Underware: An Exokernel for the Internet?

David Irwin, Jeff Chase, Laura Grit, Aydan Yumerefendi, and Jeannie Albrecht[‡]
Duke University *University of California, San Diego*[‡]
{irwin,chase,grit,aydan}@cs.duke.edu, jalbrecht@cs.ucsd.edu

Abstract

The principles for designing and building shared distributed computing environments are still evolving: today, there exist a myriad of environments targeting different applications serving different user communities. NSF's GENI initiative proposes a new shared environment to serve as an open testbed for designing and building a Future Internet. The design of GENI, along with other distributed computing environments, must confront core OS issues of isolation, coordinated multiplexing of hardware resources, and abstractions for distributed computing. This paper draws parallels to the extensible OS "kernel wars" of the past, and considers how architectural lessons from that time apply to an Internet OS. Our view echoes, in key respects, the principles of Exokernel a decade ago: the common core of an Internet OS should concern itself narrowly with physical resource management. We refer to this common core as *underware* to emphasize that it runs *underneath* existing programming environments for distributed computing.

1 Introduction

NSF's GENI initiative [1] envisions an international network testbed facility with a software framework to multiplex network resources—including clusters, storage, and other edge resources—among experimental prototypes for a new generation of network applications and services. GENI is both a shared testbed and a software architecture that confronts core OS issues of isolation, coordinated multiplexing of hardware resources, and abstractions for distributed computing. GENI calls for a fresh look at how well-studied OS principles apply to this new environment: how do we architect an OS for the future Internet? It must provide a platform for planetary-scale applications not yet imagined, absorb technologies not yet invented, and sustain decades of growth and "disruptive innovation" without fracturing or collapsing under its own weight. It is crucial that we get it right.

In this paper, we draw parallels to the extensible OS "kernel wars" that played out in the 1980s and 1990s (and yet lives on). We consider how architectural lessons from that time may apply to an Internet OS. The landscape

[1]`http://www.geni.net`

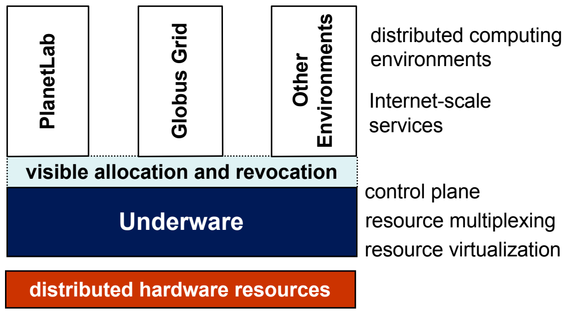

Figure 1: Distributed computing environments, such as PlanetLab and Globus Grids, are built above machine-level resource abstractions obtained from infrastructure providers through the underware control plane.

of Internet computing today is analogous in key ways. There are multiple existing platforms for resource sharing and distributed computing with large user communities (e.g., PlanetLab [5] and grid systems [13]). We are again searching for an evolutionary path that preserves and enhances existing programming environments, accommodates innovation in key system functions, and preserves the unity and coherence of a common hardware platform. Finding the elemental common abstractions or "wasp waist" of the architecture that supports all of its moving parts remains an elusive goal.

Our thinking leads us to a view that echoes, in key respects, the principles of Exokernel [11] a decade ago: the common core of an Internet OS should concern itself narrowly with physical resource management. An Internet OS must host multiple programming environments (application OSes) built above machine-level resource abstractions, and multiplex hardware resources among those environments, as shown in Figure 1. We contend that an extensible, abstraction-free resource management layer is a cornerstone of Internet OSes. We refer to this layer as *underware* to emphasize that it runs *underneath* the programming environments for distributed computing, which are implemented in node OSes, application-level servers, and "middleware" layers that mediate between applications and the node OS.

