In VINI Veritas: Realistic and Controlled Network Experimentation

Andy Bavier *, Nick Feamster †, Mark Huang *, Larry Peterson *, and Jennifer Rexford *
* Princeton University † Georgia Tech

ABSTRACT

This paper describes *VINI*, a virtual network infrastructure that allows network researchers to evaluate their protocols and services in a realistic environment that also provides a high degree of control over network conditions. VINI allows researchers to deploy and evaluate their ideas with real routing software, traffic loads, and network events. To provide researchers flexibility in designing their experiments, VINI supports simultaneous experiments with arbitrary network topologies on a shared physical infrastructure. This paper tackles the following important design question: What set of concepts and techniques facilitate flexible, realistic, and controlled experimentation (*e.g.*, multiple topologies and the ability to tweak routing algorithms) on a fixed physical infrastructure? We first present VINI's high-level design and the challenges of virtualizing a single network. We then present *PL-VINI*, an implementation of VINI on PlanetLab, running the "Internet In a Slice". Our evaluation of PL-VINI shows that it provides a realistic and controlled environment for evaluating new protocols and services.

Categories and Subject Descriptors

C.2.6 [**Computer Communication Networks**]: Internetworking; C.2.1 [**Computer Communication Networks**]: Network Architecture and Design

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Internet, architecture, virtualization, routing, experimentation

1. Introduction

Researchers continually propose new protocols and services designed to improve the Internet's performance, reliability, and scalability. Testing these new ideas under realistic network conditions is a critical step for evaluating their merits and, ultimately, for deploying them in practice. Unfortunately, evaluating new ideas in operational networks is difficult, because of the need to convince equipment vendors and network operators to deploy the solution. Accordingly, researchers are faced with the option of evaluating their proposals via simulations, driven either by synthetic models of topology and workloads or by measurements of the existing protocols, or evaluating their proposals in a small-scale testbed. Ideally, researchers should be able to conduct experiments that are both realistic *and* controlled.[1]

Even services that operate above the network layer are difficult to evaluate without some level of visibility into and control over network events at lower layers. Consider a Resilient Overlay Network (RON) that circumvents performance and reachability problems in the underlying network by directing traffic through intermediate hosts [1]. RON can offer service to real users without modifying the underlying infrastructure; unfortunately, evaluating its effectiveness requires waiting for network failures to occur "naturally". Additionally, determining when and *why* a system like RON works—and how well it works under various failure scenarios—is challenging (if not impossible) without access to either information about failures in the underlying network or the ability to inject such failures [2].

Researchers evaluating new protocols and services should not be forced to choose between realistic conditions and controlled experiments. Instead, we believe that the research community needs an experimental infrastructure that satisfies the following four goals:

- **Running real routing software:** Researchers should be able to run conventional routing software in their experiments, to evaluate the effects of extensions to the protocols and to evaluate new services over commodity network components.
- **Exposing realistic network conditions:** Researchers should be able to construct experiments on realistic topologies and routing configurations. The experiments should be able to examine system behavior in response to exogenous events, such as routing-protocol messages from the "real" Internet.
- **Controlling network events:** Researchers should be able to inject network events (*e.g.*, link failures and flash crowds) that do not occur often in practice, to enable controlled experiments and fine-grained measurements of these events.
- **Carrying real traffic:** Researchers should be able to evaluate their protocols and services carrying application traffic between real end hosts, to enable measurements of end-to-end performance and effects of feedback at the end systems.

Satisfying these four goals requires both the tools for building virtual networks and the infrastructure for deploying them. On the one hand, PlanetLab is an infrastructure that supports multiple distributed services running on hundreds of machines throughout the world [3, 4]. However, conducting controlled and realistic networking experiments on PlanetLab is quite challenging, especially considering the first three goals above. On the other hand, toolkits like

[1] We precisely define the terms "realism" and "control" in Section 2.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCOMM'06, September 11–15, 2006, Pisa, Italy.
Copyright 2006 ACM 1-59593-308-5/06/0009 ...$5.00.

X-Bone [5] and Violin [6] automate the creation of overlay networks using tunnels between hosts, allowing researchers to evaluate new protocols and services. However, these tools are not integrated with a fixed, wide-area physical infrastructure that reflects a real network deployment. Instead, we believe the community needs a shared *infrastructure* (like PlanetLab) that can support *virtual networks* (like X-Bone and Violin), in a controlled and realistic environment.

To that end, we are building VINI, a Virtual Network Infrastructure, for evaluating new protocols and services. We are working with the National Lambda Rail (NLR) and Abilene Internet2 backbones to deploy VINI nodes that have direct connections to the routers in these networks and dedicated bandwidth between the sites. VINI will have its own globally visible IP address blocks, and it will participate in routing with neighboring domains.[2] Our goal is for VINI to become shared infrastructure that enables researchers to simultaneously evaluate new protocols and services using real traffic from distributed services that are also sharing VINI resources. The nodes at each site will initially be high-end servers, but may eventually be programmable hardware devices that can better handle a large number of simultaneous experiments carrying a large volume of real traffic and many simultaneously running protocols.

Rather than presenting a complete design and implementation of VINI, this paper addresses the following important prerequisite design question: *What set of concepts and techniques facilitate flexible, realistic, and controlled experimentation (*e.g.*, multiple topologies, ability to tweak routing algorithms, etc.) on a fixed physical infrastructure?* The answer to this question and other insights we glean from the design and implementation of VINI will provide important lessons for the design of experimental infrastructures such as the National Science Foundation's Global Environment for Network Innovations (GENI) [7, 8] and similar efforts in other countries. Toward this end, our paper makes three main contributions:

Proposed design of VINI: In designing VINI, we grapple with the challenges of representing every component in the network: routers, interfaces, links, routing, and forwarding, as well as the failures of these components, as discussed in Section 3. In addition to facing similar challenges as testbeds like PlanetLab, we must confront additional issues such as sharing routing-protocol port numbers across experiments, supporting multiple network topologies, numbering the ends of a virtual link from a common subnet, forwarding data packets quickly, diverting user traffic into the infrastructure, performing network address translation to receive return traffic from the Internet, and allowing multiple experiments to share a routing adjacency with a neighboring domain.

Initial prototype of VINI on PlanetLab: In prototyping VINI, we focus first on the significant challenges of supporting one experiment on the infrastructure at a time, as discussed in Section 4. We synthesize many of the software components created by the networking research community—from software routers to configuration-management tools—into a single functioning infrastructure. We use XORP for routing [9], Click for packet forwarding and network address translation [10], OpenVPN servers to connect with end users [11], and *rcc* for parsing router configuration data from operational networks to drive our experiments [12]. We use the PlanetLab nodes in Abilene for prototyping and experimenting, while working in parallel on deploying equipment for VINI.

Evaluation of PL-VINI: We evaluate our prototype to demonstrate its suitability for evaluating network architectures and systems in a realistic and controlled setting, as discussed in Section 5. We first use microbenchmarks to show that VINI efficiently forwards data packets. Our second set of experiments validates VINI's behavior in the wide-area. We mirror the Abilene backbone—with the real topology and the same OSPF configuration—on PlanetLab nodes co-located at Abilene PoPs. We inject a link failure into our network and observe the effects of OSPF route convergence on traffic running between two of the nodes.

As we continue to build VINI, we hope to provide the research community with not only a suitable environment for testing new network protocols and architectures, but also a credible path to real-world deployment.

[2]We are in discussions with service providers about having dedicated upstream connectivity to the commercial Internet at a few exchange points.

2. VINI Usage Model

Researchers who design, implement, and deploy new network protocols and architectures may demand different amounts of *control* and *realism* over many features of the network—topology (including link bandwidths), failure modes, and traffic load—depending on the aspects of the new protocol or architecture under test. Although VINI bears resemblance to many existing tools for evaluating network protocols and architectures, we believe that one of VINI's unique strengths is that it provides the experimenter considerably more latitude in introducing various amounts of control and realism into an experiment. This latitude makes VINI an environment that is suitable both for running controlled experiments and for deploying long-running deployment studies.

