The Design Principles of PlanetLab

Larry Peterson
Princeton University

Timothy Roscoe
Intel Research – Berkeley

ABSTRACT

PlanetLab is a geographically distributed platform for deploying, evaluating, and accessing planetary-scale network services. PlanetLab is a shared community effort by a large international group of researchers, each of whom gets access to one or more isolated *slices* of PlanetLab's global resources. Because we deployed PlanetLab and started supporting users before we fully understood what its architecture would be, being able to evolve the system became a requirement. This paper examines the set of *design principles* that guided this evolution. Some of these principles were explicit at the project outset, and others have become crystallized as the platform has developed.

1. INTRODUCTION

PlanetLab is a globally distributed computing platform shared, built, and maintained by a community of researchers at 300 sites in more than 30 countries. In exchange for hosting a small number of nodes (servers), participants obtain access to a share of resources across the entire platform for deploying and evaluating planetary-scale network services [4]. In addition, the platform itself, including the essential services required to operate it, is a communal design work in progress and an interesting research area in its own right.

In this paper we concentrate on the design principles of PlanetLab. By "design principles", we mean the rules we have come to recognize and formulate that guide our decisions about how to put the platform together. In contrast, we use the term "architecture" to denote the composition of the platform itself, in other words the end result of these decisions. We do not describe PlanetLab's architecture in this paper other than to illustrate the consequences of our design principles. The architecture of the platform at time of writing is described in several PlanetLab Design Notes (PDNs) and academic publications [1].

Of course, the boundary between design principles and high-level architectural features is somewhat arbitrary. Nonetheless, we have found it useful to try to tease the two apart. It is comparatively rare for the documentation of large systems to include an attempt to reflect on the thought processes of its architects, David Clark's description of the Internet design philosophy [2] being one notable example.

The evolving design of PlanetLab has been an attempt at principled pragmatism. The principles we present here did not, in general, predate the implementation of PlanetLab, though some were explicit from the outset. Instead, they have co-evolved with the architecture itself, and thus we expect them, like the architecture itself, to continue to change over time.

2. GOALS

Underlying the design principles are the high-level goals of PlanetLab. From the beginning [4], we have identified three:

- to provide a platform for researchers to experiment with planetary-scale network services;
- to provide a platform for novel network services to be deployed and serve a real user community; and
- to catalyze the evolution of the Internet into a service-oriented architecture.

It should be clear that these goals are highly synergistic and reinforcing: early experiments with ideas lead to the deployment of new network services, and the availability of a rich set of network services effectively changes the nature of the Internet. There are, however, subtle tensions between the three. For example, a platform that supports only short-term experiments would likely be designed differently than one that must also support continuously-running services. We hope to support both workloads, but with a bias towards the latter. Similarly, a platform that supports a collection of services developed by the research community need not necessarily consider the full range of scalability, security, and autonomy issues that must be addressed by a platform which aspires to become *the* conduit through which users interact with the Internet. The balance-point between these two goals is difficult to

define: we must continue to push the scalability, security, robustness, and decentralization that an Internet-scale architecture requires, but at the same time, ensure that the evolution of the architecture is driven by the requirements of the running system rather some idealized vision.

3. TERMINOLOGY

The design principles guiding the evolution of PlanetLab as a platform have been formulated at the same time as the architecture itself has crystallized. Consequently, while nominally independent of the current architecture of PlanetLab, it is convenient to describe and illustrate the design principles in the context of PlanetLab as it currently exists. For that reason, we describe briefly here how PlanetLab's architecture looks today; readers familiar with PlanetLab internals may observe that the current architecture does not always adhere to the principles we describe here.

PlanetLab users who wish to deploy applications acquire a *slice*, which is a collection of virtual machines (VMs) spread around the world. The VMs are implemented on physical machines by some OS mechanism or virtual machine monitor (VMM), and controlled by another entity, the *node manager*, which is responsible for creating and destroying slices. We sometimes call use the term *sliver* to refer to the instantiation of a slice on a give node. There are also special *infrastructure* slices which perform essential functions on each node (such as providing a local site administrator's interface to the node).

Collectively, the node managers and infrastructure services, together with the (currently centralized) account management and node installation functions, form the *control plane* of PlanetLab.

Of course, PlanetLab is at least two things: the entire ensemble of contributed services and running applications on the platform, and the narrower set of maintained facilities that support the entire ensemble. By "architecture" in this paper we mean the structure of the core subset, maintained by the PlanetLab support team. However, we feel the design principles we outline here are applicable to other services above this layer in the platform.

4. DISTRIBUTED VIRTUALIZATION

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

> **Slices are underspecified.** While PlanetLab must prescribe low-level facilities for creating, manipulating, and destroying slices, much of the process is left unspecified, and can be performed by a variety of other services.

