

In VINI Veritas: Realistic and Controlled Network Experimentation

Andy Bavier Nick Feamster† Mark Huang Larry Peterson Jennifer Rexford
† *Computing Science and Systems Division, Georgia Institute of Technology*
Department of Computer Science, Princeton University

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 the Planet-Lab testbed, 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.

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.

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]. While RON can be deployed and offer service to real users without modifying the underlying infrastructure, to evaluate its effectiveness, the system designers must wait for network failures to arise. Worse yet, without access to logs of (say) link fail-

ures in the underlying network or the ability to inject such failures—RON must rely on active probes to detect link failures, which introduces a tradeoff between probing overhead and measurement precision [2]. Ideally, researchers would be able to inject link failures at known times and collect precise measurements of RON's behavior during these events.

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. This capability enables 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 traveling between real end hosts, to enable measurements of end-to-end performance and the effects of feedback at the end systems.

Existing experimental infrastructures do not achieve all of these goals. For example, consider PlanetLab, which supports multiple simultaneous experiments running on hundreds of machines throughout the world [3, 4]. Researchers use PlanetLab to evaluate prototypes of new network services and to offer real services to end users, but PlanetLab does not provide a controlled environment. Although some networking researchers use PlanetLab as a platform for collecting measurement data, PlanetLab is not appropriate for evaluating new network protocols and mechanisms because it does not satisfy the first three goals above.

Instead, we are building VINI (Virtual Network Infrastructure), which will enable controlled, realistic experiments with new network 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 domains¹. 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 Networked Investigations (GENI) [5, 6] 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 failure modes of these components, as discussed in Section 3. In addition to facing similar challenges to testbeds like PlanetLab, we must deal with additional issues such as sharing routing-protocol port numbers across experiments, supporting multiple 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 the VINI software, 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 functional infrastructure. We use XORP for routing [7], Click for packet forwarding and network address translation [8], OpenVPN servers to connect with end users [9], and *rcc* for parsing router configuration data from operational networks to drive our experiments [10]. We use the PlanetLab nodes in Abilene for

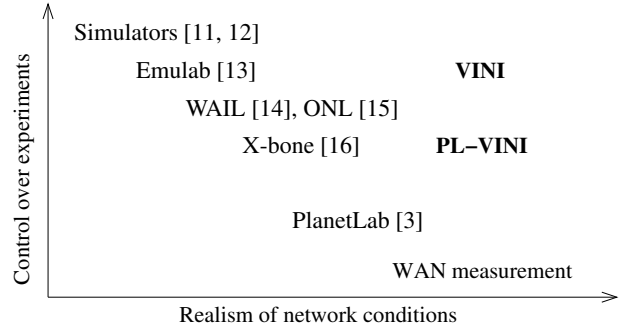


Figure 1: Comparing VINI to other evaluation techniques

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 address these three challenges, and extend the software to support multiple simultaneous experiments, we hope to provide the research community with an environment that offers not only increasing control and realism, but also a credible path to real-world deployment.

2. Related Work

VINI represents a unique design point among techniques for evaluating research ideas, as shown in the informal diagram in Figure 1. Starting in the upper left, network simulators like ns-2 [12] and SSFNet [11] give researchers a high degree of control but do not capture the operation of a prototype system running under real network conditions. Next, testbeds like Emulab [13] (and environments like DETER [17] built using Emulab), WAIL [14], and ONL [15] allow researchers to evaluate real systems under diverse configurations in a controlled environment, but do not carry real Internet traffic or experience real network conditions. Testbeds that run as overlays on the Internet, such as PlanetLab [3] and X-Bone [16], expose experiments to more realistic network conditions but offer less control over the environment. The lower right of the figure represents the many studies that analyze Internet measurements that capture realistic network conditions but give researchers no control over the underlying system. Finally, VINI will offer researchers a high degree of control over experiments running under realistic network conditions.

3. VINI Design Requirements

In this section, we flesh out the design requirements for a virtual network infrastructure. We focus in particular on the

¹We are in discussions with service providers about having dedicated upstream connectivity to the commercial Internet at a few exchange points.

general requirements of the infrastructure—and why we believe the infrastructure should provide those requirements— independent of how any particular instantiation of VINI would meet these requirements. At a high level, VINI’s design requirements are motivated by our attempt to solve the key question that we introduced in Section 1: how can VINI provide a network researcher the flexibility to perform controlled, realistic experiments with multiple network topologies on a single, fixed physical infrastructure? The answer to this question lies in a traditional concept: virtualization.