Control refers to the researcher's ability to introduce exogenous events (*e.g.*, failures, changes in traffic volume, etc.) into the system. Researchers and protocol designers often need control over an experiment to study the behavior of a protocol or system under a wide variety of network conditions. For example, an experiment that studies how link or node failures affect end-to-end performance in the context of a protocol modification requires the ability to inject failures into the routing system (rather than simply waiting for links or nodes to fail). VINI offers levels of control that are comparable to those provided in simulators such as ns-2 [13] or SSFNet [14], or in emulation testbeds (*e.g.*, Emulab [15], DETER [16], Modelnet [17], WAIL [18], or ONL [19]), which allow researchers to evaluate real prototypes of network protocols and architectures in a controlled environment.

Realism refers to the ability of a network researcher to subject a prototype network protocol or architecture to network conditions (*i.e.*, topology, failures, and traffic) that resemble those of an actual deployment as closely as possible. Although it is certainly possible to synthetically generate these features of the network according to various "realistic models", VINI's philosophy is that the best way to test a prototype under realistic conditions is to actually deploy the prototype in a real network. For example, as we will see in Section 5, VINI allows a researcher to deploy and test protocols on virtual networks that physically mirror the Abilene network, a capability that is not provided by any existing simulator or testbed.

VINI is most useful for experiments that ultimately require some level of both realism and control. These experiments fall into two broad classes: *controlled experiments* and *long-running deployment studies*. Although VINI can support controlled experiments involving synthetic traffic and network events, these experiments could arguably run in an existing testbed such as Emulab. However, a controlled experiment that requires some level of control over traffic, network events, or topology but eventually wants to incorporate realistic features can benefit tremendously from VINI.

Once a controlled experiment demonstrates the value of a new idea, the protocol might be deployed as a long-running study. Real end hosts—either users or servers—could "opt in" to the prototype

system, to achieve better performance, reliability, or security, or to access services that are not available elsewhere. In fact, one system running in VINI might even provide services for another, where end hosts subscribe to some service that, in turn, runs over a new network architecture deployed on VINI. For example, end hosts wanting certain guarantees about the integrity of content might subscribe to a content delivery service deployed over a secure routing infrastructure.

3. VINI Design Requirements

This section outlines the design requirements for a virtual network infrastructure. We focus on the general requirements of such an infrastructure—and why we believe the infrastructure should provide those requirements—independent of how any particular instantiation of VINI would meet these requirements.

VINI's design requirements are motivated by the desire for realism (of traffic, routing software, and network conditions) and control (over network events), as well as the need to provide sufficient flexibility for embedding different experimental topologies on a single, fixed physical infrastructure. Generally speaking, virtualization provides much of the machinery for solving this problem; indeed, virtualization is a common solution to many problems in computer architecture, operating systems, and even in networked distributed systems. Still, despite the promises of virtualization, its application to building *communication networks* is not straightforward.

As Table 1 shows, constructing a virtual network involves solving four main problems. First, the infrastructure must provide support for virtualizing network devices and attachment points because a network researcher may wish to use the physical infrastructure to build an arbitrary topology (Section 3.1). Second, once the basic topology is established, the infrastructure must facilitate running routing protocols over this virtual topology. This goal is challenging because each virtual node may have characteristics that are distinct from physical reality Section 3.2 discusses these requirements in more detail. Third, once the virtual network can establish its own routing and forwarding tables, it must be able to transport traffic to and from real networks (Section 3.3). Finally, the virtual network infrastructure should allow multiple network researchers to perform the above three steps using the same physical infrastructure, which presents complications that we discuss in Section 3.4.

3.1 Flexible Network Topology

To allow researchers (and practitioners) to evaluate new routing protocols, architectures, and management systems, VINI must offer the ability to configure a wide variety of nodes and links. Enabling this type of flexible network configuration requires satisfying two main challenges: the ability to configure each of these nodes with an arbitrary number of interfaces (*i.e.*, the flexibility to give each node an arbitrary degree), and the ability to provide the appearance of a physical link between any two virtual nodes (*i.e.*, the flexibility to establish arbitrary edges in the topology). Neither of these problems is straightforward: indeed, each problem involves somehow abstracting ("virtualizing") physical network components in new and interesting ways.

Problem: *Unique interfaces per experiment*. Routing protocols such as OSPF and IS-IS have configurable parameters for each interface (*e.g.*, weights and areas). To run these protocols, VINI must enable an experiment to have multiple interfaces on the same experiment, but most commodity physical nodes typically have a fixed (and typically small) number of physical interfaces. Limiting the flexibility of interface configuration to the physical constraints of each node is not acceptable: Because different experiments may need more (or fewer) interfaces for each node, massively overprovisioning each node with a large number of physical devices may prohibitively expensive and physically impossible.

Even if a node could be deployed with a plethora of physical interfaces, we ultimately envision VINI as an infrastructure that is *shared* among multiple experiments. Many experiments, each of which may configure a different number of virtual interfaces for each node, must be able to share a fixed (and likely small) number of physical interfaces.

Problem: *Virtual point-to-point connectivity*. To allow construction of arbitrary network topologies, VINI must also provide a facility for constructing virtual "links" (*i.e.*, the appearance of direct physical connectivity between any two virtual nodes). At first brush, providing this capability might seem simple: VINI can simply allow an experimenter to create the appearance of a link between any two arbitrary nodes by building an overlay network of tunnels. In principle, this approach is the essence of our solution, but our desire to make VINI look and feel like a "real" network—not just an overlay—presents additional complications.

Each virtual link must create the illusion of a physical link not only in terms of providing connectivity (*i.e.*, all physical nodes in between two endpoints of any virtual link must know how to forward traffic along that link) but also from the standpoint of resource control (*i.e.*, the performance of any virtual link should ideally be independent of the other traffic that is traversing that physical link). A primary concern is that the topology that an experimenter establishes in VINI should reflect to a reasonable degree the properties of the corresponding links in the underlying network. Virtual links in a VINI experiment will, in many cases, not consist of a single point-to-point physical connection, but may instead be overlaid on a sequence of physical links.

Providing this type of guarantee is challenging. First, some of these "links" may bear very little correspondence to how a layer-two link between the same nodes might actually behave, since each IP link comprising a single virtual link may experience network events such as congestion and failures independently. Ultimately, as we discuss in Section 3.4, the underlying links in the network may be shared by multiple topologies, and the traffic from one experiment may affect the network conditions seen in another virtual network. The challenges we face in solving these problems are similar in spirit to those faced by Emulab [20], but we are grappling with these issues over the Internet, rather than in a controlled testbed environment.

Problem: *Exposure of underlying topology changes*. A physical component and its associated virtual components should share fate. Topology changes in the physical network should manifest themselves in the virtual topology. If a physical link fails, for example, VINI should guarantee that the virtual links that use that physical link should see that failure. For example, VINI should not allow the underlying IP network to mask the failure by dynamically rerouting around it. Without this requirement, experiments on VINI would be subject to properties of the underlying network substrate (*e.g.*, IP routing), and the designer of a new network protocol, architecture, or management system would have trouble distinguishing properties of the new system from artifacts of the substrate.

3.2 Flexible Forwarding and Routing

VINI must not only provide the flexibility for constructing flexible network topologies, but it must also carry traffic over these topologies. This requirement implies that VINI must support capabilities for forwarding (*i.e.*, directing traffic along a particular path)

Design Requirement	Solution	Section
Flexible Network Topology (Section 3.1)		
Virtual point-to-point connectivity	Virtual network devices from common subnets in UML	4.1.3
	Tunnels and encapsulation in Click	4.2.1
Unique interfaces per experiment	Virtual network devices in UML	4.2.2
Exposure of underlying topology changes	Upcalls of layer-3 alarms to virtual nodes	—
Flexible Routing and Forwarding (Section 3.2)		
Distinct forwarding tables per virtual node	Separate instance of Click on each virtual node	4.2.1
Distinct routing processes per virtual node	Separate instance of XORP on each virtual node	4.2.2
Connectivity to External Hosts (Section 3.3)		
Allowing end hosts to direct traffic through VINI	End-host connection to an OpenVPN server	4.2.3
Ensure return traffic flows back through VINI	Network address translation in Click on egress	4.2.3
Support for Simultaneous Experiments (Section 3.4)		
Resource isolation between experiments	Virtual servers and network isolation in PlanetLab	4.1.1
	Extensions for CPU reservations and priorities	4.1.2
Distinct external routing adjacencies	BGP multiplexer to share external BGP sessions	—

Table 1: Design requirements for VINI. This table also discusses how our prototype implementation of VINI tackles each of these challenges; these solutions are discussed in more detail in Section 4.

and routing (*i.e.*, distributing the information that dictates how traffic is forwarded). VINI must provide its users the flexibility to arbitrarily control how routing and forwarding over the virtual topologies is done. Forwarding must be flexible because different experiments may require different virtual topologies. Routing must be flexible because each experiment may implement entirely different routing mechanisms and protocols. In this section, we describe how VINI's design facilitates node-specific forwarding and routing.