By giving slices as much flexibility as possible in defining a suitable environment for a service, as well as the process by which that environment is instantiated, we minimize the extent to which future users are constrained in what they do with the platform. We also hopefully encourage users to implement functions useful to other slices. For example, PlanetLab does not provide tunnels that connect a slice's constituent VMs into an overlay, but allows a slice-specific overlay to be created, either by the slice itself, or by another service which creates and populates the slice in the first place.

Similarly, the actual contents of a sliver—within the "box" of the VM—are of little concern to the PlanetLab. For example, it should not matter whether a sliver is running in a Java virtual machine (JVM) or is written in assembly language, and if it is a JVM, it is up to the slice to decide whether it uses version 1.5.0 or 1.4.2. The platform provides the means by which slices can install whatever software they need, but little more.

PlanetLab aims to isolate services and applications from each other, thereby maintaining the illusion that each service runs on a distributed set of private machines. The slice, or strictly speaking the sliver, is the basic abstraction for this.

> **The unit of isolation is the sliver.** Whatever base functionality the platform provides, it must be possible for the platform to isolate slices from each other as they invoke this functionality. The platform must strive to minimize the effect one slice can have on another.

This principle has three corollaries. First, the platform must deliver isolation of slivers, by allocating and scheduling node resources, partitioning or contextualizing system namespaces, and enforcing stability and security, between slivers sharing a node. Early methods for achieving this in PlanetLab are discussed in [1].

Second, care must be exercised when multiplexing any VMM-provided resources between slices, not only low-level resource abstractions like CPU time. It is most straightforward to achieve isolation by keeping the underlying VMM as "thin" as possible, and providing most of the system functionality within a slice; for example, using Xen [**?**] as a virtual machine monitor and running a complete operating system kernel in each slice. Where an implementation uses a more high-level VMM, such as the Linux/VServer kernel [1], it is

important to avoid as far as possible crosstalk between applications competing for kernel resources (such as the protocol stack, routing tables, filing system datastructures, open file descriptors, and so on.).

Third, it should not concern the platform how resources allocated to a sliver are multiplexed among the various activities of the sliver, for example by intra-slice thread scheduling policies. It is, however, desirable that the platform provide in the "execution environment" it presents to each virtual machine, the mechanisms to allow a slice to perform this internal multiplexing effectively.

5. UNBUNDLED MANAGEMENT

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

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

> **Support only local abstractions directly in the OS.** In other words, the only abstractions that the low-level VMM should deal with are local to the node. Implement all global (network-wide) abstractions by infrastructure services.

Perhaps the most notable example of this principle is slices themselves: they are a global abstraction, and neither the VMM on a PlanetLab node nor the node manager on the node deal with anything other than single virtual machines. The slice abstraction itself (as a distributed collection of VMs) is implemented by slice creation services.

We want to maximize the opportunity for services to compete with each other on a level playing field. In other words, rather than have a single privileged application controlling a particular aspect of the OS, the PlanetLab OS potentially supports many such management services. One implication of this interface being sharable is that it must be well-defined, explicitly exposing the state of the underlying OS. In contrast, the interface between an OS and a privileged control program running in user space is often ad hoc since the control program is, in effect, an extension of the OS that happens to run in user space.

> **Make all interfaces exported by the OS or control plane sharable.** Any functionality which has to be implemented by a unique piece of code should be accessible by multiple infrastructure services, none of which should require unique privilege.

Again, this principle applies to slice creation as much as to other infrastructure services. While the low-level mechanism for creating VMs on a node resides in the node manager, there can be (and are) multiple slice creation services that can instantiate slivers on a node.

The bottom line is that OS design often faces a tension between implementing functionality in the kernel and running it in user space, the objective often being to minimize kernel code. Like many VMM architectures, the PlanetLab OS faces an additional, but analogous, tension between what can run in a slice or VM, and functionality (such as slice user authentication) that requires extra privilege or access but is not part of the kernel. In addition, there is a third aspect to the problem that is peculiar to PlanetLab: functionality that can be implemented by parallel, competing subsystems, versus mechanisms which by their very nature can only be implemented once (such as bootstrapping slice creation). The PlanetLab OS strives to minimize the latter, but there remains a small core of non-kernel functionality that has to be unique on a node.

6. CHAIN OF RESPONSIBILITY

The way slices interact with the rest of the world has has turned out to be important factor in PlanetLab's design, much more so than we thought at the inception of the project. It is also a novel requirement, a consequence of PlanetLab's giving applications so many points-of-presence on the network.

> **Each interaction between PlanetLab and the rest of the network must be attributable to a PlanetLab user.** We must always consider PlanetLab's interaction with the rest of the network: unlike many systems projects, PlanetLab is and has always been inextricably embedded in the "real" Internet, and its behavior has consequently often been felt in the Internet. Moreover, each such interaction with the real Internet can,

and should be attributed to the responsible user.