However, while virtualization is, in some sense, a “time tested” and conventional approach to solving many problems (e.g., in computer architecture, operating systems, and even in networked distributed systems), its application to computer networking is far from straightforward. VINI must provide a substrate for multiple virtual topologies on the same physical infrastructure; each virtual topology must provide the appearance of nodes that run routing protocols, forward packets, and connect to end hosts running in the “real world.” The remainder of this section tackles these problems. We discuss the challenges associated with embedding a realistic virtual topology on VINI’s physical infrastructure, with routing and forwarding on this topology, and with integrating each virtual network with the rest of the Internet. In addressing these challenges, our design decisions are driven 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.

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 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). None 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.

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 re-routing 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) 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 protocols and processes. These routing processes must 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 limit the changes to the PlanetLab OS, and instead 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* and IIAS prototypes address the problems outlined in Section 3. The table emphasizes that we must solve several problems in IIAS that would ideally be addressed by the VINI infrastructure itself. 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 physi-

cal 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) [18]. Each VServer is a lightweight “slice” of the node with its own namespace. Because of the isolation provided by PlanetLab, multiple VINI experiments can run on the same PlanetLab nodes simultaneously in different slices.

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 [19]). Similarly, the Linux hierarchical token bucket (HTB) scheduler [20] provides fair share access to, and minimum rate guarantees for, outgoing network bandwidth. Network isolation on PlanetLab is provided by a module called VNET [21] 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. Version 3.2 of the PlanetLab software provides support for *CPU reservations*. A CPU reservation of 25% provides the slice with a minimum of 25% of the CPU during the times that it is active. Additionally, if no “fair share” slices are running, then the slice with a CPU reservation can receive the unused cycles—this means it may potentially get more than its reservation. CPU reservations are not yet available to the larger PlanetLab community because policies for granting them have not yet been devised; however, *PL-VINI* is able to “beta test” this feature.

Even with a CPU reservation, a slice running a network experiment may experience jitter in gaining access to the CPU, depending on the processing demands in other slices. This makes it difficult to run repeatable experiments. We have extended the PlanetLab software to provide slices with the ability to *boost the priority* of a latency-sensitive process using Proper [22]. Proper permits an authorized *PL-VINI* slice to execute `/usr/bin/chrt` and assign a Linux real-time priority to its process. A process with real-time priority always jumps to the head of the run-queue, and also preempts any running non-real-time process when it wakes up. 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 extensions to PlanetLab provide greater isolation for a VINI experiment running in a slice. In Section 3.4 we describe several additional extensions we are exploring to provide even better isolation between *PL-VINI* slices.

Design Requirement	Solution	Where
Resource isolation between experiments	Virtual servers and network isolation in PlanetLab Extensions for CPU reservations and priorities	PlanetLab (4.1.1) PL-VINI (4.1.2)
Virtual point-to-point connectivity	Numbering TAP devices from common subnet Configuring tunnels and encapsulation in Click	PL-VINI (4.1.3) IIAS (4.2.1)
Distinct forwarding tables per virtual node	Separate instance of Click on each virtual node	IIAS (4.2.1)
Distinct routing processes per virtual node	Separate instance of XORP on each virtual node	IIAS (4.2.2)
Unique interfaces per experiment	Configuring XORP with dummy interfaces	IIAS (4.2.2)
Allowing end hosts to direct traffic through VINI	End-host connection to an OpenVPN server	IIAS (4.2.3)
Ensure return traffic flows back through VINI	Network address translation in Click on egress	IIAS (4.2.3)
Exposure of underlying topology changes	Upcalls of layer-3 alarms to virtual nodes	Future work (3.4)
Distinct external routing adjacencies	BGP multiplexer to share external BGP sessions	Future work (3.4)

Table 1: How our prototype of VINI satisfies the design requirements from Section 3

4.1.3 Interfaces on a Shared Subnet

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 uses Linux’s TUN/TAP driver, which supports user-space networking by providing both a virtual point-to-point IP network device (the TUN) and a virtual Ethernet device (the TAP). Both look like standard devices—that is, `ifconfig` shows them as IP tunnel `tun0` and Ethernet device `tap0`, respectively—but 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.