Problem: *Distinct forwarding tables per virtual node*. As we described in Section 3.1, different experiments may require different topologies: Any given virtual node may connect to a different set of neighboring nodes. For example, one experiment may use a topology where every node has a direct point-to-point connection with every other node, while another experiment may wish to set up a topology with significantly fewer edges. Supporting flexible topology construction not only requires supporting flexible interface configuration, but it also implies that the each topology will require different forwarding tables. In addition, VINI must also allow experimenters to implement completely different forwarding paradigms than those based on today's IPv4 destination-based forwarding. This implies that VINI must allow network experiments to specify different forwarding mechanisms (*e.g.*, forwarding based on source *and* destination, forwarding on tags or flat identifiers, etc.).

Problem: *Distinct routing processes per virtual node*. For similar reasons of flexible experimentation, VINI must enable each experiment to construct its own routing table and implement its own routing policies. Thus, in addition to giving each slice the ability to configure its own network topology and forwarding tables, VINI must also allow each experiment to run its own distinct routing routing protocols and processes. These routing processes much each handle two cases: (1) discovering routes to destinations within VINI; and (2) discovering routes to external destinations.

3.3 Connectivity to External Hosts

A cornerstone of VINI is the ability to carry traffic to and from real end hosts, to allow researchers to evaluate their protocols and services under realistic conditions. This enables *closed-loop* experiments that capture how network behavior affects end-to-end performance and, in turn, how adaptation at the end system affects the offered traffic. Supporting real traffic requires the VINI design to address the following two problems.

Problem: *Allowing end hosts to direct traffic through VINI*. End hosts should be able to "opt in" to having their traffic traverse an experiment running on VINI. For example, end users should be able to connect to nearby VINI nodes and have their packets reach services running on VINI, as well as external services (e.g., Web sites) on the existing Internet. This requires VINI to provide the illusion of an access network between the end host and the VINI node, and ensure that all packets to and from the end host (or to/from a particular application on the end host) reach the virtual node in the appropriate virtual topology. The virtual nodes can then forward these packets across the virtual topology using the forwarding tables constructed by the experimental routing software.

Problem: *Ensuring return traffic from external services flows back through VINI*. To support realistic experiments, VINI should be able to direct traffic to and from external hosts that offer communication services, even if these hosts do not participate in VINI. For example, a VINI experiment should be able to act as a stub network that connects to the Internet to reach a wide range of conventional services (e.g., Web sites). Directing traffic from VINI to the external Internet is not especially difficult. However, ensuring that the return traffic is directed to a VINI node, and forwarded through VINI and onward to the end host, is more challenging.

Solving these two problems would enable a wide range of experiments with either synthetic or real users running real applications that direct traffic over experimental network protocols and services running on VINI. Ultimately, we envision that some VINI experiments could provide long-running services for end users and applications that need better performance, security, and reliability than they have today.

3.4 Support for Simultaneous Experiments

VINI should support multiple simultaneous experiments to amortize the cost of deploying and running the physical infrastructure. In addition, running several experiments at the same time allows researchers to provide long-running services that attract real users, while still permitting other researchers to experiment with new protocols and services. Supporting multiple virtual topologies at the

same time introduces two main technical challenges in the design of VINI.

Problem: *Resource isolation between simultaneous experiments*. Each physical node should support multiple virtual nodes that are each part of its own virtual topology. To provide virtual nodes with their own dedicated resources, each physical node should allocate and schedule resources (*e.g.*, CPU, bandwidth, memory, and storage) so that the run-time behavior of one experiment does not adversely affect the performance of other experiments running on the same node. Furthermore, the resource guarantees must be *strict*, in the sense that they should afford an experiment no more—and no less—resources than allocated, to ensure repeatability of the experiments. Each virtual node also needs its own name spaces (e.g., file names) and IP addresses and port numbers for communicating with the outside world.

Problem: *Distinct external routing adjacencies per virtual node*. Multiple virtual nodes may need to exchange routing information, such as BGP announcements, with the same operational router in the external Internet. This is crucial for allowing each virtual topology to announce its own address space to the external Internet and control where its traffic enters and leaves the network. However, external networks are not likely to establish separate routing-protocol adjacencies with each virtual node, for two reasons. First, operational networks might reasonably worry about the stability of a routing-protocol session running on prototype software as part of a research experiment, especially when session failures and implementation errors might compromise routing stability in the real Internet. Second, maintaining multiple routing-protocol sessions (each with a different virtual node) would impose a memory, bandwidth, and CPU overhead on the operational router. VINI must address these issues to strike the right trade-off between providing flexibility (for experimenters) and robustness (for the external networks).

In the next section, we describe how we address these challenges in our prototype of VINI running on the PlanetLab nodes in the Abilene backbone.

4. A VINI Implementation on PlanetLab

As a first step toward realizing VINI, we have built an initial prototype on the PlanetLab nodes in the Abilene backbone. Although we do not (yet) have dedicated bandwidth between the nodes or upstream connectivity to commercial ISPs, this environment enables us to address many of the challenges of supporting virtual networks on a fixed physical infrastructure. For extensibility and ease of prototyping, we place many key functions in user space through careful configuration of the routing and forwarding software. In this section, we describe *PL-VINI* , our extensions to PlanetLab to support experimentation with network protocols and services, and "Internet In a Slice" (IIAS), a network architecture that *PL-VINI* enables.

Table 1 summarizes how the *PL-VINI* prototype addresses the problems outlined in Section 3. The table emphasizes that we must solve several problems in user space software (e.g., providing each experiment with point-to-point connectivity and unique network interfaces) that would ideally be addressed in the kernel or dedicated hardware. This division is a direct consequence our decision to implement our initial VINI prototype on PlanetLab; since PlanetLab must continue to support a large user base, we cannot make extensive changes to the kernel. We expect more functionality to be provided by the infrastructure itself as we gain insight from our initial experiences.

4.1 PL-VINI: PlanetLab Extensions for VINI

Our prototype implementation of VINI augments PlanetLab with features that improve its support for networking experiments. This goal appears to depart somewhat from PlanetLab's original mission, which was to enable wide deployment of *overlays*—distributed systems that, like networks, may route packets, but that communicate using sockets (*e.g.*, UDP tunnels). *PL-VINI* does, however, preserve PlanetLab's vision by enabling interesting and meaningful network protocols and services to be evaluated on an overlay; we describe one such network design in Section 4.2.

4.1.1 *PlanetLab: Slices and Resource Isolation*

PlanetLab was a natural choice for a proof-of-concept VINI prototype and deployment, both due to its large physical infrastructure and the virtualization it already provides. Virtualization—the ability to partition a real node and its resources into an arbitrary number of virtual nodes and resource pools—is a defining requirement of VINI. PlanetLab isolates experiments in virtual servers (VServers) [21]. Each VServer is a lightweight "slice" of the node with its own namespace. Because of the isolation provided by PlanetLab, multiple *PL-VINI* experiments can run on the same PlanetLab nodes simultaneously in different slices. VINI also leverages PlanetLab's slice management infrastructure.

VServers enable tight control over resources, such as CPU and network bandwidth, on a per-slice (rather than a per-process or a per-user) basis. The PlanetLab CPU scheduler grants each slice a "fair share" of the node's available CPU, and supports temporary share increases (*e.g.*, via Sirius [22]). Similarly, the Linux hierarchical token bucket (HTB) scheduler [23] provides fair share access to, and minimum rate guarantees for, outgoing network bandwidth. Network isolation on PlanetLab is provided by a module called VNET [24] that tracks and multiplexes incoming and outgoing traffic. VNET provides each slice with the illusion of root-level access to the underlying network device. Each slice has access only to its own traffic and may reserve specific ports.

