Automated Deployment and Customization of Routing Overlays on Planetlab

Claudio Daniel Freire, Alina Quereilhac, Thierry Turletti, and Walid Dabbous

INRIA, Sophia Antipolis, France
{claudio-daniel.freire,alina.quereilhac,
thierry.turletti,walid.dabbous}@inria.fr

Abstract. PlanetLab testbed is widely used to evaluate protocols and applications under realistic Internet conditions, but this realism comes at the cost of uncontrolled topology and traffic behavior. The use of overlay networks on PlanetLab can solve this problem by giving more control to the experimenter. However, manually creating such overlays is far from simple, and existing solutions are either not available for all PlanetLab nodes, or lack support for low level overlays. Deployment and customization of overlay architectures are also poorly supported. In this paper we present a flexible solution to support overlay networks on PlanetLab, providing deployment automation, tunneling, routing, and traffic shaping capabilities. By building our solution into NEPI, a general framework for network experimentation, which automates design, deployment, and management of experiments, we simplify the complexity of building overlays on PlanetLab, and foster reusability and extensibility though NEPI's modular structure.

Keywords: networking, overlays, PlanetLab, NEPI.

1 Introduction

PlanetLab[4] is a globally distributed testbed composed of many nodes connected to the Internet. It provides support for the development of new network technologies, enabling researchers to evaluate new protocols and applications under realistic Internet conditions. It is difficult, or even impossible, to achieve the same level of realism using alternative experimentation environments, such as simulators [22], and this is what makes PlanetLab so valuable. However, this realism comes at the cost of a lack of control over the many factors that influence the outcome of an experiment, such as links, nodes, and external traffic conditions, since the researchers have no control over the Internet itself.

The use of overlay topologies on top of the Internet can mitigate this problem by giving more control to the experimenters. Overlays can be used to force specific routing topologies [2], or to analyze real traffic conditions[1,3] with a degree of fault isolation.

Still, creating overlays on top of PlanetLab is a non-trivial task because of the constraints imposed by the type of network virtualization [16,7] used to

T. Korakis, M. Zink, and M. Ott (Eds.): TridentCom 2012, LNICST 44, pp. 240–255, 2012.

allow multiple experiments to run independently in the same nodes. Setting up routes, tunnels, or IP filters, cannot be achieved by simply invoking appropriate system calls as in classical (non-virtualized or fully virtualized) environments, but requires instead the use of PlanetLab-specific tools.

Previous efforts have been made to enable the creation of overlay networks on PlanetLab, which were mainly focused on the tunneling and routing aspects. Existing solutions like PL-VINI [5] are limited because they depend on extensions or infrastructure that is not available for all nodes, and thus they do not currently work on the whole PlanetLab network. Other alternatives like Splay [9], do not suffer from this limitation, but can only be used with application-level overlays. While these solutions do provide enabling technology to build overlays on top of PlanetLab, they do not solve the heavy load of manual work involved in the customization and deployment of concrete overlay experiment scenarios. In general, they do not provide enough tools to easily customize overlays, and only perform a subset of the tasks needed to run and manage experiments on top of them.

In this paper we present a solution for automatic creation and customization of overlay experiment scenarios on PlanetLab, based on NEPI [12,13], the Network Experimentation Programming Interface. Similar to other tools, our work enables the creation of virtual links over the Internet, allowing both, layer-2 switching and layer 3-routing, between PlanetLab nodes. Nevertheless, by building our solution into NEPI, we provide support for the different steps involved in the construction of overlay experiment scenarios, integrating the design, configuration, and automatic deployment on top of PlanetLab in one single tool. We refer to this aspect of experimental overlay construction as *deployment automation*. Moreover, our solution provides all the necessary mechanisms for overlay customization, including custom queues and transmission mechanisms, and the functionalities required to run applications in the overlay, like IP routing.

Efficient and flexible *tunneling* between arbitrary PlanetLab nodes is accomplished by providing several tunneling alternatives. Like RiaS [6] and Trellis [8], we support layer-2 and layer-3 tunnels with options for UDP, TCP and GRE encapsulation. However, in contrast with previous work, our solution does not require specially tailored nodes, and can be deployed in any node in the PlanetLab network.

Routing table manipulation was not possible before due to the lack of an appropriate system interface. We enhanced *routing* capabilities in PlanetLab by extending the *vsys*[11] interface to allow scalable, secure, and cooperative manipulation of nodes' routing tables, a capability on top of which our solution builds application-transparent overlay networks.