We are experimenting with an underware control plane architecture to allocate shared networked resources con-

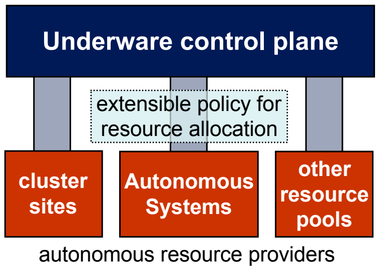

Figure 2: Policies for allocating hardware resources are outside the underware resource control plane. Resource providers control the policies by which resources are used.

tributed by autonomous providers, and a prototype called Shirako [17]. Shirako reflects core principles also present in Exokernel: in particular, resource allocation and revocation are explicit and visible to allow hosted environments to control their allocations and adapt to changing resources. In previous work we have illustrated how to use the Shirako control plane to host multiple independent Globus grid instances on shared networked clusters, with adaptive, dynamic resource control [25]. For this paper we have used Shirako to donate nodes to PlanetLab and to host independent instances of PlanetLab (MyPLC). We summarize what it took to do this, why it is important, and some issues it raises.

Underware should support extensible policies for resource allocation: this flexibility is more important for an Internet OS than it was for a node OS, where the focus on extensible resource management was feared to be "leading us astray" [10]. An Internet OS coordinates resources from multiple autonomous resource owners who must have a means to specify, implement, and harmonize policies for local resource management, and control the terms by which they contribute resources and the ways those resources are used. Figure 2 shows that policies reside outside the underware control plane and within the resource owners. This policy control is essential for the system to be sustainable: it must provide incentives to contribute and adequate protection for contributors. In addition, effective resource control is required for mission-critical applications (e.g., Internet 911 and disaster response). At the same time, resource management is coarser-grained, so the overhead concerns that made the problem so difficult for extensible kernels are not as daunting.

Shirako is designed to work with different technologies for virtualization of underlying resources, e.g., virtual machines. We may consider these "underware" as well, but in this paper we focus on the control plane architecture, and leave as an open question how these principles might apply to virtualization technologies.

2 A Brief History of the Kernel Wars

Operating systems serve at least two purposes: they manage physical resources, and they support programming abstractions (e.g., processes, threads, files) that are the building blocks of applications. These functions are closely intertwined. In hindsight, the sweep of research on microkernels and extensible kernels can be seen as an effort to separate them to the maximum extent possible.

Early microkernel research (e.g., [1]) decoupled the programming API from the kernel, which was limited to a set of low-level, neutral abstractions suitable for implementing multiple OS APIs or "personalities" in servers running in user mode. Among the claimed benefits of this approach were a minimal (and hence more secure and reliable) trusted core and support for multiple replaceable OS personalities on a common hardware base.

Some microkernel systems extended the kernel API to allow OS subsystems more control over resource management. For example, Mach supported upcall interfaces from the kernel to user-level subsystems, enabling them to interpose on basic OS functions such as paging. Despite the problem of liability inversion [16], external pagers were used to build several advanced application environments. However, they fell short of allowing control over physical resource allocation (e.g., victim selection for page replacement).

With the kernel interface liberated from the need to support an easily usable API, several projects devised interfaces that would allow OS subsystems—or at least libraries—full control over the physical resources allocated to them by the kernel, e.g., Scheduler Activations [4] and Exokernel [11]. The approach was to expose machine-level abstractions (e.g., processors) at the kernel interface, and leave the guest to implement programming abstractions (e.g., threads) above them. Exokernel proposed to "exterminate OS abstractions" within the kernel. A key principle of these systems was *visible allocation and revocation*: the core should expose resource allocation decisions to the OS instances so they can adapt and notify any affected applications.

A further step was to allow user-level components to extend low-level physical resource arbitration across multiple competing applications. The frequent interactions between the kernel and its extensions introduced difficult safety and performance concerns. One solution was to inject extensions into the kernel, with various mechanisms to verify and sandbox them (e.g., SPIN [7]). These ideas were never widely deployed for a number of reasons [10]. In retrospect, one reason is that interposing on fine-grained resource management is expensive, and node resources became cheap enough to be dedicated or partitioned at a coarse grain.

Recently, virtual machines have become popular for partitioning resources at a coarser grain. Multiplexing entire OS instances with little or no modification, and allocating resources between them using a machine-level abstraction (VMs bound to slivers), significantly reduces the overhead required to interpose on resource management compared with multiplexing at the process-level. In fact, Hand et al. [16] argue that virtual machine hypervisors are simply a point in the microkernel design space that avoid many of the problems inherent to microkernels (e.g., liability inversion, poor IPC performance) because they multiplex entire OS instances that do not depend on each other and communicate sparingly along narrow well-defined interfaces.