For *PL-VINI*, we created a virtual Ethernet device called `tap0` on every PlanetLab node, which is leveraged in two ways. First, `tap0` provides an easy and efficient way to get real traffic on and off an overlay—the intended use of the TUN/TAP driver. Second, we give each `tap0` device a unique IP address chosen from the 10.0.0.0/8 private address space to ensure that any two `tap0` devices seem to belong to the same network. This is crucial for supporting network experiments because routers only forward packets to “next hops” that are directly connected. However, PlanetLab nodes are actually computers in distinct IP subnets. Therefore, to get routing software like XORP to control an overlay data plane on PlanetLab, two “adjacent” overlay nodes must appear to be on the same network; `tap0` provides this illusion.

Although the TUN/TAP driver is standard in Linux, we modified the driver to preserve the isolation between different slices on PlanetLab. Every slice now sees a single `tap0` interface with the same IP address, but our changes allow multiple processes (in different slices) to read from `/dev/net/tun0` simultaneously, and each will only see packets sent by its own slice.

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 [23]:

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 don’t know anything 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 communities². IIAS employs the Click modular software router [8] as the forwarding engine, the XORP routing protocol suite [7] as the control plane, OpenVPN [9] as the ingress mechanism, and performs NAT (within Click) at the egress. We run IIAS on *PL-VINI*, meaning that IIAS can also use *PL-VINI*’s `tap0` device as an ingress/egress mechanism for applications running on a *PL-VINI* node.

4.2.1 Click: Links and Packet Forwarding

IIAS uses the Click modular software router [8] as its virtual data plane. Our Click configuration consists of four components that create the illusion of point-to-point links to other virtual nodes and enable the virtual nodes to forward data packets:

- **Local interface:** Click reads and writes 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 then are received by Click. Likewise, Click writes packets destined for `tap0`’s IP

²Compared to the preliminary work described in [23], our IIAS prototype incorporates a control plane (based on XORP) and a virtual interface for getting local traffic on/off the overlay (using our modified `tap0`).

address to `tap0`, 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.
- **UDP tunnels:** UDP tunnels (*i.e.*, sockets) are the links in the IIAS overlay network. Note that, somewhat counterintuitively, the Click configuration does not enforce a specific network topology—any node can send a UDP packet to any other, as long as the underlying physical network supports this. Rather, the topology of IIAS is configured entirely by XORP as described in Section 4.2.2.
- **Encapsulation table:** Click’s forwarding table maps the IP destination of each packet to a next hop in IIAS’s private 10.0.0.0/8 address space. The preconfigured encapsulation table matches the next hop to a UDP tunnel by mapping the private 10.0.0.0/8 address of the next hop to the public IP address of a PlanetLab node.

Three points about the IIAS data plane are worthy of note. First, IIAS is not tied to the 10.0.0.0/8 private addressing scheme; this addressing just made the most sense for running on *PL-VINI* based on the functionality provided by the `tap0` interface. Second, 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). Finally, though IIAS currently performs IP forwarding, we can also support new forwarding paradigms beyond IP—for example, one could implement DHT-based addressing and forwarding simply by writing new forwarding and encapsulation table elements.

4.2.2 XORP: Unique Interfaces and Routing

IIAS uses the XORP open-source routing protocol suite [7] 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 significant features of our XORP configuration, as well as the modifications we made to the XORP software to support deployment on *PL-VINI*, are described below.

The most interesting feature of our configuration is that it leverages XORP’s OSPF implementation not only to control routing within an IIAS network, but also to provide its very topology. In a normal setup, XORP would send multicast OSPF packets with TTL=1 in order to discover its neighbors. Because our Click configuration cannot yet handle multicast, we had to disable this feature and instead run XORP in “point-to-multipoint” mode. In this mode, each XORP instance is configured with a specific set of neighbors that XORP can install as “next hops” in Click’s forwarding table.

Because our Click configuration does not limit the connectivity between virtual nodes, the topology of IIAS is determined entirely by XORP: the inverse situation of a normal router setup. Changes in the underlying topology of *PL-VINI* do not necessarily affect the virtual topology of IIAS—a limitation that we discuss in Section 3.4. To inject network failures in IIAS, we instead configure particular Click tunnels with packet filters that drop all packets.