Our solution was designed to enable the use of experimental queuing and aggregation methods with low implementation overhead. It allows the researcher to easily perform *traffic shaping* within experimental overlays.

Using these tools, researchers can experiment with routing overlays on PlanetLab with minimal effort, and through NEPI they can perform the experiments repeatedly and automatically in a controlled way.

The focus of this paper will be set on the techniques used to support deployment automation, tunneling, routing, and traffic shaping capabilities for the construction of routing overlays on PlanetLab using NEPI, which we consider the be the core contributions of this work.

The rest of the paper is organized as follows: we begin by discussing the challenges involved in the development of our solution, then we cover the implementation details of NEPI overlay construction and customization support for PlanetLab. We finally evaluate the solution with a concrete use case showing how the tools we develop can benefit to the evaluation of networking protocols with increased accuracy.

2 Related Work

Two of the most complete previous solutions for low-level overlay deployment in PlanetLab are PL-VINI [5] and RiaS [6]. Both have serious drawbacks that limit their usability.

PL-VINI is an implementation of a virtual network infrastructure on PlanetLab, that uses User Mode Linux virtual machines [17] to provide full network virtualization for independent experiments running in the same node. Contrary to the Linux-VServer [16] container-based virtualization approach used natively in PlanetLab, in PL-VINI, virtual machines in the same node have direct access to the kernel network stack, making it possible to trivially manipulate routing tables and create tunnels without interfering each other.

PL-VINI has a good performance and enables to easily create layer-2 overlay topologies, but only around 40 nodes out of the many (1040) nodes in PlanetLab support it. It depends on a set of extensions to PlanetLab that have not been deployed on all nodes, and access to dedicated infrastructure that is also not globally available, making it not suitable for big scale deployments in PlanetLab.

While PL-VINI allows the implementation of custom routing and queuing algorithms, this can only be done through the Click modular router [5,21], which precludes the possibility to reuse prototype code. In contrast, our solution was crafted upon the idea of easy re-usability and it does not relinquish overlay customization to a single application, thus it makes the researcher's task of testing prototype algorithms in realistic conditions easier. Furthermore, we provide methods to run the experimental code in any PlanetLab node, expanding available resources beyond what PL-VINI provides.

Trellis [8], is another platform for network virtualization which implements container based virtual machines by using Linux network namespaces [18]. Although being the most performant alternative out there, by constructing its tunnel implementation using GRE and kernel-mode switches, it suffers from scalability issues. It creates one bridge and four taps (at least) per tunnel, all connected via virtual (software) switches, stateful and expensive to maintain objects. If every experiment running in a PlanetLab node created such tunnels, the kernel would be overwhelmed very easily. Trellis also does not respect the administrative and fair share bandwidth limits that are imposed per node

(and critical) to PlanetLab. Although it does provide bandwidth management, it is not integrated with PlanetLab's mechanism, so large-scale deployment of the technique would be impractical. Like PL-VINI, it also would require heavy changes in the PlanetLab system to enable its wide deployment, and thus it remains an unavailable solution on most PlanetLab nodes.

RiaS [6], on the other hand, is supported by all the nodes in PlanetLab. It is built upon an architecture of user-mode packet forwarders that create a network of *point-to-point* links with layer-2 tunnels. By operating in user-mode, it scales significantly more graciously than previous approaches under pressure, since it is subject to PlanetLab's fair-share scheduler. However, its support for layer-3 tunneling is limited because nodes with more than one interface (routers) in the overlay topology are required to be PL-VINI nodes, thus for layer-3 tunneling RiaS shares PL-VINI and Trellis limitations.

RiaS is heavily focused on layer-2 experiments, and because of this its architecture is ill-suited for layer-3 overlays. It only implements layer-2 routing techniques, to circumventing PlanetLab nodes' inability to set IP routes at the kernel level for one, and to support experimentation with arbitrary layer-2 protocols. It works more like a switch than a router. This imposes limitations on the experiments, since RiaS by itself provides no layer-3 routing, and nodes cannot reach other networks. Since RiaS does it all in user mode, it incurs heavy CPU overhead, resulting in artificial bandwidth and scalability limits, and packet loss well above that experienced in the underlying network. Rias does provide some support for overlay deployment automation, in the form of a Resource Allocator and Virtual Network Mapper tools. Although these tools consider the topology level of an experiment, they do not cover the application level.