Hypervisors and virtualization are key building blocks for resource-controlled underware. We adopt the Exokernel principles—physical resource management based on machine-level abstractions with visible allocation and revocation of resources—-as a basis for managing resources at a coarser grain (e.g., VM slivers) in large infrastructures.

3 Internet Operating Systems

As with node kernels, the architecture of Internet operating systems involves two different but intertwined issues: resource allocation and programming model. We contend that any viable architecture for an Internet OS should be free of resource allocation policy. There is much discussion about allocation policy (e.g., economic models and virtual currencies?), but there is no one-size-fits-all policy. Thus it must be possible to change these policies or use different allocation approaches in different parts of the infrastructure.

The architecture should also avoid constraining the programming model. Researchers have put forward several competing systems; we may view them as "OS personalities" for distributed computing on shared resources, with models and abstractions that differ subtly. We want to run multiple environments over common hardware, in such a way that resource owners can redeploy hardware easily from one environment to another. This "pluralist" approach has a familiar justification (e.g., [8, 12]). Given the wide diversity of needs among users, it is useless to wait for one environment to "win". Also, pluralism enables innovation and customization—we want a thousand flowers to bloom, not just one.

These goals are similar to the goals for microkernels and extensible kernels. They are more significant now because the resources are networked, so there must be a common set of protocols to enable resource sharing across the system, subject to policy control. The policies are now defined by multiple stakeholders who operate or manage different parts of the infrastructure, and not just by an OS kernel and guest.

In our approach, resource control plane *underware* allocates resources to different platform "flavors" or "personalities". In contrast to middleware, underware is software that resides, conceptually, underneath the operating systems. Its primary role is to coordinate and control all software, including the operating system as well as any middleware, that runs on the hardware allocated to the different environments (slices). As with exokernel (and hypervisors), the guest environments are free to implement their own higher-level abstractions and services (Figure 1) above machine-level resource abstractions provided by the underware.

3.1 Examples: Globus and PlanetLab

We consider two well-known examples of distributed computing systems: grid computing with Globus and PlanetLab. These platforms are "moving targets" that are evolving to incorporate new technology and new capabilities. Among the key architectural issues are: who controls the hardware allocation, how is it exposed, and what components are replaceable.

Globus grids support familiar abstractions: they allow users to run their jobs and workflows on somebody else's operating system. Distribution is seen as a necessary evil to use remote resources. Globus is middleware that runs above operating systems installed by the resource owners; Globus sacrifices control over the OS to enable some degree of heterogeneity. It derives from the "metacomputing" idea, which introduces standard protocols and API libraries to weld these diverse resources and diverse operating systems into a uniform execution platform. Globus also provides services to establish a common notion of identity, a common distributed middleware for routing jobs and scheduling them on local resources.

PlanetLab is designed to support large-scale network services. They run continously, but on varying resource allotments, and they need to adapt. PlanetLab mandates uniformity of operating systems and physical node configurations across the system. It provides abstractions and system services (e.g., distributed virtualization, resource discovery, monitoring) to enable deployment of widely distributed applications that control their own communication patterns and overlay topology.

3.2 Architectural Choices

Our concern is not with the programming models and features, but with the core architectural choices for managing physical resources and trust.

Globus provides a uniform programming model, but because it is middleware, and does not control operating systems or hardware resources, it has limited control over resource management. QoS, reservations, and flexible site control are important for grid computing, but they have been elusive in the practice. The problem is that

Globus can only control when to submit jobs to queues or operating systems; it cannot predict or control what resources are allocated by the lower layer, unless the lower layer provides those hooks (in our proposal those functions are provided by the underware).

In contrast, PlanetLab is a distributed operating system designed to control a set of dedicated hardware resources, which it controls and images centrally. Much of the common API is provided by a Linux kernel flavor mandated by PlanetLab. The abstractions are OS abstractions: local file name space, processes, and a Unix/ssh security model with keys controlled by PlanetLab Central (PLC). Much of the research focus has been on extending the OS to virtualize these abstractions to isolate multiple virtual servers running on the same physical server [5].