The main complication to running stock XORP on *PL-VINI* was a 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, and we would like to be able to specify different link costs for different neighbors. In our Click data plane, interfaces conceptually map to sockets and links to tunnels. Upon querying the OS, XORP sees only the physical interfaces present on the *PL-VINI* node, much as `ifconfig` would. To present XORP with a view of multiple physical interfaces, we enable its “dummy” FEA, usually used only for testing and debugging. The dummy FEA allows XORP to be configured with an arbitrary set of dummy interfaces that do not forward through an actual engine. We modified the dummy FEA to preserve these interfaces, while pushing real forwarding table entries for all of them down to Click.

Though we provide XORP with one dummy interface per neighbor, we must configure each one with `tap0`’s IP address. This requirement stems from the interaction between XORP, Click, and `tap0`. All of XORP’s packets pass through `tap0` to Click, which then forwards them through the data plane. Likewise, Click writes control packets destined to the local XORP through `tap0`, where they are delivered to XORP by the kernel. Local delivery is the reason why XORP’s interfaces must all use `tap0`’s IP address. If OSPF packets to the local XORP have a destination IP address that is not the same as `tap0`’s address, then the kernel thinks that these packets are non-local and drops them. Similarly, if the IP destinations on OSPF packets delivered to XORP do not match that of the dummy interface (*e.g.*, packets from a neighbor do not appear to have arrived on that neighbor’s interface), then XORP drops them.

We made two changes to XORP to support multiple interfaces with the same IP address: (1) we configured each “dummy” interface as the endpoint of a point-to-point link; and (2) we modified XORP to demultiplex incoming packets to interfaces based on the source IP address. Each neighbor is placed on its own dummy point-to-point link and configured as the remote endpoint. Our demultiplexing code matches incoming packets to dummy point-to-point interfaces by matching the source IP address with a point-to-point link’s remote address, *i.e.*, the neighbor that sent it.

In summary, an important feature of IIAS is that it decouples the control and data planes by putting the routing protocol in a different virtual world than the forwarding engine. Though we modified XORP to create its own virtual world, we are also evaluating running an unmodified XORP in User-Mode Linux, while leaving the Click outside, as discussed in Section 3.4. In fact, decoupling the control and data planes

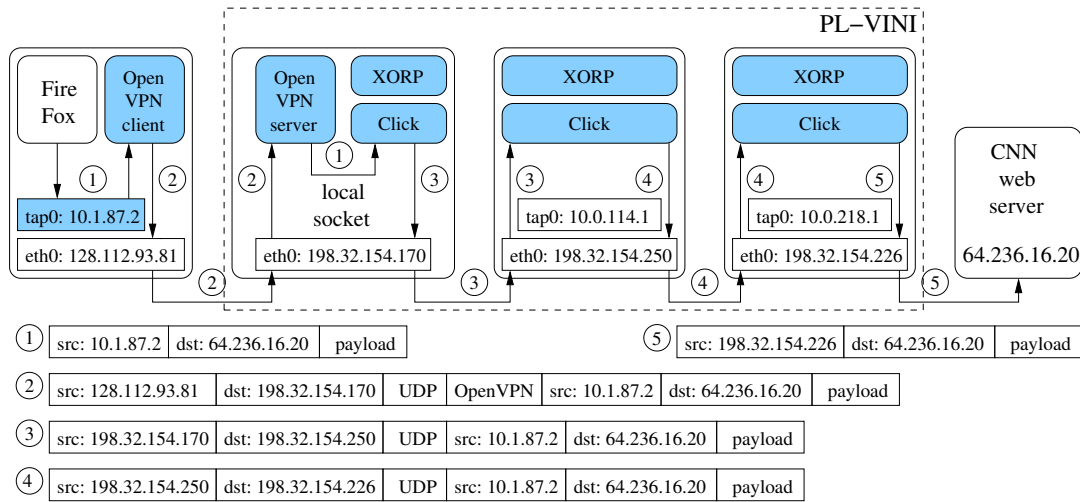


Figure 2: The life of a packet in IAS (shown shaded) running on PL-VINI (dotted box)

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 IIAS nodes themselves. IIAS uses OpenVPN [9] 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.