Other solutions like Splay [9] and Plush [14], do not even support low level overlays (i.e. at layer 2 or 3), but rather they build application-level overlays. Splay [9] is a good example, as it automates deployment both at topology and application levels, has good performance, and is easy enough to use. However, it only handles packets at the application level. Even its packet loss models apply at the application level, which is an inaccurate rendition of link packet loss in most cases. It is also not able to handle all applications, since they must be written in Ruby using splay-specific support libraries.

PlanetLab support in NEPI was built to address the need to easily create and customize overlays, both at topology and application levels, a problem which current solutions do not fully address.

3 Challenges

PlanetLab [4] uses container-based virtualization to allow multiple experiments to run independently on the same node while sharing its resources. The implementation of network virtualization [7], however, has a cost: some tasks are made more complex than in a classical (non-virtualized or fully virtualized) environment. Slivers (isolation units for experiments running in PlanetLab) do not have full root privileges, so system calls to set up routes, tunnels, or IP filters,

among others, are forbidden, requiring the use of PlanetLab-specific tools to perform these tasks. These limitations make many of the existing tools for overlay creation unusable within PlanetLab [10].

In order to build overlays on PlanetLab, the first challenge that needs to be addressed is tunneling. Packets have to be captured, encapsulated, and transmitted over the Internet to their destination in a way that is transparent for applications, and neutral for the intervening networks. Since PlanetLab nodes host many slivers, 300 in average, which are container-based virtual machines that share the same network stack, lower layer tunnels have to be implemented to respect network isolation, in an environment of high concurrency.

User mode tunnels have their own challenges too. Since only the kernel knows when a link is congested, and it will happily discard outgoing packets in this case. To avoid packet loss at the kernel's queue, which is undesirable when applying customized queuing, it is necessary to limit packet egress rate to match available bandwidth. But bandwidth can only be guessed, so user mode queues become delicate to use.

Routing packets is another key challenge. Kernel routing table virtualization is not able to handle as many slivers as are normally present in a node, so a technique that shares resources safely is key to achieve routing successfully.

Per-sliver routing, a technique designed for UMTS/3G-connected nodes in PlanetLab in [20] and embodied as a PlanetLab *vsys* script, allows slivers to create their own private routing tables. This technique has scalability issues since it only supports 150 concurrent slivers on each node, whereas the average on PlanetLab is around 300.

Another problem that requires special attention is the testbed's unstable and unreliable nature. Nodes can be brought up and down without any warning, or be overloaded and unresponsive, and their state is not always faithfully mirrored in the PLC API which we use for resource discovery. Our automation procedures had to be made resilient on these situations, otherwise deployment of big experiments resulted in unacceptably high failure rates.

4 PlanetLab Overlays in NEPI

NEPI [12,13] is a framework for network experimentation that automates experiment design, deployment, and control, providing a uniform way of interacting with different testbeds. NEPI provides an *experiment description* (ED) language that allows researchers to design experiments, both at topology and application levels, by adding and connecting testbed-specific component abstractions, such as nodes and links. New abstractions can be implemented to support other testbeds or to add new functionality.

An *experiment controller* (EC) entity is in charge of orchestrating the experiment from the given experiment description. The EC is responsible for allocating resources, configuring components, running applications, and retrieving experiment results. It also handles the coordination of the (possibly several) testbeds

involved in the experiment. The EC is independent from any specific testbed, providing a large body of pre-existing automation functionality.

Because of these characteristics NEPI was an ideal choice to support automated deployment and overlay customization on PlanetLab. Our work was focused, then, on adding the PlanetLab specific functionality to allow NEPI to deploy experiments on PlanetLab. This not only included implementing the necessary abstractions on NEPI, but also modifying the available interface in PlanetLab to add the required routing and tunnel-enabling functionality.

In this section we will describe how overlays are created and customized on PlanetLab using NEPI. We will also explain the mechanisms developed to support network-level routing overlays in PlanetLab, that is, overlays that route IP or Ethernet packets. Other kinds of overlays are application-specific, and thus require application-specific abstractions that are beyond the scope of our work. The new mechanisms we contributed to PlanetLab to enable routing and tunneling, can be invoked by any user and application through the *vsys* interface. With them, any PlanetLab user can now easily set up GRE links, and manipulate routing tables.