Architecturally, PlanetLab has made similar choices to Mach. Since it exports OS-level abstractions, PlanetLab can also support a wide range of hosted environments, including rich middleware environments such as Globus. PlanetLab has taken the first step of extensibility from the microkernel progression with its emphasis on "unbundled management" of infrastructure services. Unbundled management defines key system interfaces to enable alternative implementations of foundational infrastructure services outside of the system's trusted core. That enables evolution, and a competitive market for extensions.

But like Mach, PlanetLab retains basic resource management in the core and does not expose its resource allocation choices or allow significant control over policy. It unbundles some resource management functions to subsystems, but only with the consent of the central point of trust (PLC). For example, a contributing resource provider site cannot change the resource allocation policy for its own resources without the consent of PLC.

PlanetLab established a "best effort" resource management model as a fundamental architectural choice. At the earliest meetings it was observed that any stronger model requires admission control, which is a barrier to use. PlanetLab papers have also argued that it is a good choice for PlanetLab because it is underprovisioned, claiming that conservative resource allocation is undesirable because it wastes resources, and exposing machine abstractions or multiple OS instances (e.g., using Xen instead of vservers) consumes too much memory. It has also been stated that the best-effort model mirrors the reality of the current Internet, in which edge resources and transit are unreliable. A best-effort model forces applications to evolve the means to adapt reactively to whatever confronts them, which is a good skill for any long-running network service to have in a dangerous world.

This philosophy is in keeping with Butler Lampson's hints to keep it simple and keep secrets, i.e., don't make promises to applications that you might not keep. But at the same time PlanetLab users do have expectations about stability, which was presented as a goal and quantified in the recent PlanetLab paper [24]. As one example, it is considered bad manners to withdraw a node from PlanetLab without warning, but PlanetLab's abstractions do not provide a means to deliver such a warning other than to broadcast on an e-mail list. In addition to complicating maintenance, this limitation discourages cluster sites from contributing resources to PlanetLab on a temporary basis. PlanetLab has recently added programmatic APIs to contribute and withdraw machines, but warn against their casual use. More importantly, although best-effort is a reasonable policy choice and should not be excluded, it is too limiting as a basis for an Internet operating system. Some applications require predictable service quality, which must be supported "at the bottom or not at all" [9].

4 Shirako: Clean Underware

The Shirako control plane [17] runs as a collection of servers on behalf of resource providers, hosted environments (e.g., guests such as Globus or PlanetLab), and brokers that encapsulate resource arbitration policy. Guests interact with resource servers to obtain *lease* contracts for resources with a specified type, attributes, and duration. Shirako does not constrain the nature of the resources or contracts—however, we have focused our efforts on allocating cluster resources instantiated as blocks of physical or virtual machines. Users are free to select kernels or install software on the nodes to integrate them into the guest environment. Shirako is compatible with self-certifying secure tickets and accountable delegation following the SHARP framework [14].

Leases are a form of visible resource allocation and revocation advocated with Exokernel. The Shirako leasing core upcalls prologue/epilogue actions in the guest on lease state transitions to notify it of changes to the resource allotment. The upcalls provide a hook for middleware environments or other guest platforms to manage their resources. We have used Shirako to host dynamic instances of several systems:

PlanetLab. MyPLC is a downloadable PlanetLab for creating and managing new PlanetLab instances. We completed the integration to run PlanetLab kernels on Xen virtual machines; this required minor modifications to the PlanetLab boot process along with a recent patch to enable *kexec* in Xen-capable kernels. We wrote a small wrapper that leases resources from a Shirako broker and instantiates a MyPLC central server and one or more PL hosts. It uses the new PLC API to add and remove nodes from the MyPLC instance. This system could be used to add and remove local machines from the public PlanetLab.