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. Currently NAPT is only supported for TCP and UDP packets. 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, XORP uses it to send OSPF packets to its neighbors, and in our 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 consults its forwarding table and discovers that the next hop for a packet to 64.236.16.20 is 10.0.114.1. Click consults the encapsulation table to map the 10.0.114.1 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.
4. The Click process running on 198.32.154.250 receives the original packet from the UDP tunnel (*i.e.*, with the encapsulation headers stripped). Click consults its forwarding table and discovers that the next hop for a packet to 64.236.16.20 is 10.0.218.1. Click consults the encapsulation table to map the 10.0.218.1 address to 198.32.154.226, and sends the packet over a UDP tunnel to the latter address.
5. The Click process running on 198.32.154.226 receives the original packet from a UDP tunnel, consults the forwarding table, and discovers that it is the egress

node for 64.236.16.20. Click sends the packet through its NAT 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 experimentation, *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 [17], which is based on Emulab [13]; 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



Figure 3: DETER topology for microbenchmarks

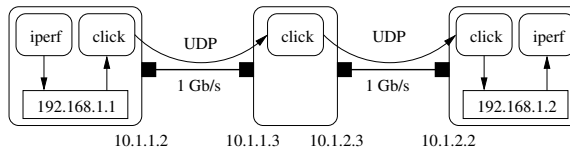


Figure 4: Overlay topology on DETER

CPU reservations and real-time priority reduce CPU contention.

The microbenchmark experiments are run using `iperf` version 1.7.0 [24]. We measure capacity using `iperf`’s TCP throughput test 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 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. Clearly IIAS is not nearly as efficient as the network alone: it manages to

	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

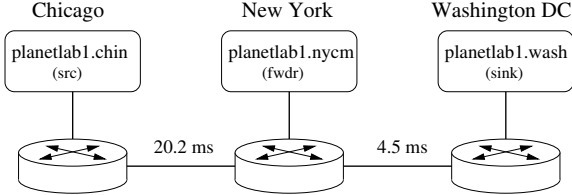


Figure 5: PlanetLab topology for microbenchmarks

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 $5\mu\text{s}$ 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 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 $130\mu\text{s}$ 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 DC 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 back-

	Default share		PL-VINI	
	mean	stddev	mean	stddev
Network	0.35	0.3	0.27	0.16
IIAS	2.4	3.7	1.3	0.9

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

bone 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 that CPU contention in particular is likely to be a problem for *PL-VINI* on PlanetLab; however, *PL-VINI* offers two solutions: CPU reservations and real-time priorities. We wish to measure the effects of CPU contention on our experiment as well as evaluate the resource allocation capabilities of *PL-VINI*. Therefore, we run our experiments from Section 5.1.1 using three sets of CPU scheduling parameters: a default fair share, a CPU reservation of 25%, and a 25% CPU reservation plus giving the IIAS Click process real-time priority (we refer to the last scenario as “*PL-VINI*”, since it makes use of both of *PL-VINI*’s resource allocation knobs). Our expectation is that the CPU reservation will improve the overall capacity of IIAS by giving it more CPU, while the real-time priority will reduce the scheduling latency of the Click process and so improve end-to-end latency observed in IIAS.

Table 4 shows the results of the bandwidth test with each set of parameters. We note three things. First, the CPU reservation increases the measured performance and decreases the observed variability in the underlying network; this is because *iperf* itself is scheduled more regularly. Second, the CPU reservation alone provides somewhat better performance and less variability to IIAS, but it still falls far short of the raw network in this case. Third, with both a 25% CPU reservation and real-time scheduling provided by *PL-VINI*, IIAS approaches the underlying network in both observed throughput and variability of the result. (Note that, since only the Click process receives real-time priority, the measurements of the raw network are the same as with only a CPU reservation.) 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 and even a CPU reservation: the standard deviation in *PL-VINI* ping times is over 20X that of the network. This is not a surprise, since we did not expect that a CPU reservation would improve scheduling latency in the IIAS slice. Rather, as expected, *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 network.