Effectively limiting and auditing legitimate users has been as significant an issue as securing the PlanetLab to prevent malicious users from hijacking machines. Experience shows that the Internet, and the design of intrusion detection systems in particular, is highly sensitive to the kinds of traffic that experimental planetary-scale services tend to generate, and this has had to be reflected in the design of PlanetLab's filtering, rate limiting, and packet auditing functionality.

However, our experience is that that simply limiting slices is not sufficient since even a single unexpected or unwanted packet can trigger an incident report. Thus, an important responsibility of PlanetLab is to preserve the *chain of responsibility* among all the relevant principals. That is, it must be possible to map externally visible activity (e.g., a transmitted packet) to the users responsible for that packet. Note that PlanetLab does not attempt to eliminate the possibility that bad things might happen, it just requires that the system be able to identify the responsible party when something does go wrong.

Preserving the chain of responsibility is not just a matter of being able to respond to security complaints; it is also essential to preserving the implicit trust relationships. Consider that on the one hand, nearly 300 autonomous organizations have contributed nodes to PlanetLab. They each require autonomous control over the nodes they own. On the other hand, 375 research groups want to deploy their services across PlanetLab. The node owners need assurances that these services will not be disruptive. Clearly, establishing 300×375 pairwise trust relationships is an unmanageable task: a researcher would have to obtain permission to create VMs on nodes owned by 300 organizations, while a hosting site would need to approve requests for use of its nodes from 375 independent research groups [5].

A well-understood way to reduce such a $N \times N$ problem into a $N + N$ problem is to use a trusted intermediary. The *PlanetLab Consortium* (PLC) is one such trusted intermediary: Node owners trust PLC to manage the behavior of slivers that run on their nodes while preserving their autonomy, and researchers (slice users) trust it to provide access to a set of nodes that are capable of hosting their services. In general, we observe:

> **Keep explicit the trust relationships between node owners, authorities, and slice users.** PlanetLab is a decentralized system that spans multiple autonomous organizations. To have long-term viability, it must identify the critical trust relationships among the principals, and provide mechanisms that give these principals the control and assurances they require.

For example, as described elsewhere [3], PlanetLab provides an auditing mechanism that ensures that the chain of responsibility is preserved. It also provides mechanisms that allows principals the ability to dictate how their resources are used.

7. EVOLUTION VS. CLEAN SLATES

PlanetLab has never aimed to start from a "clean slate". Moreover, architectural features of PlanetLab have always been designed with a view to their eventual replacement.

This is partly because the research community was ready to use PlanetLab the moment the first nodes were deployed; the lengthy process of designing a completely new architecture from scratch was never an option. As a result, the first version of PlanetLab had to be built quickly from preexisting ideas and technologies. We had a strong intuition that much of what we originally designed would have to be reworked as we gained experience with the system, since for for the first version there was very little usage data or requirements data available.

The approach of evolving preexisting technology as opposed to pursuing a clean slate design is also motivated by our experience with previous testbeds, in which application writers exhibited two strong biases: (1) they are seldom willing to port their applications to a new API, and (2) they expect a full-featured system rather than a minimalist API tuned for someone else's OS research agenda. Again, since PlanetLab's value is defined by the applications that run on it, a clean slate design was simply not an option.

While the concept of built-in obsolescence of building blocks has been explicit in PlanetLab from the very beginning, it is only recently that the consequences of these decisions have been codified as design principles:

> **Avoid "clean slate" designs.** When designing a large piece of infrastructure that is closely related to existing technology (in the way PlanetLab is deeply embedded in the current Internet and commodity operating systems), a clean slate, top-down architecture cannot be developed in the time available, once the need for the infrastructure is recognized. In contrast, a bottom-up, evolutionary design will immediately gain momentum.

However, any new architecture, whether bottom-up or top-down, is always in danger of atrophy, particularly if it is not designed with the evolutionary process in mind.

> **Design the architecture with an open-ended view of future evolution.** Design decisions must always be taken with a view to the future evolution of the architecture, in possibly unforeseen ways. Architectural options should be kept as open as possible.

This second principle is subtle, in that it goes beyond mere extensibility of interfaces. We expect few current features of PlanetLab's architecture to be recognizable in a few years time. Above all, PlanetLab has been designed to *efficiently evolve*.

Two other principles follow from these in the specific context of PlanetLab's evolution. They are not orthogonal, but rather overlap and mutually reinforce each other. The first is a direct consequence of eschewing the "clean slate" approach in favor of producing a usable but highly evolvable system as soon as possible.

> **Leverage existing software and hardware infrastructure.** Wherever possible, avoid modifying existing packages (and thereafter having to track changes) if they can be used as-is.