Globus grids. In recent work we showed how Globus-

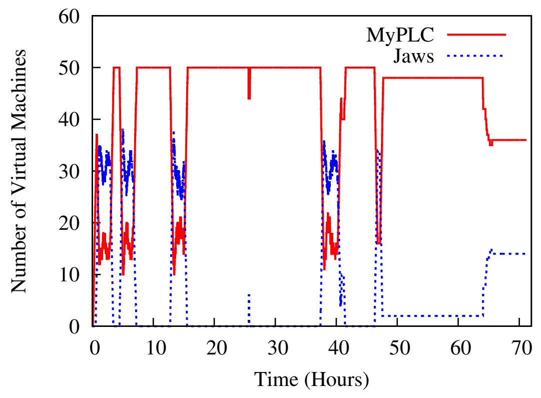

Figure 3: MyPLC and Jaws obtaining Xen virtual machines from the Shirako underware. The underware manages sharing of hardware resources between MyPLC and Jaws with a simple policy: MyPLC gets any resources not used by Jaws.

based grids can lease resources from networked clusters through Shirako [25]. The underware layer makes it possible for a controller to support dynamic provisioning, advance reservations, and site control over resource sharing without modifying Globus itself.

Jaws. Jaws is a new batch scheduler that uses Shirako to run batch jobs in their own resource-isolated Xen virtual machines; it uses the resource isolation provided by Xen to actively learn models mapping performance (e.g., job runtime) to resource profiles [26]. Jaws is an example of a novel service that requires explicit resource allocation/revocation and strict resource isolation.

4.1 Illustration

To illustrate the role of underware we crafted a simple demonstration using instances of MyPLC and Jaws on a shared cluster. PlanetLab is not sufficient as a foundation for Jaws: resources can neither be isolated nor scheduled—Jaws requires strong resource isolation to learn accurate performance models.

Figure 3 depicts the number of VMs allocated to Shirako-enabled MyPLC and Jaws in the days leading up to this deadline. Jaws users submitted a few bursts of jobs to collect data for another impending paper deadline, and received priority to reclaim nodes from MyPLC as their leases expire, according to local policy.

The key points from this demonstration are that (1) PlanetLab is not a sufficient platform for all environments—especially those requiring strict resource isolation (see also recent work on VINI [6])—and (2) we were able to, relatively easily, port PlanetLab's platform to use the Shirako underware to share hardware resources with a different platform.

While simple, this illustration raises several issues. The next subsection discusses the impact of a dynamic PlanetLab that provisions servers on-demand to meet excess load.

4.2 Discussion

An underware-enabled PlanetLab requires new logic to determine *when* to request new resources, *how much* to request, and for *how long*. The request policy may derive from monitoring the testbed (i.e., using CoMon [23]) to detect when resources are scarce, as often occurs around conference deadlines, and pro-actively adding resources to satisfy users. Monitoring resource usage is a good first-order approximation of resource demands, but as resource utilization nears 100% it is impossible to quantify the need for additional capacity.

PlanetLab differs from grid environments that can use queue size, along with data from previously executed jobs, to determine the need for additional capacity. As an initial solution PlanetLab may provide an interface for users or applications to log their desire for more resources in order to provide hints about application resource requirements to better guide request policy. In either case, for PlanetLab applications to take advantage of new capacity they must adapt to incorporate the resources not available at startup.

Adding resources to PlanetLab pro-actively may prevent periods of high resource contention without requiring PlanetLab to be unnecessarily over-provisioned. PlanetLab applications and services will benefit, indirectly, if they can adapt to use less loaded resources as they are added. It remains to be seen how robust current, long-lived PlanetLab services are to an increased level of node-churn that would occur if nodes are added and removed on-demand. As an initial solution we propose adding lease times to PlanetLab's `AdmGetNodes` function: applications may use an XML-RPC interface to determine when a node is currently scheduled for removal from PlanetLab. Note that even today nodes leave PlanetLab for maintenance or due to failures—warnings about removal are broadcast to users over email.

PlanetLab claims that applications must be robust to inherent wide-area failures (i.e., "bad tracks make for good trains"); the idea being that network failures and occasional node failures will occur (and, in the past, have occured frequently on PlanetLab) and that long-lived applications and services must adapt to survive. That said, PlanetLab's stability has continually increased over time making node failures much less likely. In any case, PlanetLab's best-effort model prevents sufficiently adaptive applications from receiving any guarantee of their level of resource allotment preventing an application from supporting any higher-level service level agreements.