4.1 Deployment Automation

In order to automate deployment of experiments involving overlays, an experiment must first be described in a machine-understandable way. NEPI provides

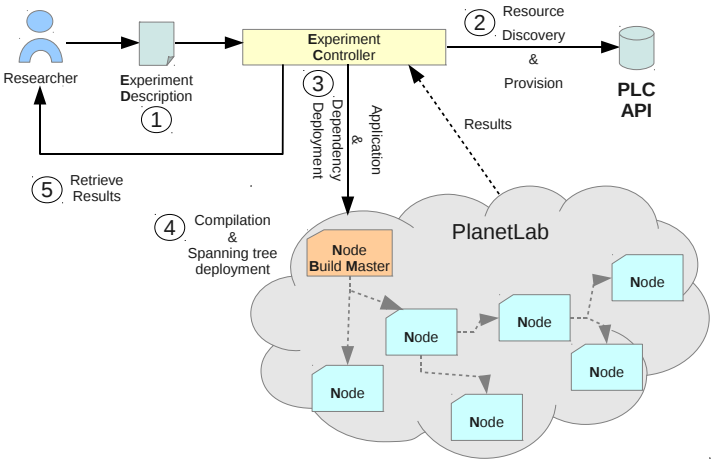

Fig. 1. NEPI-managed PlanetLab overlays. (1) The user creates the Experiment Description (ED) and passes it to the Experiment Controller (EC). (2) The EC uses the instructions in the ED to discover and provision available resources. (3) The EC instructs the allocated PlanetLab nodes to perform compilation and installation of applications. (4) Build masters prepare application data which is then distributed Peer-to-Peer among all nodes prior to experiment execution. (5) Results are relayed back to the researcher automatically upon completion.

an experiment description language based on interconnecting experiment component abstractions, called *boxes*. These boxes can be grouped into two categories: *topology*, and *application*.

To describe the topology-level of a PlanetLab experiment, we added boxes to represent PlanetLab *nodes*, *interfaces*, and *internet access*. Picking nodes is a critical design step. Since experiments can be affected by the presence of overloaded nodes, we allow researchers to specify constraints on various node metrics that will be resolved at deployment time. An experiment could accept any node, or it might require nodes with some amount of CPU or bandwidth unused. Or, the other way around, we might need overloaded nodes, perhaps, if they were trying to evaluate what happens when a node in the network is congested. All those criteria can be useful for reproducing previous experiments.

To describe the application level, we added *application* and *application-dependency* boxes, which can be connected to a node box, and represent an instruction to deploy the application or dependency, and run it on the specified node. Application boxes allow the specification of all the dependencies needed for deployment: packages they depend on, to be automatically installed in the nodes, and user programs to be launched, built and installed if necessary.

NEPI supports the concept of application traces, which capture the output of applications or even network traffic. NEPI can automatically gather all the traces at any point during execution, for inspection or analysis.

In contrast with Splay [9] and Plush [14], during experiment execution, NEPI acts as a PlanetLab controller which can manage PlanetLab slices (groups of slivers) and execute commands within nodes, rather than an agent that runs directly in PlanetLab. NEPI only gives instructions to PlanetLab nodes the way regular users would, using SSH. In this sense, our solution can be regarded as a big, specialized scripting engine. The main benefits of this approach arise from its flexibility: researchers can easily add functionality to it, and they do not require any modifications made to PlanetLab itself, making it very easy to build a range of reusable experimentation modules for deploying common topologies or utilities in PlanetLab slices. Also, experiments descriptions remain valid and usable for experiment reproduction, since NEPI and PlanetLab are decoupled and changes to one do not usually affect the other

Once given access to the slice, NEPI uses the PLC API to perform resource discovery, based on the criteria specified by the researcher in the experiment description, and provision the nodes adding them to the slice.

To map overlay nodes into PlanetLab nodes, we implemented an algorithm that constructs a set of viable hosts for each node, based on explicit and implicit constraints. Implicit constraints include the number of real interfaces as specified in the experiment description, and capabilities as required by the presence or absence of virtual interfaces and custom routing tables. After such a set is built for all nodes in the experiment, a simple backtracking procedure constructs a subset of all viable assignments. Not one solution, but a whole class of solutions are represented in a way that lets NEPI pick the best one in terms of *qualitative health* metrics, such as node load and reliability.