4.1.2 *Improved CPU Isolation*

PlanetLab provides a fair share of the CPU resources to each slice, but fluctuations in the CPU demands of other slices can make running repeatable networking experiments challenging. If a node supports a large number of slices, a routing process running in one slice may not have enough processing resources to keep up with sending heartbeat messages and responding to events, and a forwarding process may not be able to maintain a desired throughput. Many slices simultaneously contending for the CPU can also lead to jitter in scheduling a forwarding process, which manifests itself in an overlay network as added latency.

PL-VINI leverages two recently exposed CPU scheduling knobs on PlanetLab: CPU reservations and Linux real-time priorities [25]. A CPU reservation of 25% provides the slice with a minimum of 25% of the CPU during the times that it is active, though it may get more than this if no "fair share" slices are running. Boosting a process to real-time priority on Linux cuts the time between when a process wakes up (e.g., receives a packet) and it runs. A real-time process that becomes runnable immediately jumps to the head of the run-queue and preempts any non-real-time process. Note that even real-time processes are still subject to PlanetLab's CPU reservations and shares, so a real-time process that runs amok cannot lock the machine. These two PlanetLab capabilities provide greater isolation for a VINI experiment running in a slice. In Section 6.2 we describe several additional extensions we are exploring to provide even better isolation between *PL-VINI* slices.

4.1.3 *Virtual Network Devices*

A networking experiment running in a slice in user space needs the illusion that each virtual node has access to one or more network devices. Our prototype leverages User-Mode Linux (UML) [26], a full-featured Linux kernel that runs as a user-space process, for this purpose. For each user-space tunnel in our overlay topology, *PL-VINI* creates a pair of interfaces on a common subnet in the UML instances at its endpoints. Routing software running inside UML is in this way made aware of the structure of an overlay network. *PL-VINI* then maps packets sent on these network interfaces to the appropriate tunnel at a layer beneath UML. We note that Violin [6] also uses an overlay network to connect UML instances. However, the goal of Violin is to hide topology from Grid applications, whereas *PL-VINI* uses network interfaces in UML to expose a tunnel topology to the routing software that runs above it.

Our prototype also uses a modified version of Linux's TUN/TAP driver to allow applications running in the networking experiment's slice to send and receive packets on the overlay. A process running in user space can read from `/dev/net/tunX` to receive packets routed by the kernel to the TUN/TAP device; similarly, packets written to `/dev/net/tunX` are injected back into the kernel's network stack and processed as if they arrived from a network device. Our modifications to the driver allow it to preserve the isolation between different slices on PlanetLab: every slice sees a single TUN/TAP interface with the same IP address, but our changes allow multiple processes (in different slices) to read from `/dev/net/tunX` simultaneously, and each will only see packets sent by its own slice.

For *PL-VINI* , we create a virtual Ethernet device called `tap0` on every PlanetLab node. We give each `tap0` device a unique IP address chosen from the 10.0.0.0/8 private address space. This means that each PlanetLab node's kernel will route all packets matching 10.0.0.0/8 to `tap0` and onto that slice's own overlay network.

4.2 IIAS: "Internet In a Slice" Architecture

The *Internet In a Slice* (IIAS) is the example network architecture that we run on our *PL-VINI* . Researchers can use IIAS to conduct controlled experiments that evaluate the existing IP routing protocols and forwarding mechanisms under realistic conditions. Alternatively, researchers can view IIAS as a reference implementation that they can modify to evaluate extensions to today's protocols and mechanisms. An IIAS consists of five components [27]:

1. a forwarding engine for the packets carried by the overlay (an overlay *router*);
2. a smart method of configuring the engine's forwarding tables (a *control plane*); and
3. a mechanism for clients to opt-in to the overlay and divert their packets to it, so that the overlay can carry real traffic (an overlay *ingress*);
4. a means of exchanging packets with servers that know nothing about the overlay, since most of the world exists outside of it (an overlay *egress*);
5. a collection of *distributed machines* on which to deploy the overlay, so that it can be properly evaluated and can attract real users.

Our IIAS implementation synthesizes many components created by the networking research and open source communities. IIAS employs the Click modular software router [10] as the forwarding engine, the XORP routing protocol suite [9] as the control plane, OpenVPN [11] as the ingress mechanism, and performs NAT (within Click) at the egress. Since we run IIAS on *PL-VINI* , IIAS can also use *PL-VINI* 's `tap0` device as an ingress/egress mechanism for applications running on a *PL-VINI* node.

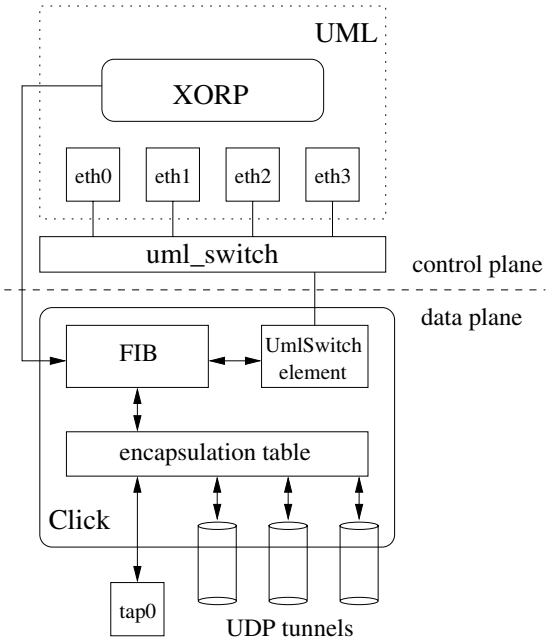


Figure 1: An IIAS router on *PL-VINI*

Figure 1 shows the IIAS router supported by *PL-VINI* . Routing protocols implemented by XORP, running unmodified in a UML kernel process, construct a view of the overlay network topology exposed by the virtual Ethernet interfaces. Each XORP instance then configures a forwarding table (FIB) implemented in a Click process running outside of UML. This means that data packets forwarded by the overlay do not enter UML, which leads to better performance since forwarding data packets in the UML kernel incurs nearly 15% additional overhead [6]. Next we discuss significant features of each component in the IIAS software.

4.2.1 *Click: Links and Packet Forwarding*

IIAS uses the Click modular software router [10] as its virtual data plane. Our Click configuration consists of five components that create the illusion of point-to-point links to other virtual nodes and enable the virtual nodes to forward data packets:

- **UDP tunnels:** UDP tunnels (*i.e.*, sockets) are the links in the IIAS overlay network. Each Click instance is configured with tunnels to each of its neighbors in the overlay.
- **Local interface:** Click reads and writes Ethernet packets to *PL-VINI* 's local `tap0` interface. Packets sent by local applications to a 10.0.0.0/8 destination are forwarded by the kernel to `tap0` and are received by Click. Likewise, Click writes packets destined for `tap0`'s IP address to the interface, injecting the packets into the kernel which delivers them to the proper application.
- **Forwarding table:** Click's forwarding table maps IP prefixes (both within and outside of IIAS's private address

space) to "next hops" within IIAS. The forwarding table is initially empty and is populated by XORP. Since XORP sees a network of virtual Ethernet interfaces, the "next hops" inserted by XORP are the IP addresses of the virtual interfaces on neighboring nodes.

- **Encapsulation table:** The preconfigured encapsulation table matches the "next hop" selected by the forwarding table to a UDP tunnel by mapping it to the public IP address of a PlanetLab node.

- **UML Switch:** Click exchanges Ethernet packets with the local UML instance via a virtual switch (`uml_switch`) distributed with UML. We wrote a Click element so that Click could connect to this virtual switch.

Two points about the IIAS data plane are worthy of note. First, the forwarding table in IIAS controls both how data and control traffic is forwarded between IIAS nodes, and how traffic is forwarded to external destinations (*i.e.*, on the "real" Internet). Second, though IIAS currently performs IPv4 forwarding, it can also support new forwarding paradigms beyond IP. Our design has no fundamental dependence on IP since Click exchanges Ethernet frames with UML (via the virtual switch) and the local `tap0` interface. One could implement a new addressing scheme in IIAS, for instance based on DHTs, simply by writing new forwarding and encapsulation table elements.

4.2.2 *XORP: Routing*

IIAS uses the XORP open-source routing protocol suite [9] as its control plane. XORP implements a number of routing protocols, including BGP, OSPF, RIP, PIM-SM, IGMP, and MLD. XORP manipulates routes in the data plane through a Forwarding Engine Abstraction (FEA); supported forwarding engines include the Linux kernel routing table and the Click modular software router (which is why we chose XORP for IIAS).

