



PLANETLAB

PlanetLab Phase 1: Transition to an Isolation Kernel

Larry Peterson
Princeton University

Timothy Roscoe
Intel Research – Berkeley

PDN-02-003
September 2002

Status: Final Version.

PlanetLab Phase 1: Transition to an Isolation Kernel

Larry Peterson and Timothy Roscoe

1 Introduction

A serious problem facing any testbed is that it is difficult to simultaneously do the research needed to create an effective testbed, and use the testbed as a platform for writing applications. Users require a stable platform, which is at odds with the need to do research on the platform. To make matters worse, such research often results in new APIs, requiring that applications be written from scratch. The problem has a strong temporal aspect: planned obsolescence of building blocks is a key component of the design philosophy. We need a way to evolve the platform and its interfaces over time without unduly impacting the users.

This document addresses the problem of how to evolve the kernel component of PlanetLab toward a strong notion of an *isolation kernel*, while maintaining the system as operational and usable by researchers for both experimentation and long-term deployment. We discuss the goals of the transition, and outline two strategies for moving in this direction.

2 Factors

We begin by identifying four factors that influence the design of a kernel for PlanetLab. The factors are ranked in order of importance, starting with the most crucial factor:

Security: The kernel must provide strong inter-service security. A service should not be able to obtain unauthorized access (read or write) to the state of another service. This should be the case even if the service has been compromised by malicious third party, or if the service itself is buggy or intentionally malicious. Consequently, the kernel itself should be immune to attack from the services that run over it.

Ideally, we would like to assure a trusted code base consisting of the kernel and a small number of infrastructure services.

Familiar API: Programmers want to use a familiar API to write their applications. Mandating a new API for PlanetLab services is an unacceptable burden for users. PlanetLab has already made the decision to maintain a POSIX-like API, initially that provided by Linux.

This position can be refined, however. A new API can be provided alongside or as an alternative to the existing one, particularly if it offers significant benefits in code simplicity, or efficient access to new functionality. One example might be controlled access to fast forwarding hardware. Another might be an alternative execution environment based on events and upcalls rather than processes.

Finally, we note that the API that users code to need not coincide with the boundaries in the system at which protection is enforced, or resources allocated.

Isolation: We need to keep services from impacting each other. We use the term isolation with regard to resources (CPU, network bandwidth, disk space, etc.) used by a service. Thus, isolation is distinguished from security in that it is concerned with performance interactions rather than access to information.

We are less concerned in this document with specific mechanisms or policies (such as resource reservations, shares, quotas) as with the high level goal of minimizing any adverse performance impact one service might have on another.

Performance: By performance, we mean the absolute performance of services running on PlanetLab nodes. Consequently, this is a difficult notion to quantify and is rather application-specific. It also depends on the workload for an application. Some services might be asymmetric, with server-like components running on some nodes and router-like components running on others.

However, performance is a factor in evaluating kernel design decisions, and at the very least, we would like to avoid having to apologize for the performance of PlanetLab nodes.

Performance has several aspects for a platform like PlanetLab, where a diverse range of network services will be hosted. Two of the most important are latency of access to a resource, and the scalability of the platform as a whole as the number of services increases.

There is a well-known tension between *efficiency* (meaning the fraction of the resources of a node are used by services, what might also be termed *yield*) and the degree of isolation obtained between services.

3 Strategies

We are considering two general strategies, distinguished by whether the interfaces for programming and isolation coincide or appear at different places in the system.

3.1 Decoupled Isolation and Programming Interfaces

Isolation kernels like Denali [?] and Xenoservers [?], provide a low-level isolation interface which closely resembles virtualization of the hardware. Operating systems such as Linux, BSD, and Windows XP can be ported to this virtual machine, an operation which is greatly simplified by the similarity between the virtual machine architecture and the “real” hardware the operating systems were originally developed for. Alternatively, a lightweight and simplified “library operating system” can be used, which is more tailored to network services and exploits the fact that each service has the complete virtual machine to itself.

A related but more heavyweight approach is to completely virtualize the physical hardware by means of a monitor like VMware [?] or Plex86 [?]. In this case, the operating system binaries can be installed directly.

These schemes perform protection and resource allocation at the low-level isolation interface, while users write to the API of the operating system running in the virtual machine.

We call this approach Plan A, and we revisit the four factors:

Security: Isolation kernels have a compelling security story. Since the machine is virtualized above a very simple interface, the security properties of this interface can be relatively easily verified. If they hold, the degree of protection offered to each service should be comparable with that obtained by running each service on a separate machine.

In practice, there are some complications. The interface is complicated by the presence of virtual devices to be used by the guest operating system, and by the design issues in the networking model between virtual machines on

the same physical hardware. However, the situation is still much simpler from a security standpoint than a complex, high-level interface like POSIX.

Familiar API: Since the API is largely decoupled from the isolation interface, providing a desired API is a matter of providing a suitable library or porting a suitable operating system. This allows great flexibility: each service can potentially use a different API, and new APIs can be introduced without impact on existing services.

Isolation: In principle, resource isolation is provided by the isolation kernel. This has the merit of simplicity, in that the isolation kernel does not need to take into account dependencies between schedulable entities in the system, since the resource principals it is dealing with correspond to individual services.