In the next section we discuss how to support adaptive, un-modified applications on PlanetLab and the implications for the underware control plane.

Component	*Description*
Application	Applications may be jobs (as often executed on the Grid) or long-lived distributed services. Applications have resource requirements that must be satisfied as they execute—most applications do not contain logic for adapting their resource allotment based on prevailing conditions.
Plush	Plush controls a distributed application's lifecycle: it deploys the application, monitors its status, and adapts to new conditions. Most importantly, it requires no changes to the application itself. Plush requires resources to deploy an application: it may use PlanetLab resources or may obtain them from the underware control plane (Shirako) directly.
PlanetLab	Provides a shared pool of resources available for applications; PlanetLab provides no resource guarantees, but does provide a consistent software stack for development and a world-wide resource presence. PlanetLab may obtain resources from an underware control plane that focuses solely on physical resource management.
Shirako	An example of the underware control plane. Shirako focuses narrowly on physical resource management: currently it allows middleware environments to obtain virtual machines on which they can control the entire software stack, including the OS. Shirako is federated: resource owners control how they donate resources to other environments.

Table 1: A description of the components discussed in Section 4, and their relation to one another.

4.3 Plush

To provide application adaptivity we use Plush [2]: a distributed execution environment originally developed to ease application and service deployment on PlanetLab. Plush allows unmodified applications under its control to adapt to PlanetLab's volatile environment. It uses a resource discovery service (SWORD [21]) to match (and re-match) an application to a suitable set of resources, and abstract application deployment and execution using an XML-specification that describes software installation, process execution workflows (i.e., the process execution order), and failure modes (i.e., what should I do if something fails). Plush exports an XML-RPC interface that allows control software to add or remove resource instances from a running application at any time; although, applications must include support for using resources once added.

Table 1 provides a description of the key software artifacts. To start, an application is written by an end user and deployed on a set of resources. The application may be a job that computes for some period of time and terminates, or it may be a long-lived service with its own user-base such as Coral or OpenDHT. Plush is responsible for deploying, monitoring, and adjusting the application's resource allotment over time to contend with demands from competing applications or workload surges. PlanetLab is an available pool of resources, with consistent per-node operating system, disk image, and user identity, that may be used by Plush to execute the application. Shirako is an example of the underware control plane from which PlanetLab may obtain "raw" hardware resources and boot PlanetLab-specific kernels and software. Note that Plush may obtain resources from the underware control plane directly, but either Plush, or the application, assumes the responsibility of providing the software stack. Figure 4 depicts Plush obtaining resources for an application from the underware and also from PlanetLab; Plush is bound to the application, but may obtain resources from a variety of sources.

Applications that use Plush can adapt to the current conditions on PlanetLab. For example, since PlanetLab provides only best-effort resource guarantees the availability of CPU, memory, or disk on any node may decrease dramatically at any time; Plush is able to recognize this and transfer (or migrate) application instances to new, less-loaded nodes. As described in [22], distributed PlanetLab applications often may "migrate" nodes by stopping processes on one node and starting processes on another; the applications have support for handling the departure of one instance and the arrival of another. While Plush-enabled PlanetLab applications can migrate to the "best" set of resources they can do nothing if PlanetLab has little or no available resources. An underware-enabled PlanetLab may request more resources to provide excess capacity; likewise, an over-provisioned PlanetLab may release resources for other uses instead of wasting them.

Even if PlanetLab adds additional capacity on-demand to meet load spikes applications have no guarantees: additional resources will be time-shared equally between competing applications. To support applications that require guarantees, we extended Plush to interface with the Shirako underware API and request resource-isolated virtual machines. Plush hides the underlying complexity of Shirako and presents to the user the same simple shell interface that a PlanetLab Plush user sees—in some sense, Plush acts like a minimalist OS that abstracts the underlying set of resources whether they be from PlanetLab or Shirako.