The main complication of running XORP on PlanetLab is the lack of physical interfaces to correspond to each virtual link in our configuration. XORP generally assumes that each link to a neighboring router is associated with a physical interface; OSPF also assigns costs to network interfaces. In our Click data plane, interfaces conceptually map to sockets and links to tunnels. Therefore, to present XORP with a view of multiple physical interfaces, we run it in UML and map packets from each UML interface to the appropriate UDP tunnel in Click.

An important feature of IIAS is that it decouples the control and data planes by placing the routing protocol in a different virtual world than the forwarding engine. In fact, decoupling the control and data planes in this way means that XORP could run in a different slice than Click, or even on a different node.

4.2.3 *OpenVPN and NAT: External Connectivity*

IIAS is intended to enable realistic experiments by carrying real traffic generated by outside hosts, as well as applications running on the local node. IIAS uses OpenVPN [11] as an ingress mechanism; IIAS runs an OpenVPN server on a set of designated ingress nodes, and hosts "opt-in" to a particular instance of IIAS by connecting an OpenVPN client that diverts their traffic to the server. OpenVPN is a robust, open-source VPN access technology that runs on a wide range of operating systems and supports a large user community. Note that OpenVPN creates a TUN/TAP device on the client to intercept outgoing packets from the operating system, just as we do in *PL-VINI* and IIAS.

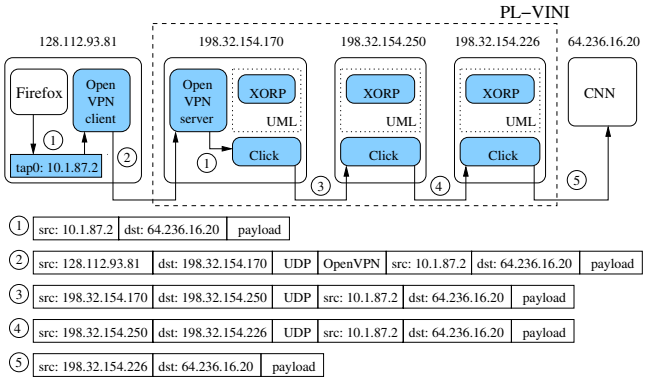


Figure 2: The life of a packet in IIAS (shown shaded) running on PL-VINI (dotted box)

IIAS's Click forwarder implements NAPT (Network Address and Port Translation) to allow hosts participating in IIAS to exchange packets with external hosts that have not "opted-in" (like a Web server). IIAS forwards packets destined for an external host to an egress point, where they exit IIAS via NAPT. This involves rewriting the source IP address of the packet to the the egress node's public IP address, and rewriting the source port to an available local port. After passing through Click's NAPT element, a packet is sent out and forwarded to the destination by the "real" Internet. Note that, since the packets reaching the external host bear the source address of the IIAS egress node, return traffic is sent back to that node, where it is intercepted by IIAS and forwarded back to the client.

PL-VINI 's `tap0` interface provides another ingress/egress mechanism for other applications running in the same slice as IIAS. For example, in the experiments described in Section 5, we send `iperf` packets through the overlay using `tap0`.

4.2.4 *IIAS Summary: Life of a Packet*

Figure 2 ties together the discussion of the various pieces of IIAS by illustrating the life of a packet as it journeys through the IIAS overlay. In Figure 2, the Firefox web browser on the client machine at left is sending a packet to `www.cnn.com` at right through IIAS (shown shaded). The steps along the packet's journey are:

1. Firefox sends a packet to CNN. The routing table of the client directs the packet to the local `tap0` device that was created by OpenVPN. This device bounces the packet up to the OpenVPN client on the same machine. The packet has a source of 10.0.87.2 (the local `tap0` address) and a destination of 64.236.16.20 (the IP address of CNN's web server).

2. The OpenVPN client tunnels the packet over UDP to an OpenVPN server running on a nearby IIAS node. The packet is encapsulated in IP, UDP, and OpenVPN encryption headers. The OpenVPN server removes the headers and forwards the original packet to Click over a local Unix domain socket.

3. Click looks up 64.236.16.20 in its forwarding table and maps it to the IP address of a UML interface on a neighboring node. Click consults the encapsulation table to map the UML address to 198.32.154.250 (the real IP address of the next hop), and sends the packet over a UDP tunnel to the latter address. The same process happens again on the next node.

4. The Click process running on 198.32.154.226 receives the original packet from a UDP tunnel, consults the forwarding

Figure 3: DETER topology for microbenchmarks

table, and sees that it is the egress node for 64.236.16.20. Click sends the packet through its NAPT element, which rewrites the source IP address to the local `eth0` address, and rewrites the source port to an available local port (port rewriting is not shown in Figure 2). Click then directs the packet to `www.cnn.com` via the public Internet.

Then, the packet traverses the rest of the path through the Internet to the CNN Web server. The response packets from CNN have a destination IP address of 198.32.154.226, ensuring they return to the client through the VINI node.

5. Preliminary Experiments

In this section, we describe two experiments that we have run in IIAS on *PL-VINI* . These experiments are intended not to demonstrate *PL-VINI* as a "final product", but rather as a proof of concept that highlights the efficiency, correctness, and utility of the VINI design. The microbenchmark experiments (Section 5.1) demonstrate that *PL-VINI* provides a level of support for networking experiments comparable to running on dedicated hardware, allowing the experiment's throughput and traffic flow characteristics to mirror that of the underlying network. Next, intra-domain routing experiments (Section 5.2) on the Abilene topology demonstrate that meaningful results for such experiments can be obtained using *PL-VINI* on PlanetLab.

5.1 Microbenchmarks

The purpose of the microbenchmarks is to demonstrate that *PL-VINI* can support an interesting networking experiment on PlanetLab. To this end, we first establish that the IIAS overlay behaves like a real network when run on dedicated hardware in an isolated environment, and then show that *PL-VINI* can provide IIAS with a similar environment on PlanetLab.

In order to provide a realistic environment for network experiments, *PL-VINI* must enable IIAS to deliver along two dimensions:

- **Capacity:** To attract real users and real traffic, IIAS must be able to forward packets at a relatively high rate. If IIAS's performance is bad, nobody will use it.
- **Behavior:** To boost our confidence that observed anomalies are meaningful network events and not undesirable artifacts of the *PL-VINI* environment, IIAS should exhibit roughly the same behavioral characteristics as the underlying network.

We run two sets of experiments to measure the capacity and behavior of IIAS. The first set of experiments runs on dedicated machines on DETER [16], which is based on Emulab [15]; we quantify the efficiency of the IIAS overlay by evaluating the performance of DETER's emulated network topology versus IIAS running over that same topology. The second set of experiments repeats the DETER experiments on PlanetLab; here we quantify the effects of moving IIAS from dedicated hardware (DETER/Emulab) to a shared platform (PlanetLab), and then show how PL-VINI's support for CPU reservations and real-time priority reduce CPU contention.

The microbenchmark experiments are run using `iperf` version 1.7.0 [28]. We measure capacity using `iperf`'s TCP throughput test

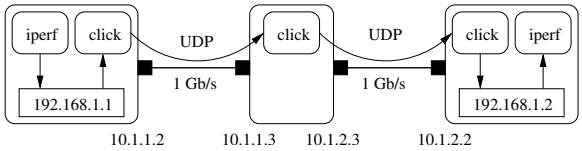


Figure 4: Overlay topology on DETER

	mean (Mb/s)	stddev	mean CPU%
Network	940	0	48
IIAS	195	0.843	99

Table 2: TCP throughput test on DETER testbed

	min	avg	max	mdev	% loss
Network	0.193	0.414	0.593	0.089	0
IIAS	0.269	0.547	0.783	0.080	0

Table 3: `ping` results on DETER; units are ms

to send 20 simultaneous streams from a client to a server through the underlying network and *PL-VINI* . We measure behavior with `iperf`'s constant-bit-rate UDP test, observing the jitter and loss rate of packet streams (with 1430-byte UDP payloads) of varying rates. Each test is run 10 times and we report the mean and standard deviation. When measuring the capacity of *PL-VINI* , we also report the mean CPU percentage consumed by the Click process (using the TIME field as reported by `ps`).