Performance: Virtual machine approaches such as isolation kernels have some impact on performance, particularly in terms of the number of VMs supportable on a single physical machine. This impact is high for complete virtualization approaches like VMware [?], which can consequently support only a small number of virtual machines per processor.

In the case of systems like Denali and Xenoservers, the performance implications when running a large number of virtual machines concurrently are a subject of active research. Denali has demonstrated high scalability (thousands of services per machine), but at the cost of turning off memory protection between services.

The principle performance bottleneck appears to be the virtual memory system, firstly in the lack of sharing of operating system data and code between VMs in naive implementations, and secondly in the duplication of page tables between the guest operating system and the virtual machine monitor.

Recently published work from VMware[?] demonstrates considerable benefits of sharing pages between VMs based on hashes of their contents. It is also possible that partial virtualization approaches like Denali and Xenoservers, where the guest operating system must be ported to the virtual machine, can offer novel virtual MMU designs which facilitate sharing between VMs and more efficient paging by cooperating with the virtual machine monitor.

Another area of potential impact for services running over isolation kernels is in packet forwarding for services which need to perform a routing function, particularly with regard to forwarding latency. Fortunately, packet forwarding can be pushed into hardware. However, this pushes it outside the scope of our system. An open issue is the design of a virtual machine for services

that have a significant packet forwarding component.

A final area of concern is the overhead involved in service composition, that is, allowing one service to build on another. Two services running in separate virtual machines can be composed only by connecting them over the network. The extent to which this connection can be optimized when the two services reside on the same node is yet to be demonstrated.

3.2 Combined Isolation and Programming Interfaces

Traditional Unix-like operating systems such as Linux do not distinguish between the interface at which resource allocation and protection is applied, and the system call interface used by application developers.

An alternative to the isolation kernel approach is to take an existing operating system, and augment it with functionality to provide better security and resource isolation between services running over it. The most attractive way of doing this includes virtualizing the kernel (rather than the hardware) in the style of Ensim [?], Linux/Vservers [?], BSD Jail [?], and User-Mode Linux [?]. A number of resource isolation solutions exist complementary to kernel virtualization, such as SILK [?] and QLinux [?] proportional share mechanisms.

We call this second approach Plan B, and we revisit the four factors:

Security: While the virtualization of the kernel can simplify security issues between services, the complexity of the Linux system call interface means that security is still problematic. It should be noted, however, that systems like Vservers do address this issue (by means of Linux “capabilities”), and that there are security modifications to Linux available to provide mandatory access control on all kernel objects.

Familiar API: A familiar API is implied in Plan B. However, providing alternative APIs requires work. Note that while writing a compatibility library for, say, FreeBSD over Linux may make sense for supporting a single application, in general this problem is very hard and requires a large engineering effort (see the Wine project’s attempts to implement the Win32 API over Linux, or the Cygwin approach to the inverse).

Isolation: Resource isolation in commodity operating systems is by now relatively well-understood. The SILK module for Linux provides most, if not all, the functionality we might require and can be configured to use a VServer se-

curity context as a resource principal. A number of other approaches and mechanisms exist, many of them open source.

Performance: Performance is generally better with virtualized kernel systems in comparison with hardware virtualization, generally because most of the operating system kernel and support binaries are shared. Intel Research at Berkeley has successfully run 500 Linux VServers on a PlanetLab node with a total cost of 9MB disk space per VServer, with comparable performance to running in vanilla Unix processes.

This strategy does not shy away from putting functionality in the kernel. Consequently, should performance considerations dictate, it is not out of the question that certain common routing services could run in the kernel to avoid the performance impact of virtualization. In general, this approach seems more amenable to service composition.

4 Transitional Issues

Plan B in the form of Linux VServers and SILK is in alpha test right now, and so we plan to deploy this as soon as it is ready, while Plan A depends on research happening first. Should Plan A prove feasible (and desirable), transition then becomes the issue.

One approach for the transition is to re-implement the Plan B approach in a virtual machine over Plan A. This will most likely require a port of VServers and SILK to whatever virtual machines appear as candidates for the Plan A implementation. Plan A nodes can then be run alongside the “native” Plan B nodes for the transition period. Alternatively, Linux/SILK could run VMWare, and host experimental versions of Denali or Xenoserver during the transitional period. In general, the ability to run new isolation kernels in a virtual machine over existing ones is highly desirable as PlanetLab evolves.

We have at least two competing efforts to implement Plan A. Ideally, we could agree on a common isolation interface, and allow multiple implementations. In fact, there might be three such implementations: (1) Denali, (2) Xenoserver, (3) Linux/SILK. The latter effectively augments the Linux programming interface with the isolation interface. Nodes might elect to run different versions depending on their workloads and trust assumptions; e.g., Linux/SILK might run on a future “router” that allows only certain parties to run “forwarding services”.

Both implementations of Plan A still need fine-grain isolation mechanisms. It may be that aspects of SILK can be reused in the context of these kernels.

Version Control Information

CVS version information:

```
$Id: transition.tex,v 1.4 2002/09/21 17:19:52 troscoe Exp $
```

This paper was run off October 21, 2002.