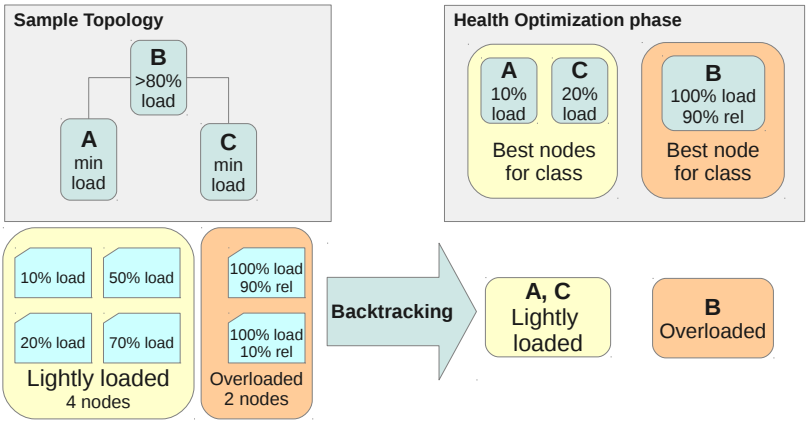

Fig. 2. Resource allocation strategy. Available nodes are partitioned into categories defined by the constraints derived from the experiment description. Each node will require assignment from one or more categories. A backtracking algorithm computes a set of solutions represented as an assignment node-to-category. Since each category contains more than one node, the final solution is built using only the healthiest nodes of the category (i.e. taking into account the node reliability).

NEPI obatins these node *qualitative health* metrics, tracked by the CoMon [15] tool, through the PLC API. If apparently healthy nodes turn out not to be so, NEPI adds those *"seemingly live"* nodes to a blacklist. With these precautions, allocation on nodes with unstable communication is minimized, and thus deployment success rates improve, which is a real problem when deploying large experiments consisting of a great many nodes.

In order to achieve a certain level of *reproducibility*, we had to take into account the constantly changing landscape of PlanetLab. The controller will record effective running parameters, like node load at the moment of deployment, physical locations, available bandwidth. When those parameters are not constrained by experiment design, effective execution values will provide all the information needed for posterior experiment reproduction, even when resource availability has changed. Later runs can use this information to constrain node selection in a way that resembles prior executions, or that ensures sufficient resource availability.

Once the nodes are successfully added to the slice and become responsive, NEPI coordinates the process of deploying applications and dependencies. Application resources, such as source code, are copied over to a few select nodes, called the build masters. These build masters take care of building applications, and downloading required rpm packages. In this way we avoid inefficient use of PlanetLab's resources - binaries only need to be built once per architecture, and bandwidth can be better utilized by sharing required downloads in a peer-to-peer fashion, preferring fast connections over slow ones. The rest of the nodes, the build slaves, wait for their master to be done to copy the resulting binaries and rpms from them.

When everything is ready, the controller launches the applications. The researcher can further interact with the experiment in realtime, modify certain configurations, start and stop applications, and retrieve results though the controller's API.

4.2 Tunneling

NEPI supports tunneling both at layer 2 and layer 3, and provides encapsulation over TCP, UDP, and GRE. Encapsulation over GRE is highly efficient since it is supported at the kernel level. UDP and TCP encapsulation, in contrast, requires user-mode packet forwarding, which adds processing overhead. However, it is better suited for overlay customization, since the user-mode packet forwarding daemon used by NEPI has been coded to support easily pluggable user-specified behavior.

The creation of tunnels has two aspects. The first one is the creation and configuration of the virtual interfaces, which is done through PlanetLab's *vsys* interface. We extended the existing interface to support creation of GRE links, which was previously only possible through Trellis [8], with all the scalability issues mentioned in Section 2. We also addressed potential conflicts related to concurrent use of GRE tunnels by many slivers, a problem present in Trellis, by marking all GRE packets between any two endpoints as belonging to the slice, and in this way allowing the kernel to de-multiplex traffic belonging to each slice both efficiently and securely, and allowing widespread adoption of GRE tunneling without interference.

We enhanced point-to-point links configuration in PlanetLab by allowing passing the remote endpoint to the kernel when appropriate, and automatically setting point-to-point routes, which is essential in the correct operation of many applications. The system as a whole, though, still needs network-level routes, so a mechanism for routing table manipulation is still necessary and will be discussed in 4.3.

The second aspect of tunnel creation, is the actual forwarding of packets. NEPI deploys a special purpose application for this, *tun_connect*. It creates UDP, TCP or GRE tunnels between two virtual interfaces, a basic building block of more complex topologies. GRE tunnels require no user-mode packet processing, so they perform best. But, when GRE links are inappropriate, such as when custom queues or stream filters are applied, user-space packet forwarding is performed. NEPI only performs packet encapsulation, routing and packet forwarding is done in the kernel, avoiding any kind of heavyweight packet processing that was a limiting factor in previous approaches.

