those that came up that day (and stayed up), those that went down that day (and stayed down), and those that have rebooted at least once. Our experience is that, on a typical day, about 2 nodes come up, about 2 nodes go down, and about 4 nodes reboot. On 10% of days, at least 6 nodes come up or go down, and at least 8 nodes reboot.

Looking at the archives of the support and planetlab-users mailing lists, we are able to identify the most common reasons nodes come up or go down: (1) a site takes its nodes offline to move them or change their network configuration, (2) a site takes its nodes offine in response to a security incident, (3) a site accidently changes a network configuration that renders its nodes unreachable, or (4) a node goes offline due to a hardware failure. The last is the most common reason for nodes being down for an extended period of time; the third reason is the most frustrating aspect of operating a system that embeds its nodes in over 300 independent IT organizations.

Understanding the relative frequency of different sorts of site events may be important for designers of other large-scale distributed systems; this is a topic for further study.

5.3 Security Complaints

Of the operational issues that PlanetLab faces, responding to security complaints is perhaps the most interesting, if only because of what they say about the current state of the Internet. We comment on three particular types of complaints.

The most common complaints are the result of IDS alerts. One frequent scenario corresponds to a perceived DoS attack. These are sometimes triggered by a poorly designed experiment (in which case the responsible researchers are notified and expected to take corrective action), but they are more likely to be triggered by totally innocent behavior (e.g., 3 unsolicited UDP packets have triggered the threat of legal action). In other cases, the alerts are triggered by simplistic signitures for malware that could not be running on our Linux-based environment. In general, we observe that any traffic that devi-

ates from a rather narrow range of acceptable behavior is increasingly viewed as suspect, which makes innovating with new types of network services a challenge.

An increasingly common type of complaint comes from home users monitoring their firewall logs. They see connections to PlanetLab nodes that they do not recognize, assume PlanetLab has installed spyware on their machines, and demand that it be removed. In reality, they have unknowingly used a service (e.g., a CDN) that has imposed itself between them and a server. Receiving packets from a location service that also probes the client to select the most appropriate PlanetLab node to service a request only exacerbates the situation [35, 9]. The take-away is that even individual users are becoming increasingly security-senstive (if less security-sophisticated than their professional counterparts), which makes the task of deploying alternative services increasingly problematic.

Finally, PlanetLab nodes are sometimes identified as the source or sink of illegal content. In reality, the content is only cached on the node by a slice running a CDN service, but an overlay node looks like an end node to the rest of the Internet. PlanetLab staff use PlanetFlow to identify the responsible slice, which in turn typically maintains a log that can be used to identify the ultimate source or destination. This information is passed along to the authorities, when appropriate. While many hosting sites are justifiably gun-shy about such complaints, the main lesson we have learned is that trying to police content is not a viable solution. The appropriate approach is to be responsive and cooperative when complaints are raised.

6 Discussion

Perhaps the most fundamental issue in PlanetLab's design is how to manage trust in the face of pressure to decentralize the system, where decentralization is motivated by the desire to (1) give owners autonomous control over their nodes and (2) give third-party service developers the flexibility they need to innovate.

At one end of the spectrum, individual organizations could establish bilateral agreements with those organizations that they trust, and with which they are willing to peer. The problem with such an approach is that reaching the critical mass needed to foster a large-scale deployment has always proved difficult. PlanetLab started at the other end of the spectrum by centralizing trust in a single intermediary—PLC—and it is our contention that doing so was necessary to getting PlanetLab off the ground. To compensate, the plan was to decentralize the system through two other means: (1) users would delegate the right to manage aspects of their slices to third-party services, and (2) owners would make resource allocation decisions for their nodes. This approach has had mixed success, but it is important to ask if these limitations are fun-damental or simply a matter of execution.

With respect to owner autonomy, all sites are allowed to set bandwidth caps on their nodes, and sites that have contributed more than the two-node minimum required to join PlanetLab are allowed to give excess resources to favored slices, including brokerage services that redistribute those resources to others. In theory, sites are also allowed to blacklist slices they do not want running locally (e.g., because they violate the local AUP), but we have purposely not advertised this capability in an effort to "unionize" the research community: take all of our experiments, even the risky ones, or take none of them. (As a compromise, some slices voluntarily do not run on certain sites so as to not force the issue.) The interface by which owners express their wishes is clunky (and sometimes involves assistance from the PlanetLab staff), but there does not seem to be any architectural reason why this approach cannot provide whatever control over resource allocation that owners require (modulo meeting requirements for joining PlanetLab in the first place).

With respect to third-party management services, success has been much more mixed. There have been some successes—Stork, Sirius, and CoMon being the most notable examples—but this issue is a contentious one in the PlanetLab developer community. There are many possible explanations, including there being few incentives and many costs to providing 24/7 management services; users preferring to roll their own management utilities rather than learn a third-party service that doesn't exactly satisfy their needs; and the API being too much of a moving target to support third-party development efforts.