Interfacing Plush with Shirako allows easy experimentation with many existing applications, originally packaged by Plush users for PlanetLab, on Shirako. In particular, we have experimented with Bullet [19], a typical PlanetLab application, to see if it can benefit from

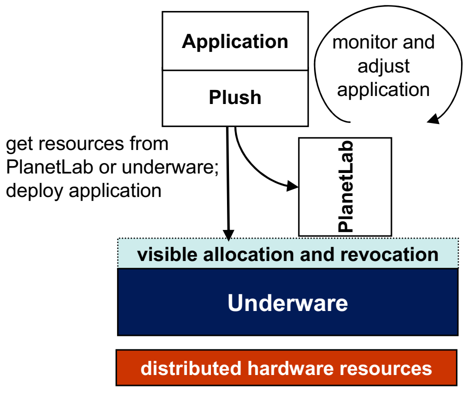
Application
Plush
monitor and adjust application
get resources from PlanetLab or underware; deploy application
PlanetLab
visible allocation and revocation
Underware
distributed hardware resources

Figure 4: Plush controls an application's lifecycle: it deploys an application on a set of resources, monitors its progress, and adapts its resource allotment in response to competition, failures, or flash-crowds. Plush can obtain resources from different sources: in this figure, Plush deploys an application on resources available on PlanetLab where it is provided a consistent software stack (i.e. OS and disk image) and also obtains resources the underware control plane directly (potentially providing its own application-specific software stack).

the increased resource control and isolation provided by Shirako. Our results, thus far, have been mixed: clearly changing resource allotments can improve performance, however, we have found that working knowledge of the distributed application is required to know on what machine to adjust resources, what resource to adjust, when to adjust the resources, and by how much. Effectively using explicit resource allocation and revocation for most PlanetLab applications is a research challenge in its own right—as a result underware, itself, is not sufficient as a platform for quickly deploying and testing experimental overlay networks.

5 Related

Underware enables a range of capabilities with respect to virtualization and resource isolation, Internet Infrastructures, federation, and autonomic computing.

Related Systems There are other systems that broadly fit our notion of underware; for instance, the XenoServer platform for global computing [20] is similar in that it offers virtual machines for use by applications. XenoServer applications use a resource discovery service to locate available resources at cluster sites. Once applications have been approved to use resources at a cluster site they are free to load any environment (including the OS) on the virtual machine.

Virtualization and Isolation As virtualization technologies become more prevalent it becomes increasingly more important to provide the hooks to leverage virtualization. Virtual machines have been a driving force in recent years, and in the future, storage virtualization and network virtualization will become more prevalent. Recent work [15] shows that resource isolation issues still must be addressed, although we believe the technology is sufficiently mature to partition resources at coarse granularities. Underware principles, particularly visible allocation/revocation, may also be applied to virtualization technology like Xen, however, as with previous work on extensible operating systems, partitioning at such a fine grain may not prove beneficial.

Internet Infrastructure Cabo [12] and Plutarch [8] share our vision of a pluralist Internet that is able to support multiple Internet architectures. We believe underware is a cornerstone for such an architecture and that PlanetLab and GENI should adopt a pluralist stance.

A recent paper [3] has recommended nine PlanetLab design choices to revisit (e.g., for GENI). Seven of them deal with resource control and flexible management policy, and could be addressed directly in an underware layer, as an alternative to adding "barnacles" to the PlanetLab architecture. The others deal with new programming abstractions; again, running independent PlanetLab instances (in provisioned containers) would foster innovation by enabling multiple alternative designs for the new features in its programming model and APIs. PlanetLab is a complex and important system with a large and growing base of users and software. Resource management underware would also enable very different models for programming applications, without placing so much pressure on PlanetLab to support the full diversity of experiments envisioned for GENI.

Underware would mitigate the "chaos" that Anderson and Roscoe [3] claim would ensue if resource owners were able to decide who manages their nodes, what runs on them, and the number of resources that are donated. Note that Anderson and Roscoe advocate more flexible management despite any short-term chaos it may cause; underware allows more flexible trust and increased innovation with limited chaos and a clear evolutionary path for new architectures. In fact, since underware is designed explictly as a dynamic artifact new underware architecures can be developed, deployed, and tested using an existing underware control plane.

Federation We advocate federation at the underware layer: resource owners export virtualized resources for use by other middleware environments. Federation at the underware layer does not preclude federating middleware, which typically involves extending the middleware environment across multiple resource owners. For instance, Grids federate by allowing remote users to submit jobs to local queues. PlanetLab appears to be taking this approach by extending the distributed virtualization abstraction across multiple PlanetLab instances [24], although PlanetLab's federation model is not yet complete.