4.3 Routing

Connecting network segments requires the presence of routing-capable nodes. That is, nodes that can forward packets to the intended destination IP, for which they need knowledge of the network's topology. There are two main ways to do this: static and dynamic routing.

In static routing, routes are predefined, either manually or automatically, but they do not change. This is the most common case, where routing tables are populated with fixed rules.

In dynamic routing, routing tables are present just as in static routing, but routers communicate with router-specific protocols to dynamically maintain optimum paths. There are many algorithms to do this, so the routing algorithms usually work in user space and are highly customizable, sending routing table updates to the kernel which does the actual forwarding. This mechanism is a compromise between flexibility and performance, with highly customizable routing daemons in user space, and packet forwarding taking place in the kernel, to avoid unnecessary copying and thus achieving higher performance.

In order to support both static and dynamic routing, a mechanism to manipulate the kernel's forwarding information base (FIB) is required. To this end we developed a new PlanetLab-specific interface, the *vroute vsys* script, that implements the standard IP routing manipulation system calls in a straightforward fashion while enforcing certain rules aimed at maintaining proper separation between slivers. *vroute* adds entries to the main routing table after validating that the routes will not interfere with other slivers. This is done by checking the private network address space assigned to the slice through the *vsys_vnet* tag. This tag can only be assigned by administrators, who are responsible of making sure that the assigned segments are non-overlapping.

Prior to our work, PlanetLab only provided the *sliceip* interface to create routing tables for slivers, which only supports 150 concurrent slivers per node, an insufficient amount for the average number of slivers in a node (see Section 2). By using both, the *sliceip* per-sliver routing, and the new *vroute* per-node routing methods, we solve the scalability problem and afford every sliver necessary control over the routing tables.

Using the main routing table has the benefit of being limited by the number of rules per node, rather than the number of slivers in a node. In fact, NEPI will pick the *vroute* method if it does not need many rules, and *sliceip* if the table is big enough to warrant a separate table, or the routes do not belong to the slice's private address space. With this we hope to make the system scale better to high number of users sharing resources on the same nodes.

In combination with *per-sliver routing* [20], our solution creates a scalable and secure way of configuring IP-routing at the kernel-level, that can be widely adopted without disrupting PlanetLab. This means userland routing daemons like *olsrd* are easily adaptable.

In this way, bandwidths up to 300Mbps have been achieved, enabling high-bandwidth, low-overhead, and highly-customized overlays, which were previously only possible in PL-VINI (using the handful PL-VINI-enabled nodes).

4.4 Traffic Shaping

In order to support overlay customization, including new queuing policies, userland packet forwarding and routing, like Xorp and Click, and even new tunnelling protocols like OverQoS[1], we introduced several hooks into the framework in or-

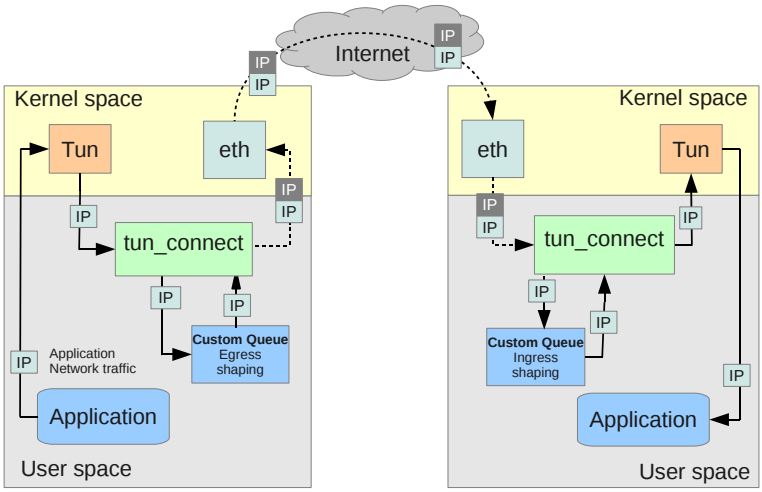

Fig. 3. Customizing overlay packet handling. Interprocess communication is used to send all packets through a user process. Custom queuing, filtering and aggregation algorithms can in this way be applied to the packet stream, both before encapsulation and transmission, and after reception and de-encapsulation. When this is used in combination with the built-in rate limiting options, arbitrary loss models, among other things, can be applied to the overlay.

der to let experiments process network packets without having to implement all the low-level details.