5.1.1 *Microbenchmark #1: Overlay Efficiency*

First we compare the capacity and behavior of IIAS's user-space Click forwarder versus in-kernel forwarding. The experiments are run on the DETER testbed, which allows a researcher to specify an arbitrary network topology for an experiment, including emulated link characteristics such as delay and loss rate, using a *ns* script. The machines used in the experiment are `pc2800` 2.8 GHz Xeons with 2 GB memory and five 10/100/1000 Ethernet interfaces, and are running Linux 2.6.12.

Our experiments run on a simple topology shown in Figure 3, consisting of three machines connected by Gigabit Ethernet links that do not have any emulated delay or loss. In this topology, the machine *Fwdr* is configured as an IP router; a packet sent from *Src* to *Sink*, or vice-versa, is forwarded in *Fwdr*'s kernel. We compare the performance of the network with that of IIAS running on the same three nodes. We configure a Linux TUN/TAP device on each node to divert packets sent by `iperf` to the local Click process. Click then tunnels the packets over the topology as shown in Figure 4. The key difference between the two scenarios is that IIAS makes the the forwarding decisions in user space rather than in the Linux kernel.

Table 2 shows the results of the TCP throughput test for the IIAS overlay versus the underlying network. IIAS is not as efficient as the network alone: it manages to achieve about 10% of the throughput with an equal amount of CPU. The throughput achieved by the Linux kernel, 940Mb/s, was roughly the maximum supported by the configuration, and even at this maximum rate the CPU of *Fwdr* was 52% idle. In comparison, Click's forwarding rate is CPU-bound. Running `strace` on the Click process indicates (not surprisingly) that the issue is system-call overhead: for each packet forwarded, Click calls `poll`, `recvfrom`, and `sendto` once, and `gettimeofday` three times, with an estimated cost of 5μs per call. For `sendto` and `recvfrom`, this cost appears to be independent of packet size. Reducing this overhead is future work. However, stepping back, we observe that even 200Mb/s is a significant amount

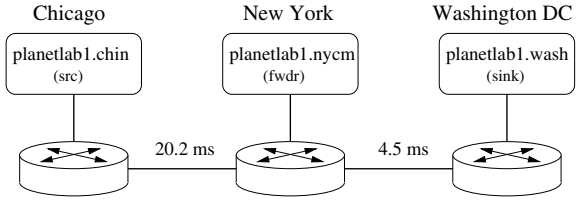


Figure 5: PlanetLab topology for microbenchmarks

	Mb/s	stddev	CPU%
Network	90.8	0.53	N/A
IIAS on PlanetLab	22.5	4.01	13
IIAS on *PL-VINI*	86.2	0.64	40

Table 4: TCP throughput test on PlanetLab

	min	avg	max	mdev	loss
Network	24.4	24.5	28.2	0.2	0%
IIAS on PlanetLab	24.7	27.7	80.9	4.8	0%
IIAS on *PL-VINI*	24.7	25.1	28.6	0.38	0%

Table 5: `ping` results on PlanetLab; units are ms

	mean	stddev
Network	0.27	0.16
IIAS on PlanetLab	2.4	3.7
IIAS on *PL-VINI*	1.3	0.9

Table 6: Summary of jitter results on PlanetLab; units are ms

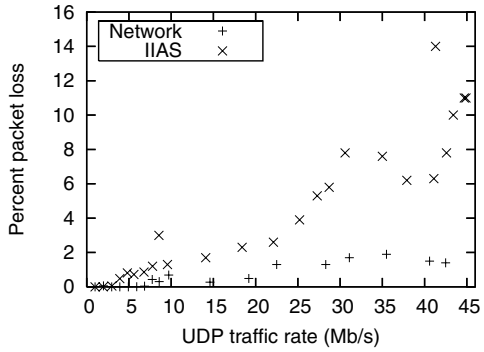
of throughput for a networking experiment, as it far outstrips the available bandwidth between edge hosts in the Internet today.

Next we compare the fine-grained behavior of the network and IIAS. Table 3 shows the results of measuring latency on the overlay and network using `ping -f -c 10000`. We see that IIAS adds about 130μs latency on average, but doesn't change the standard deviation of ping times. Likewise, running UDP CBR streams at rates from 1Mb/s to 100Mb/s over the network and IIAS did not reveal significant jitter in either case. In all UDP CBR tests, `iperf` observed jitter of less than 0.1ms and no packet losses.

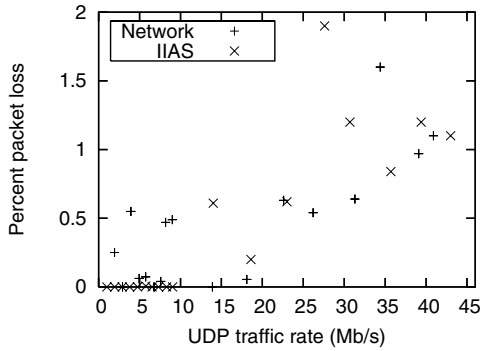
5.1.2 *Microbenchmark #2: Overlay on PlanetLab*

The next set of microbenchmarks contrasts the behavior of IIAS running on dedicated hardware (DETER) to a shared platform (PlanetLab) and *PL-VINI* . Our main concern is that the activities of other users on a shared system like PlanetLab can negatively affect the performance of IIAS. To test this, we repeat the experiments of Section 5.1.1 on three PlanetLab nodes co-located with Abilene PoPs. Figure 5 shows the topology of the PlanetLab nodes and the underlying Abilene network, as revealed by running `traceroute` between the three nodes. The Chicago and Washington, D.C. PlanetLab nodes are 1.4 GHz P-III, and the New York node is a 1.267 GHz P-III; all nodes have 1 GB of memory. Again, we compare the capacity and behavior of IIAS with that of the underlying network. Note that the network traffic between Chicago and Washington traverses the three routers only, but IIAS traffic traverses *four* router hops since it is forwarded by the Click process on the New York node and so visits the local router twice. Because the links in the Abilene backbone are lightly loaded, we do not expect to see significant interference from cross traffic.

PlanetLab makes running meaningful experiments challenging because it is shared among many users, whose actions may change the experimental results. The Emulab microbenchmarks indicate



(a) With default share



(b) With PL-VINI

Figure 6: Packet losses in IIAS on PlanetLab

that CPU contention in particular is likely to be a problem for *PL-VINI* on PlanetLab; however, *PL-VINI* uses CPU reservations and real-time priorities to provide consistent CPU scheduling behavior. Therefore, we run our experiments from Section 5.1.1 using PlanetLab's default fair share ("IIAS on PlanetLab"), as well as a 25% CPU reservation plus a priority boost for the IIAS Click process ("IIAS on *PL-VINI* " in the tables and graphs). The CPU reservation improves the overall capacity of IIAS by giving it more CPU, while the boost to real-time priority reduces the scheduling latency of the Click process and so improves end-to-end overlay latency.

Table 4 shows the results of the bandwidth test with both sets of CPU scheduling parameters. We note that, with *PL-VINI* , IIAS approaches the underlying network in both observed throughput and variability of the result. Running IIAS on *PL-VINI* provides a 4X increase in throughput and reduces variability by over 80%.

Focusing on fine-grained behavior of IIAS on PlanetLab, Table 5 presents results using ping. IIAS clearly introduces significant variability in the latency measurements when run with the default share: the standard deviation in *PL-VINI* ping times is over 20X that of the network. *PL-VINI* again improves IIAS's overall behavior, reducing maximum latency by two-thirds and standard deviation by over 90%. In this case IIAS introduces a small amount of additional latency, and the variability in ping times is roughly double that of the underlying network.

Table 6 shows the effects of *PL-VINI* on jitter in the IIAS overlay. The experiment sends CBR streams between 1Mb/s and 50Mb/s on the network and overlay; jitter did not appear to be correlated with stream size and so we report the the jitter results across all streams. Here we see that running IIAS on *PL-VINI* halves the mean jitter and reduces the variation in test results by 75%.