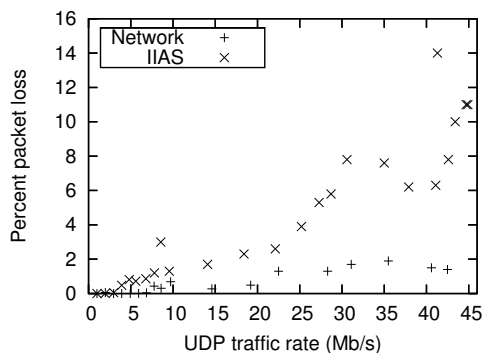
Table 6 again shows that *PL-VINI*’s combination of CPU reservations and real-time priorities is useful to IIAS. We ran CBR streams between 1Mb/s and 50Mb/s on the network and overlay; in our experiments, jitter did not appear to be corre-

	Default share			25% resv			PL-VINI		
	Mb/s	stddev	CPU%	Mb/s	stddev	CPU%	Mb/s	stddev	CPU%
Network	86.1	1.40	N/A	90.8	0.53	N/A	90.8	0.53	N/A
IIAS	22.5	4.01	13	35.1	2.92	20	86.2	0.64	40

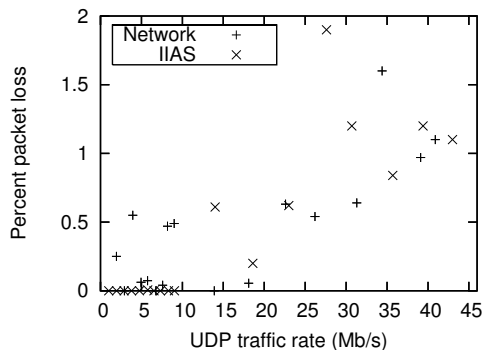
Table 4: TCP throughput test on PlanetLab

	Default share					25% resv					PL-VINI				
	min	avg	max	mdev	loss	min	avg	max	mdev	loss	min	avg	max	mdev	loss
Network	24.4	24.5	28.5	0.2	0%	24.4	24.5	28.2	0.2	0%	24.4	24.5	28.2	0.2	0%
IIAS	24.7	27.7	80.9	4.8	0%	24.7	27.8	97.5	4.9	0%	24.7	25.1	28.6	0.38	0%

Table 5: ping results on PlanetLab; units are ms



(a) With default share



(b) With PL-VINI

Figure 6: Packet losses in IIAS on PlanetLab

lated with stream size and so we report the the jitter results across all streams. (Since the ping experiment showed that CPU reservations alone have little effect on the fine-grained network behavior, we omit this case from the jitter experiment.) Here we see that running IIAS on *PL-VINI* halves the mean jitter and reduces the variation in test results by 75%.

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

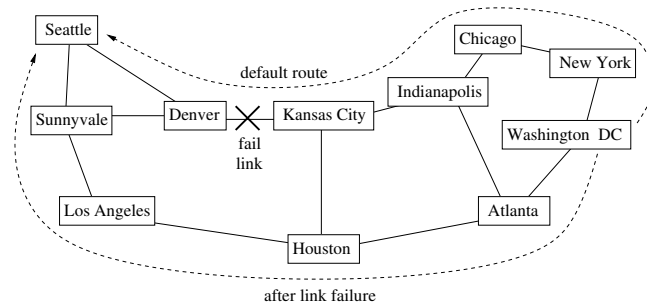


Figure 7: Abilene Topology

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 in *PL-VINI*: here, we measure packet loss in IIAS 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 on PlanetLab 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.

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

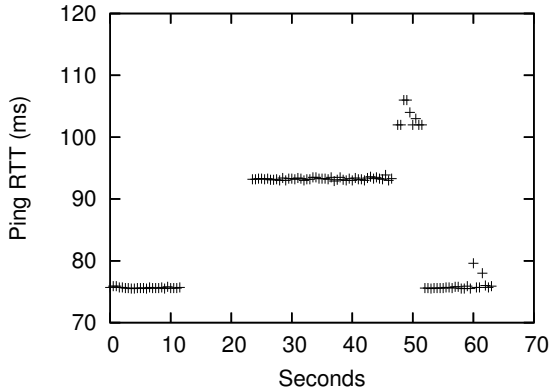


Figure 8: Observing OSPF route convergence (using ping)

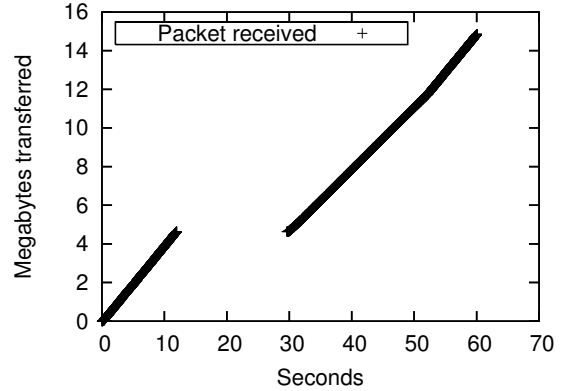
virtual link (UDP tunnel) connecting two Abilene nodes, and at the same time deleting the interface from XORP’s OSPF configuration³. We use ping, iperf, and tcpdump to measure the effects on data traffic. Experiments such as this can help researchers study routing pathologies that are difficult to observe on a real network, where a researcher has no control over network conditions.