While these possibilities provide interesting fodder for debate, there is a fundamental issue of whether the centralized trust model impacts the ability to deploy third-party management services. For those services that require privileged access on a node (see Section 3.5) the answer is yes—the PLC support staff must configure Proper to grant the necessary privilege(s). While in practice such privileges have been granted in all cases that have not violated PlanetLab's underlying trust assumptions or jeopardized the stability of the operational system, this is clearly a limitation of the architecture.

Note that choice is not just limited to what management services the central authority approves, but also to what capabilities are included in the core system—e.g., whether each node runs Linux, Windows, or Xen. Clearly, a truly scalable system cannot depend on a single trusted entity making these decisions. This is, in fact, the motivation for evolving PlanetLab to the point that it can support federation. To foster federation we have put together a software distribution, called MyPLC, that allows anyone to create their own private PlanetLab, and potentially federate that PlanetLab with others (including the current "public" PlanetLab).

This returns us to the original issue of centralized versus decentralized trust. The overriding lesson of PlanetLab is that a centralized trust model was essential to achieving some level of critical mass—which in turn allowed us to learn enough about the design space to define a candidate minimal interface for federation—but that it is only by federating autonomous instances that the system will truly scale. Private PlanetLabs will still need bilateral peering agreements with each other, but there will also be the option of individual PlanetLabs scaling internally to non-trivial sizes. In other words, the combination of bilateral agreements and trusted intermediaries allows for flexible aggregation of trust.

7 Conclusions

Building PlanetLab has been a unique experience. Rather than leveraging a new mechanism or algorithm, it has required a synthesis of carefully selected ideas. Rather than being based on a pre-conceived design and validated with controlled experiments, it has been shaped and proven through real-world usage. Rather than be designed to function within a single organization, it is a large-scale distributed system that must be cognizant of its place in a multi-organization world. Finally, rather than having to satisfy only quantifiable technical objectives, its success has depended on providing various communities with the right incentives and being equally responsive to conflicting and difficult-to-measure requirements.

Acknowledgments

References

[1] ANNAPUREDDY, S., FREEDMAN, M. J., AND MAZIERES, D. Shark: Scaling File Servers via Cooperative Caching. In *Proc. 2nd NSDI* (Boston, MA, May 2005).

[2] AUYOUNG, A., CHUN, B., NG, C., PARKES, D., SHNEIDMAN, J., SNOEREN, A., AND VAHDAT, A. Bellagio: An Economic-Based Resource Allocation System for PlanetLab. http://bellagio.ucsd.edu/about.php.

[3] BAVIER, A., BOWMAN, M., CULLER, D., CHUN, B., KARLIN, S., MUIR, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating System Support for Planetary-Scale Network Services. In *Proc. 1st NSDI* (San Francisco, CA, Mar 2004).

[4] BAVIER, A., FEAMSTER, N., HUANG, M., PETERSON, L., AND REXFORD, J. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proc. SIGCOMM 2006* (Pisa, Italy, Sep 2006).

[5] BRETT, P., KNAUERHASE, R., BOWMAN, M., ADAMS, R., NATARAJ, A., SEDAYAO, J., AND SPINDEL, M. A Shared Global Event Propagation System to Enable Next Generation Distributed Services. In *Proc. 1st WORLDS* (San Francisco, CA, Dec 2004).

[6] CLARK, D. D. The Design Philosophy of the DARPA Internet Protocols. In *Proc. SIGCOMM '88* (Stanford, CA, Aug 1988), pp. 106–114.

[7] DAVID LOWENTHAL. Sirius: A Calendar Service for PlanetLab. http://snowball.cs.uga.edu/dkl/pslogin.php.

[8] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIERES, D. Democratizing content publication with Coral. In *Proc. 1st NSDI* (San Francisco, CA, Mar 2004).

[9] FREEDMAN, M. J., LAKSHMINARAYANAN, K., AND MAZIERES, D. OASIS: Anycast for Any Service. In *Proc. 3rd NSDI* (San Jose, CA, May 2006).

[10] FU, Y., CHASE, J., CHUN, B., SCHWAB, S., AND VAHDAT, A. SHARP: An Architecture for Secure Resource Peering. In *Proc. 19th SOSP* (Lake George, NY, Oct 2003).

[11] HUANG, M., BAVIER, A., AND PETERSON, L. PlanetFlow: Maintaining Accountability for Network Services. *ACM SIGOPS Operating Systems Review 40*, 1 (Jan 2006).

[12] JUSTON CAPPOS AND JOHN HARTMAN. Stork: A Software Packagement Management Service for PlanetLab. http://www.cs.arizona.edu/stork.