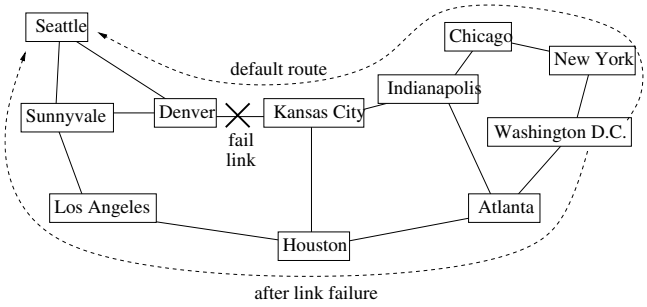


Figure 7: Abilene Topology

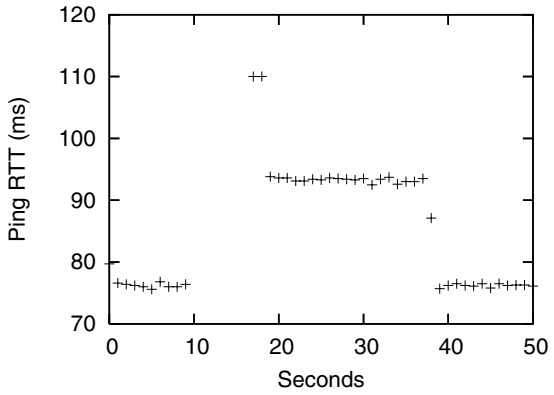


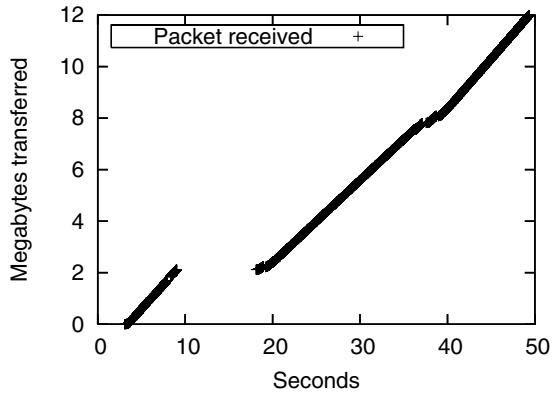
Figure 8: Observing OSPF route convergence (using ping)

Figure 6 shows packet loss in the same set of experiments. Interestingly, with the default share on PlanetLab, IIAS loses packets dramatically as the traffic rate increases as shown in Figure 6(a). Our hypothesis is that this is due to scheduling latency of the Click process: packets are arriving at a constant rate on the UDP tunnel, and Click needs to read them at a faster rate than they are arriving or else the UDP socket buffer will overflow and the kernel will drop packets. However, if Click's scheduling latency is high, it may not get to run before packets are dropped. This hypothesis is confirmed by running IIAS on *PL-VINI* : Figure 6(b) shows packet loss with *PL-VINI* comparable to that measured in Abilene itself.

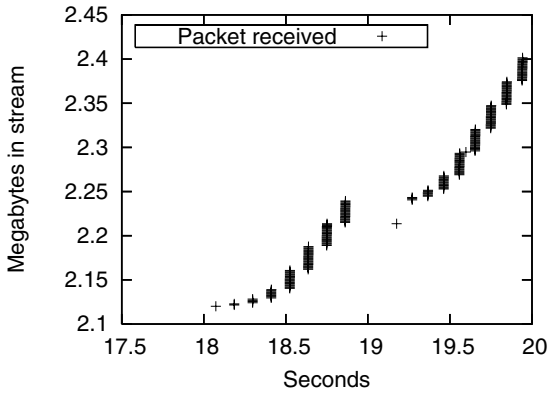
We conclude from these microbenchmarks that *PL-VINI* and IIAS together provide a close approximation of the underlying network's behavior. Clearly, running traffic through an overlay does introduce some overhead and additional variability. In the next experiment we try to demonstrate that the value of being able to run IIAS using *PL-VINI* outweighs this additional overhead.

5.2 Intra-domain Routing Changes

To validate that together IIAS and *PL-VINI* provide a reasonable environment for network experiments, we use them to conduct an intra-domain routing experiment on the PlanetLab nodes co-located with the eleven routers in the Abilene backbone, as shown in Figure 7. To conduct a realistic experiment, we configure IIAS with the same topology and OSPF link weights as the underlying Abilene network, as extracted from the configuration state of the eleven Abilene routers. That is, each virtual link maps directly to a single physical link between two Abilene routers. Analyzing routing traces collected directly from the Abilene routers enables us to verify that the underlying network did not experience any routing changes during our experiment.



(a) Total bytes transferred



(b) TCP slow-start restart after new route is found

Figure 9: TCP throughput during OSPF routing convergence

Our experiment injects a failure, and subsequent recovery, of the link between Denver and Kansas City, and measures the effects on end-to-end traffic flows. For this experiment, we "fail" the link by dropping packets within Click on the virtual link (UDP tunnel) connecting two Abilene nodes. We use ping, iperf, and tcpdump to measure the effects on data traffic.

Figure 8 shows the effect on ping times between D.C. and Seattle of failing the link between Kansas City and Denver 10 seconds in to the experiment, and restoring the link at time 34 seconds. Initially, IIAS routes packets from D.C. through New York, Chicago, Indianapolis, Kansas City, and Denver to Seattle, with a mean round-trip time (RTT) of $76ms$. At time 17, 7 seconds after the link fails, OSPF briefly finds a path with $110ms$ RTT before settling on a new route through Atlanta, Houston, Los Angeles, and Sunnyvale with a mean RTT of $93ms$[3]. A few seconds after the link comes back up at time 34, we see that OSPF briefly finds a path with $87ms$ before falling back to the original path.

Figure 9 shows the performance of TCP during the same experiment by using `iperf` to send a bulk TCP transfer from Washington, D.C. to Seattle. The TCP receiver window size is set to `iperf`'s default of 16 KB, so TCP's throughput is limited to roughly 3 Mb/s. The figure plots the arrival time of data packets at the receiver, as reported by `tcpdump`. Figure 9(a) shows that packets stop getting through when the link fails at time 10, and resume at time 18 when OSPF finds the a new route. Figure 9(b) shows what happens at

[3]For this experiment, the interval between OSPF hello packets is set at 5 seconds, and the "router dead" interval is 10 seconds.

time 18 in more detail; the y axis shows the position in the byte stream of each arriving TCP packet. The figure shows TCP slow-start restart in action, then a retransmitted packet, and slow start again. Figure 9(a) also shows some disruption in the TCP throughput when OSPF falls back to the original path around time 38.

These experiments do not illustrate any new discoveries about OSPF or its interaction with TCP. Rather, we argue that they demonstrate one *could* make such discoveries using *PL-VINI* and IIAS, since *PL-VINI* enables IIAS to behave like a real network on PlanetLab. Experiments such as this can help researchers study routing pathologies that are difficult to observe on a real network, where a researcher has no control over network conditions.

6. Ongoing Work

In this section, we discuss our ongoing work on VINI. Specifically, we discuss some of the design goals from Section 3 that we have yet to address and describe possible solutions to these problems. First, we discuss two ways to improve the realism of VINI's experiments: by exposing the underlying topology changes and by enabling experiments to exchange routing-protocol messages with neighboring domains. We then describe our ongoing efforts to provide researchers with better experimental control by allowing seamless migration from simulators (*e.g.*, ns-2) and emulation environments (*e.g.*, Emulab) to VINI. Finally, we propose techniques to provide better isolation between experiments.

6.1 Improving Realism

Exposing network failures and topology changes: The failure or recovery of a physical component should affect each of the associated virtual components, as discussed in Section 3.1. Our PL-VINI prototype does not achieve this goal because the underlying network automatically reroutes the traffic between two IIAS nodes when the topology changes. Although masking failures is desirable to most applications, researchers using VINI may want their protocols and services to adapt to these events themselves, in different ways; at a minimum, the researchers would want to know that these events happened, since they may affect the results of the experiments. As we continue working with NLR and Abilene, we are exploring ways to expose the topology changes to VINI in real time, and extending our software to perform "upcalls" to notify the affected slices.

Participating in routing with neighboring networks: As discussed in Section 3.4, multiple VINI experiments may want to exchange reachability information with neighboring networks in the real Internet. Having each virtual node maintain separate BGP sessions introduces problems with scaling (because the number of sessions may be large as the number of experiments grows), management (because *both* sides of the BGP session must be configured), and stability (unstable, experimental software could introduce instability into neighboring networks and the rest of the Internet).