Figure 8 shows the effect on ping times between DC and Seattle of failing the link between Kansas City and Denver 12 seconds into the experiment, and restoring the link at time 38 seconds. Initially, IAS routes packets from DC through New York, Chicago, Indianapolis, Kansas City, and Denver to Seattle, with a mean round-trip time (RTT) of 76 ms. When the link fails at time 12, XORP’s OSPF takes 12 seconds to find a new route through Atlanta, Houston, Los Angeles, and Sunnyvale, with a mean RTT of 93 ms⁴. After the link comes back up at time 38, we see another 12 seconds of delay before the RTT increases briefly to 103 ms. We hypothesize that these larger RTTs correspond to an intermediate stage in the convergence process, where data packets are briefly traversing longer paths or even a forwarding loop. Finally, 52 seconds into the experiment, *PL-VINI* returns to the initial state with an RTT of 76 ms.

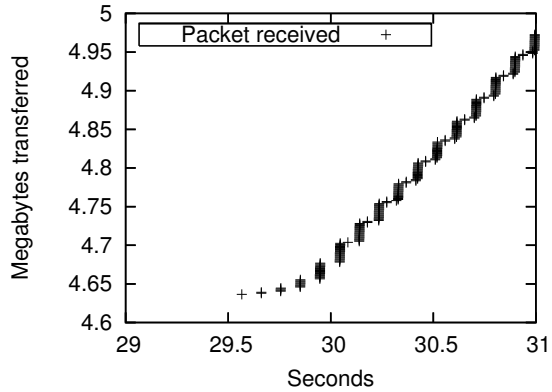
Figure 9 shows another run of the same experiment, this time using *iperf* to send a bulk TCP transfer from Washington DC to Seattle. The TCP window size is set at *iperf*’s default of 16 KB, so TCP’s throughput is limited by the window size to around 3 Mb/s. The figure plots the arrival time of data packets at the receiver, as reported by *tcpdump*. At time 12 we fail the link and packets stop getting through, as shown in Figure 9(a); at approximately time 30 a new route is found and the TCP transfer continues. Figure 9(b) shows what happens at time 30 in more detail. Here we can clearly see TCP slow-start restart in action and each individual win-

³Dropping all packets on the virtual link should be enough to cause OSPF to choose new routes. However, due to a bug in XORP, we also needed to manually modify XORP’s OSPF configuration to trigger it to send out new LSAs. We have notified the XORP team about the problem and plan to rerun the experiments once the bug is fixed. Although not our initial goal, this experience illustrates that our IAS deployment provides a valuable environment for testing routing software under a wide range of conditions.

⁴XORP has a built-in 5-second delay in recalculating the route table, and the script that modifies XORP’s OSPF configuration takes 3 seconds. Exploring the remaining sources of convergence delay is the subject of ongoing work.



(a) Total bytes transferred



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

Figure 9: TCP throughput during OSPF routing convergence

dow of data arriving at the receiver. The link is brought back up at time 38. At time 52, the throughput increases slightly after the routing protocol converges back to the old path with the smaller round-trip time.

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 IAS, since *PL-VINI* enables IAS to behave like a real network on PlanetLab.

6. Ongoing Work

In this section, we discuss our ongoing work on VINI, as briefly summarized in Table 1. In particular, we have yet to address a few of the design goals from Section 3. In this section, we describe possible solutions to these problems, which fall into three main areas: (1) exposing failures at lower layers, (2) supporting simultaneous experiments, and (3) providing more experimental control. After revisiting these issues and the solutions we are exploring, we briefly discuss ways to improve the isolation between *PL-VINI* slices.