Autonomic Computing There has been ongoing work that adapts application and service resource usage to gain better performance. Oppenheimer et al [22] examine the benefits of altering placement on PlanetLab to gain service performance improvements. Applications on PlanetLab can only adjust their placement because the pool of resources is static.

For environments where resource pools are not static feedback control can be used to adjust resource usage to meet higher-level service agreements [18]. Active learning of application profiles has also been used recently to build models that map application performance to resource allotment [26]. Both of these techniques require stronger guarantees than currently provided by PlanetLab, and are a foundation for autonomic computing.

6 Conclusion

This paper advocates a new underware layer as a cornerstone for an Internet operating system. Underware focuses narrowly on physical resource management—allowing hosted environments, such as the Grid or PlanetLab, to acquire virtual machines from resources donated by cluster sites. Like exokernel, underware does not attempt to provide abstractions; instead applications must provide their own abstractions or use existing middleware environments that provide abstractions.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, July 1986.

[2] J. Albrecht, C. Tuttle, A. Snoeren, and A. Vahdat. PlanetLab Application Management Using Plush. *ACM Operating Systems Review (SIGOPS-OSR)*, 40(1), January 2006.

[3] T. Anderson and T. Roscoe. Learning from PlanetLab. In *Proceedings of the Third Workshop on Real, Large Distributed Systems (WORLDS)*, November 2006.

[4] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles (SOSP)*, October 1991.

[5] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[6] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proceedings of the SIGCOMM Conference*, September 2006.

[7] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, and C. C. S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth Symposium on Operating System Principles (SOSP)*, December 1995.

[8] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: An Argument for Network Pluralism. In *ACM SIGCOMM Communications Review*, October 2003.

[9] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. QoS's Downfall: At the bottom, or not at all! In *Proceedings of the Workshop on Revisiting IP QoS*, August 2003.

[10] P. Druschel, V. S. Pai, and W. Zwaenepoel. Extensible Kernels are Leading OS Research Astray. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, May 1997.

[11] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles (SOSP)*, December 1995.

[12] N. Feamster, L. Gao, and J. Rexford. How to Lease the Internet in Your Spare Time. In *Technical Report GT-CSS-06-10*, August 2006.

[13] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[14] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.

[15] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the Seventh International Middleware Conference (Middleware)*, November 2006.

[16] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are Virtual Machines Monitors Microkernels Done Right? In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, June 2005.

[17] D. Irwin, J. S. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing Networked Resources with Brokered Leases. In *Proceedings of the USENIX Technical Conference*, June 2006.

[18] C. Karamanolis, M. Karlsson, and X. Zhu. Designing Controllable Computer Systems. In *Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS)*, June 2005.

[19] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *Proceedings of the Nineteenth ACM Symposium on Operating System Principles (SOSP)*, October 2003.

[20] E. Kotsovinos, T. Moreton, I. Pratt, R. Ross, K. Fraser, S. Hand, and T. Harris. Global-scale Service Deployment in the XenoServer Platform. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS)*, December 2004.

[21] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. In *Proceedings of the Fourteenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[22] D. Oppenheimer, B. Chun, D. A. Patterson, A. Snoeren, and A. Vahdat. Service Placement in a Shared Wide-Area Platform. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, June 2006.

[23] K. Park and V. S. Pai. CoMon: A Mostly-Scalable Monitoring System for PlanetLab. In *ACM SIGOPS Operating Systems Review (SIGOPS-OSR)*, January 2006.

[24] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences Building PlanetLab. In *Proceedings of the Seventh Symposium on Operating Systems Design anmd Implementation (OSDI)*, November 2006.

[25] L. Ramakrishnan, L. Grit, A. Iamnitchi, D. Irwin, A. Yumerefendi, and J. Chase. Toward a Doctrine of Containment: Grid Hosting with Adaptive Resource Control. In *Supercomputing (SC06)*, November 2006.

[26] P. Shivam, S. Babu, and J. S. Chase. Active and Accelerated Learning of Cost Models for Optimizing Scientific Applications. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, September 2006.