To avoid these potential issues, we are designing and implementing a multiplexer that manages BGP sessions with neighboring networks and forwards (and filters) routing protocol messages between the external speakers and the BGP speakers on the virtual nodes. Each experiment might have its own portion of a larger address block that has already been allocated to VINI. The multiplexer ensures that each virtual node announces only its own address space and may also impose limits on the rate of BGP update messages that are propagated from each experiment. Our current implementation of the BGP multiplexer is implemented as multiple instances of XORP, each running in UML and communicating with a single external speaker. Each instance of XORP maintains BGP sessions with the routing software running on the virtual nodes, to allow experiments to exchange BGP messages with neighboring domains.

6.2 Improving Control

Better isolation: VINI should be able to support multiple simultaneous experiments with strict resource guarantees for each slice, as discussed earlier in Section 3.4. Adding support for CPU reservations and real-time priority helps isolate a PL-VINI experiment from other slices, but PL-VINI arguably needs better isolation. The first step is to implement a non-work-conserving scheduler that ensures that each experiment always receives the same CPU allocation (*i.e.*, neither less nor more), which is necessary for repeatable experiments. To allow researchers to vary link capacities, we also plan to add support for setting link bandwidths, either via configuration of traffic shapers in Click, or in the kernel itself.

Experiment specification: Beyond the existing support for constructing arbitrary topologies and failing links, VINI should also provide the ability to *specify* experiments. In an *ns* simulation [13], an experimenter can generate traffic and routing streams, specify times when certain links should fail, and define the traces that should be collected. VINI should provide similar facilities for creating an experiment. We envision that VINI experiments would be specified using the same type of syntax that is used to construct *ns* or Emulab [15] experiments, so that researchers can move an experiment from Emulab to VINI as seamlessly as possible, as part of a natural progression. We are currently working on such a specification for IIAS, which already allows an experimenter to specify the underlying topology, the intradomain routing adjacencies and internal BGP sessions, and the times these links and sessions fail.

We envision that aspects of a VINI experiment, such as topologies, routing configurations, and failures, could be driven by "real world" routing configurations and measurements. PL-VINI's current machinery for mirroring the Abilene topology automatically generates the necessary XORP and Click configurations (and determines the appropriate co-located nodes at Abilene PoPs) for a VINI experiment from the actual Abilene routing configuration, exploiting the configuration-parsing functionality from previous work on router configuration checking [12]. Eventually, we intend to augment VINI to incorporate more of the routing configuration into XORP and Click and also support playback of routing traces.

7. Conclusion

This paper has described the design of *VINI*, a virtual network infrastructure for supporting experimentation with network protocols and architectures in a realistic network environment. VINI complements the current set of tools for network simulation and emulation by providing a realistic network environment whereby real routing software can be evaluated under realistic network conditions and traffic loads with closed-loop experimentation. We first outlined the case for VINI, providing both design principles and an implementation-agnostic design. Based on this high-level VINI design, we have presented one instantiation of VINI on the PlanetLab testbed, *PL-VINI* . Our preliminary experiments in Section 5 demonstrate that *PL-VINI* is both efficient and a reasonable reflection of network conditions.

Once VINI is capable of allowing users to run multiple virtual networks on a single physical infrastructure, it may also ultimately serve as a substrate for new network protocols and services (making it useful not only for research, but also for operations). Because VINI also provides the ability to virtualize *any* component of the network, it may lower the barrier to innovation for network-layer

services and facilitate new usage modes for existing protocols. We now briefly speculate on some of these possible usage modes.

First, VINI allows a network operator to simultaneously run different routing protocols (and even different forwarding mechanisms) for different network services. Previous work has observed that operators occasionally route external destinations with an internal routing protocol (*e.g.*, OSPF, IS-IS) that scales poorly but converges quickly for applications that require fast convergence (*e.g.*, voice over IP) [12]. With VINI, a network operator could run multiple routing protocols in parallel on the same physical infrastructure to run different routing protocols for different applications.

Second, VINI could be used to help a network operator with common management tasks. For example, operators routinely perform planned maintenance operations that may involve tweaking the configurations across multiple network elements (*e.g.*, changing IGP link costs to redirect traffic for a planned maintenance event). Similarly, they may occasionally wish to incrementally deploy new versions of routing software, or test bleeding-edge code. A VINI-enabled network could allow a network operator to run multiple routing protocols (or routing protocol versions) on the same physical network, controlling the forwarding tables in the network elements in one virtual network at any given time, while providing the capability for atomic switchover between virtual networks.

VINI's future appears bright, both as a platform for both experimentation and more flexible network protocols and services. This paper has demonstrated VINI's feasibility, as well as its potential for enabling a new class of controlled, realistic routing experiments. The design requirements we have specified, and the lessons we have learned from our initial deployment, should prove useful as we continue to develop VINI and deploy it in various forms.

Acknowledgments

We thank Changhoon Kim, Ellen Zegura, and the anonymous reviewers for their comments and suggestions. This work was supported by HSARPA (grant 1756303), the NSF (grants CNS-0519885 and CNS-0335214), and DARPA (contract N66001-05-8902). We would also like to thank the many people at NLR and Abilene who are providing VINI with physical infrastructure, including rack space in their PoPs, bandwidth between sites, and upstream connectivity to the Internet.

REFERENCES

[1] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris, "Resilient Overlay Networks," in *Proc. Symposium on Operating Systems Principles*, pp. 131–145, October 2001.

[2] N. Feamster, D. Andersen, H. Balakrishnan, and M. F. Kaashoek, "Measuring the effects of Internet path faults on reactive routing," in *Proc. ACM SIGMETRICS*, June 2003.

[3] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A blueprint for introducing disruptive technology into the Internet," in *Proc. SIGCOMM Workshop on Hot Topics in Networking*, October 2002.

[4] 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. Networked Systems Design and Implementation*, March 2004.

[5] J. Touch and S. Hotz, "The X-Bone," in *Proc. Global Internet Mini-Conference*, pp. 75–83, November 1998.

[6] X. Jiang and D. Xu, "Violin: Virtual internetworking on overlay infrastructure," in *Proc. International Symposium on Parallel and Distributed Processing and Applications*, pp. 937–946, 2004.

[7] The GENI Initiative. `http://www.nsf.gov/cise/geni/`.

[8] GENI: Global Environment for Network Innovations. `http://www.geni.net/`.

[9] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov, "Designing extensible IP router software," in *Proc. Networked Systems Design and Implementation*, May 2005.

[10] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems*, vol. 18, pp. 263–297, August 2000.

[11] "OpenVPN: An open source SSL VPN solution." `http://openvpn.net/`.

[12] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *Proc. Networked Systems Design and Implementation*, pp. 49–56, May 2005.

[13] "ns-2 Network Simulator." `http://www.isi.edu/nsnam/ns/`.

[14] "SSFNet." `http://www.ssfnet.org/`.

[15] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. Symposium on Operating Systems Design and Implementation*, pp. 255–270, December 2002.

[16] "DETER: A laboratory for security research." `http://www.isi.edu/deter/`.

[17] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," in *Proc. Symposium on Operating Systems Design and Implementation*, December 2002.

[18] "WAIL: Wisconsin Advanced Internet Laboratory." `http://wail.cs.wisc.edu/`.

[19] "Open Network Laboratory (ONL)." `http://onl.arl.wustl.edu/`.

[20] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau, "Feedback-directed Virtualization Techniques for Scalable Network Experimentation," Tech. Rep. FTN-2004-02, University of Utah, May 2002. `http://www.cs.utah.edu/flux/papers/virt-ftn2004-02.pdf`.

[21] Linux VServers Project. `http://linux-vserver.org/`.

[22] D. Lowenthal, "PlanetLab Sirius Calendar Service." `https://snowball.cs.uga.edu/~dkl/pslogin.php`.

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

[24] M. Huang, "VNET: PlanetLab Virtualized Network Access," Tech. Rep. PDN–05–029, PlanetLab Consortium, June 2005.

[25] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir, "Experiences Building PlanetLab," Tech. Rep. TR–755–06, Princeton University, June 2006.

[26] "User-Mode Linux." `http://user-mode-linux.sourceforge.net/`.

[27] A. Bavier, M. Huang, and L. Peterson, "An overlay data plane for PlanetLab," in *Proc. Advanced Industrial Conference on Telecommunications*, July 2005.

[28] "Iperf 1.7.0: The TCP/UDP bandwidth measurement tool." `http://dast.nlanr.net/Projects/Iperf/`.