6.1 Exposing Underlying Topology Changes

As discussed in Section 3.1, the failure or recovery of a physical component should affect each of the associated virtual components. Our PL-VINI prototype does not achieve this goal because the underlying network automatically reroutes the traffic between two IAS 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.

6.2 Distinct External Routing Adjacencies

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 VINI routing software could introduce instability into neighboring networks, and the rest of the Internet).

To avoid these potential issues, we plan to design and implement a multiplexer that manages the 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 would ensure that each virtual node announces only its own address space and potentially impose limits on the rate of BGP update messages that are propagated from each experiment. The multiplexer could be built as an extension to route-reflector functionality already available in existing open-source routing software such as Quagga [25].

6.3 Experimental Control

Beyond the existing support for constructing arbitrary topologies and failing links, VINI should also provide the ability to *specify* experiments. In an *ns* simulation [12], 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 [13] 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, 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 the specification of an actual experiment would be a part of a larger experimental workflow, where as-

pects of the 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 PlanetLab PoPs) for a VINI experiment from the actual Abilene routing configuration, exploiting the configuration-parsing functionality from previous work on router configuration checking [10]. Eventually, we intend to augment VINI to incorporate more of the routing configuration into XORP and Click and also support playback of routing traces.

6.4 Better Isolation

As discussed in Section 3.4, VINI should be able to support multiple simultaneous experiments with strict resource guarantees for each slice. 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.

Recall from Section 4 that, in order to create the appearance of multiple interfaces, we needed to modify XORP to allow configuration of dummy interfaces. To allow XORP to run unmodified in PL-VINI, we are currently experimenting with extensions to PL-VINI to run XORP in User-Mode Linux (UML). Our preliminary results suggest that running Click in UML would compromise packet-forwarding performance too much. As such, we are investigating ways to run XORP in UML separately from Click, as well as running Click in the kernel, on dedicated hardware, or both.

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) [10]. 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 network 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.

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. USENIX/ACM Symposium on Networked Systems Design and Implementation*, March 2004.
- [5] The GENI Initiative.
<http://www.nsf.gov/cise/geni/>.
- [6] GENI: Global Environment for Network Innovations.
<http://www.geni.net/>.
- [7] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov, "Designing extensible IP router software," in *Proc. USENIX/ACM Symposium on Networked Systems Design and Implementation*, May 2005.
- [8] 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.
- [9] "OpenVPN: An open source SSL VPN solution."
<http://openvpn.net/>.
- [10] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *Proc. USENIX/ACM Symposium on Networked Systems Design and Implementation*, pp. 49–56, May 2005.
- [11] "SSFNet." <http://www.ssfnet.org/>, 2003.
- [12] "ns-2 Network Simulator."
<http://www.isi.edu/nsnam/ns/>, 2000.
- [13] 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.
- [14] "WAIL: Wisconsin Advanced Internet Laboratory."
<http://wail.cs.wisc.edu/>.
- [15] "Open Network Laboratory (ONL)."
<http://onl.arl.wustl.edu/>.
- [16] J. Touch and S. Hotz, "The X-Bone," in *Proc. Global Internet Mini-Conference*, pp. 75–83, November 1998.
- [17] "DETER: A laboratory for security research."
<http://www.isi.edu/deter/>.
- [18] Linux VServers Project.
<http://linux-vserver.org/>.
- [19] D. Lowenthal, "PlanetLab Sirius Calendar Service."
<https://snowball.cs.uga.edu/~dkl/pslogin.php>.
- [20] Linux Advanced Routing and Traffic Control.
<http://lartc.org/>.
- [21] M. Huang, "VNET: PlanetLab Virtualized Network Access," Tech. Rep. PDN-05-029, PlanetLab Consortium, June 2005.
- [22] S. Muir, L. Peterson, M. Fiuczynski, J. Cappos, and J. Hartman, "Proper: Privileged Operations in a Virtualised System Environment," in *Proc. USENIX*, (Anaheim, CA), pp. 367–370, April 2005.
- [23] A. Bavier, M. Huang, and L. Peterson, "An overlay data plane for PlanetLab," in *Proc. Advanced Industrial Conference on Telecommunications*, July 2005.
- [24] "Iperf 1.7.0: The TCP/UDP bandwidth measurement tool."
<http://dast.nlanr.net/Projects/Iperf/>.
- [25] "Quagga software router."
<http://www.quagga.net/>, 2006.