Stream filters can be arbitrary applications that process the packets flowing through the overlays. They can be used to implement custom queues, packet filters or transformations, tunnelling protocols. These modules can be provided to the *tun_connect* userland packet forwarder, to route all packets through user-specified code. The mechanism only works when userland forwarding is taking place, so it cannot be used with GRE tunnels.

The mechanism was designed so that even previous code not designed to work with NEPI could be used. It takes either Python or C code, and provides several ways in which experiments can process packets. Packets can be rejected by implementing a filtering predicate, or a customized queuing class can be provided that will be used instead of the primitive default FIFO. For more complex processing, a connection to an external process that filters packets can be made, covering most customization scenarios.

External filters can include piping all packets through a shell command, or forwarding them to separate daemons, as would be the case if we wanted to use Click. This is accomplished by writing a small module that returns two file descriptors (i.e. a socketpair) through which all packets are piped. Thus, no matter how a researcher decides to implement the prototype, the same code could be reused for testing within PlanetLab. All this provides a level of *flexibility* no other framework does.

The *tun_connect* module also performs user-land queuing. PlanetLab does not allow slivers to control virtual interface's queuing parameters, so queuing can be accomplished in user-land by specifying custom queuing sizes or classes. This allows researchers to very easily experiment with new queuing disciplines, as we will demonstrate in 5.1.

Implementing OverQoS in PlanetLab would be trivial with this framework, once the prototype has been written, significantly lowering the cost of experimentation. The only foreseeable drawback of this technique is performance, as the usage of stream filters incurs significant performance overhead.

5 Evaluation

In order to evaluate the solution in terms of ease of use, effectiveness and practicality, we present an experiment case conducted on PlanetLab using NEPI. We evaluate the practicality of our framework for real research cases by reproducing an experiment from a published paper: POPI [19], a tool for packet forwarding priority inference. It was originally validated in PlanetLab in a costly and laborious way, by probing routes between nodes and then manually requesting information from intermediate providers by contacting their administrators.

We re-validated POPI using NEPI in a controlled PlanetLab overlay, a possibility that was not available at the time the paper by Guohan et al[19] was published: by controlling routing behavior and comparing known overlay characteristics against POPI's inferred ones, we managed to achieve verifiable results, in an automated and effortless fashion.

5.1 POPI

POPI[19], which stands for Packet fOrwarding Priority Inference, is a tool that attempts to infer packet priorities in the intervening routers between two endpoints. In the paper by Guohan Lu et al, the tool is evaluated by simulation as a first step, and in PlanetLab as a second step. During the PlanetLab run, however, researchers had to ask ISPs about their routing policies, because they could not otherwise verify that the priorities reported by the tool corresponded to actual prioritization policies. Even then, their success was limited, because not every provider answered, and because the information so gathered was very rough.

We re-evaluated POPI, with the intention to evaluate NEPI's adequacy in creating controlled routing overlays for protocols and application validation. NEPI provides here the ability to create an overlay spanning lossy and congested links, while at the same time granting controllable packet prioritization at selected routing points by the use of customized queuing.

Designing the experiment was straightforward. NEPI already provided a reference queue class with TOS support [1], out of which a classifier queue based on

[1] NEPI provides a base queue implementation, which can be attached to network devices in PlanetLab, and defines queuing policies by inspecting the Type Of Service (TOS) field in the IPv4 header.

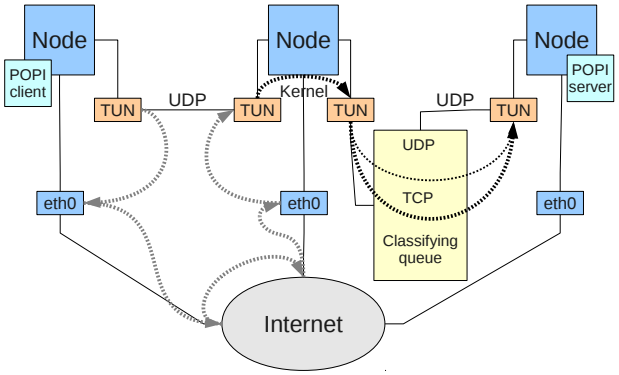

Fig. 4. POPI experiment concept. Three PlanetLab nodes connected physically by the Internet, on top of which a routing overlay provides a controllable environment in which to test POPI.