Adapting rather than simply using an off-the-shelf software package takes time, and raises the question of who now supports the modified package. Typically, the PlanetLab development team has to track changes to the mainstream package, and keep their own modified version in sync. This can result in considerable extra work, on an ongoing basis, and is to be avoided if possible.

For example, PlanetLab makes use of standard, readily available software such as OpenSSH, Linux, XML-RPC libraries and bindings, and a variety of Unix utilities. When PlanetLab does modify existing software, such as patching the Linux kernel, this leads to an increased maintenance burden as we must then track changes in the kernel more closely and port our modifications. However, we mitigate this burden somewhat by using patches which are themselves maintained by other groups wherever possible (most notably the Vservers patch), and by using loadable kernel modules (VNET).

When adding new functionality to a PlanetLab node over and above that provided by the node operating system, the question inevitably arises as to where the functionality should be implemented: as a service, in the control plane, or as a kernel extension.

> **Implement any new system function at the "highest" level possible.** Running a service in a slice with limited privileged capabilities is preferred to a slice with widespread privileges, which in turn is preferred to augmenting the node manager, all of which are preferable to adding the function to the VMM or operating system. This leads to a PlanetLab version of the principle of least privilege: Privileged slices should be granted the minimal privileges necessary to support the desired behavior. They should not be granted blanket superuser privileges.

8. OS AND CONTROL PLANE

Maintaining a split between the operating system and the control plane enables us to separate the execution environment of an application, which we keep as non-PlanetLab-specific as possible, from the interface to PlanetLab itself.

> **Keep the Control Plane and the Operating System orthogonal.** Don't pollute the OS interface by adding new functionality to it, when this can be added to the out-of-band control plane interface instead.

The control plane (essentially, the node manager and associated services) appears to a VM to be separate from the OS proper. Adding functionality to the control plane interface means that new features or capabilities are added to a PlanetLab node in an OS-independent way. A good example of this is the use of PlanetLab sensors to obtain information about node status.

This also means that the operating system interface changes little, facilitating cross-development and testing between PlanetLab nodes, clusters, and users' development machines. We can extend this principle further to a version of the traditional idea of "least surprise":

> **Use existing interface semantics as far as possible:** Any OS operation, initiated in a VM, should have an effect that is as close as possible to the effect it would have on the corresponding dedicate machine, subject to the external policies limiting the capabilities of the slice. No operations should be added to the PlanetLab control plane or OS interfaces if the desired functionality can already be accessed through the existing OS interfaces.

For example, access to raw sockets on PlanetLab was originally restricted to administrative programs. Normal slices used a new, PlanetLab-specific address family to open "safe raw sockets", which could only send or receive packets that might have been sent or received on an existing socket owned by the slice. A better approach has now been implemented: safe raw sockets are accessed in exactly the same away as "real" raw sockets are. Whether the client program receives all packets depends on the privileges granted to the slice.

A final principle, closely related to those above, has been identified as PlanetLab begins to consider both

a transition to a more heterogeneous deployment (in OS and processor terms), and the federation issues that arise when a slice can span multiple PlanetLab-like platforms.

> **Don't tackle porting issues.** Use the orthogonality of control plane and OS to ensure that the overhead of porting an application is the same when running on PlanetLab or on a native operating system.

This captures PlanetLab's position on portability of applications and support for heterogeneity moving forward, even though at present PlanetLab provides a single OS/instruction set environment. Specifically, the cost of porting an application from one operating system or processor architecture to another should be the same on PlanetLab as it would be in a conventional environment.

Put another way, we deliberately hide nothing of the operating system or processor architecture under a layer of abstractions. Porting between environments is orthogonal to porting to or from PlanetLab, since features specific to PlanetLab are added in an OS-neutral way to the control plane interface. Our goal is to decouple those aspects of the PlanetLab API that are unique to PlanetLab from the underlying execution environment.

9. ACKNOWLEDGEMENTS

The development of PlanetLab, and of these principles, has been a highly communal effort. We would like to acknowledge the contribution of the large number of people (too many to mention here) who have shaped our way of thinking about PlanetLab and continue to do so as the platform evolves.

10. REFERENCES

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

[2] D. D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proceedings of the SIGCOMM '88 Symposium*, pages 106–114, Stanford, 1988.

[3] M. Huang, A. Bavier, and L. Peterson. PlanetFlow: Maintaining Accountability for Network Services. In *Operating Systems Review*, Jan. 2006.

[4] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. of ACM HotNets-I*, Princeton, New Jersey, Oct. 2002.

[5] L. Peterson, A. Bavier, M. Fiuczynski, S. Muir, and T. Roscoe. Towards a Comprehensive PlanetLab Architecture. Technical Report PDN–05–030, PlanetLab Consortium, June 2005.