[13] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Transactions on Computer Systems 18*, 3 (Aug 2000), 263–297.

[14] LAI, K., RASMUSSON, L., ADAR, E., SORKIN, S., ZHANG, L., AND HUBERMAN, B. A. Tycoon: An Implemention of a Distributed Market-Based Resource Allocation System. Tech. Rep. arXiv:cs.DC/0412038, HP Labs, Palo Alto, CA, USA, Dec. 2004.

[15] LINUX ADVANCED ROUTING AND TRAFFIC CONTROL. http://lartc.org/.

[16] LINUX VSERVERS PROJECT. http://linux-vserver.org/.

[17] MUIR, S., PETERSON, L., FIUCZYNSKI, M., CAPPOS, J., AND HARTMAN, J. Privileged Operations in the PlanetLab Virtualised Environment. *SIGOPS Operating Systems Review 40*, 1 (2006), 75–88.

[18] OPPENHEIMER, D., ALBRECH, J., PATTERSON, D., AND VAHDAT, A. Distributed Resource Discovery on PlanetLab with SWORD. In *Proc. 1st WORLDS* (San Francisco, CA, 2004).

[19] PARK, K., AND PAI, V. S. Scale and Performance in the CoBlitz Large-File Distribution Service. In *Proc. 3rd NSDI* (San Jose, CA, May 2006).

[20] PARK, K., PAI, V. S., PETERSON, L. L., AND WANG, Z. CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups. In *Proc. 6th OSDI* (San Francisco, CA, Dec 2004), pp. 199–214.

[21] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. HotNets–I* (Princeton, NJ, Oct 2002).

[22] PETERSON, L., BAVIER, A., FIUCZYNSKI, M., MUIR, S., AND ROSCOE, T. PlanetLab Architecture: An Overview. Tech. Rep. PDN–06–031, PlanetLab Consortium, Apr 2006.

[23] RAMASUBRAMANIAN, V., PETERSON, R., AND SIRER, E. G. Corona: A High Performance Publish-Subscribe System for the World Wide Web. In *Proc. 3rd NSDI* (San Jose, CA, May 2006).

[24] RAMASUBRAMANIAN, V., AND SIRER, E. G. Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays. In *Proc. 1st NSDI* (San Francisco, CA, Mar 2004), pp. 99–112.

[25] RAMASUBRAMANIAN, V., AND SIRER, E. G. The Design and Implementation of a Next Generation Name Service for the Internet. In *Proc. SIGCOMM 2004* (Portland, OR, Aug 2004), pp. 331–342.

[26] RHEA, S., GODFREY, B., KARP, B., KUBIATOWICZ, J., RATNASAMY, S., SHENKER, S., STOICA, I., AND YU, H. OpenDHT: A Public DHT Service and its Uses. In *Proc. SIGCOMM 2005* (Philadelphia, PA, Aug 2005), pp. 73–84.

[27] RITCHIE, D. M., AND THOMPSON, K. The UNIX Time-Sharing System. *Communications of the ACM 17*, 7 (Jul 1974), 365–375.

[28] RYAN HUEBSCH. PlanetLab Application Manager. `http://appmanager.berkeley.intel-research.net/`.

[29] SPRING, N., BAVIER, A., PETERSON, L., AND PAI, V. S. Using PlanetLab for Network Research: Myths, Realities, and Best Practices. In *Proc. 2nd WORLDS* (San Francisco, CA, Dec 2005).

[30] SPRING, N., WETHERALL, D., AND ANDERSON, T. Scriptroute: A Public Internet Measurement Facility. In *Proc. 4th USITS* (Seattle, WA, Mar 2003).

[31] VIVEK PAI AND KYOUNGSOO PARK. CoMon: A Monitoring Infrastructure for PlanetLab. `http://comon.cs.princeton.edu`.

[32] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. 1st OSDI* (Monterey, CA, Nov 1994), pp. 1–11.

[33] WANG, L., PARK, K., PANG, R., PAI, V. S., AND PETERSON, L. L. Reliability and Security in the CoDeeN Content Distribution Network. In *Proc. USENIX '04* (Boston, MA, Jun 2004).

[34] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. 5th OSDI* (Boston, MA, Dec 2002), pp. 255–270.

[35] WONG, B., SLIVKINS, A., AND SIRER, E. G. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In *Proc. SIGCOMM 2005* (Philadelphia, PA, Aug 2005).

[36] ZHANG, M., ZHANG, C., PAI, V. S., PETERSON, L. L., AND WANG, R. Y. PlanetSeer: Internet Path Failure Monitoring and Characterization in Wide-Area Services. In *Proc. 6th OSDI* (San Francisco, CA, Dec 2004), pp. 167–182.