IP protocol was rather trivial to derive. We exploited our ability to control routing to introduce a mediating node between two arbitrary PlanetLab nodes. All traffic between the endpoints is routed through this mediating node, which applies a controlled class of queuing, and which should result in the application of recognizable statistical bias to the packet stream. Figure 4 shows the experiment design, which includes deployment of POPI, unmodified, in PlanetLab.

We ran the experiment with numerous queue configurations, with the help of a small script that leverages NEPI's programmatic API. The experiment was run in PlanetLab Europe (PLE) as well as a private PlanetLab-like cluster, allowing us to experiment with different environments with little effort. The experiment took 179 hours to run, but required no supervision. Of 325 runs, only 11 failed to be deployed, because of connectivity issues (the controller was operating from a domestic network), but were automatically retried or flagged as bad runs so they could be skipped when analyzing results.

Since any PlanetLab node would be able to run the experiment, we could have cut experiment run time significantly by running several configurations in parallel, using more nodes instead, and illustrating the convenience of not being limited to using only PL-VINI nodes for routing, as RiaS is. However, this would have undermined our ability to compare against our small, private cluster.

Table 1 shows execution details. We induced multiple connectivity glitches by connecting and disconnecting from the network, and joining and leaving VPNs to stress NEPI's failure recovery capabilities. Within the remaining runs, 32 failed because of problems with POPI's tool itself and produced no results. A vastly lower failure rate can be observed in our dedicated cluster, evidencing that the strain on PLE nodes does have an effect on experiment success. Our experiment was conducted at a particularly busy time of the year, yet failure rates are acceptable, due to NEPI's automated recovery procedures.

POPI could successfully infer queue prioritization in clear channels as long as the different classes had different rate limits, but in instances where protocols

Table 1. POPI experiment runs. Good runs are those where NEPI returned a successful code. Bad runs when it did otherwise. Failed runs are good runs that produced no output, in every case, because POPI could not establish the control connection due to connectivity glitches.

runs/sets	Good	Bad	Fail
PlanetLab Europe	142/30	8/0	16/0
Dedicated cluster	172/35	3/0	16/1

Table 2. POPI results, based on PLE runs only. Columns specify the bandwidth at the node with the classifier queue, while rows specify the classifier queue configuration. Marks show sets that contain cases (1) where the correct classification was detected (2) where there was underpartitioning (3) where there was overpartitioning. Marks are ordered according to predominance within the set. *"4x TCP"* represents a queue with 4 times as much bandwidth allocated to TCP traffic than other kinds.

config \ k	32	64	128	256	384
4x TCP	(3,2)	(1,2)	(1,2)	(3,1)	(1,3)
4x ICMP	(1,2)	(1)	n/a	n/a	(1)
4x UDP	(1,3)	(1)	(1)	(1,3)	(1)
4x TCP 16x I	(1)	(1,2)	(1)	(1,3)	(1)
4x U+T 16x I	(2,3)	(1)	(3)	(1)	(1)
4x T+I	(1)	(1)	n/a	(1,3)	(1)
indep. U,T,I	(2)	(2)	n/a	(1)	n/a

were assigned separate classes with equal bandwidth (fair share queuing), POPI could not tell the difference from a plain FIFO queue. There was a certain amount of overpartitioning (cases where POPI inferred more classes than in reality) and underpartioning (cases where POPI failed to infer a class), as expected, although somewhat more frequent than expected according to the original research paper. Table 2 summarizes our results.

In our experiment, 256k and 384k cases should not result in any classification, because other nodes were uniformly limited to 256k. Any partitioning there is considered overpartitioning, except when it detects the exact queue configuration. It is interesting to note that TCP priority detection is consistently less precise than with other protocols, as shown by the results of *"4x TCP"* and *"4x U+T 16x I"* at 128k. In the former, there was a tendency to overpartitioning, while in the latter TCP was consistently detected as having less priority than UDP. This, after checking the resulting packet captures, could most probably be due to reactive traffic generated by target nodes responding to POPI's synthetic TCP packets. This can interfere with the measurements, by exerting more pressure than expected on the bottleneck queue. This effect is even more prominent in our private cluster, where POPI is generally more precise due to the absence of background traffic. This only highlights the bias experienced in TCP measures, that is less evident when running in PLE.

All the required sources and scripts to reproduce the experiment have been made available on-line. With our tools, researchers can easily (assuming they do have access to PlanetLab) reproduce the experiment, and even build other experiments on top of it. NEPI's experiment description XML contains not only topology information, but also deployment instructions: where to get POPI sources, how to build it, how to patch it if it were required. NEPI's execution XMLs contain valuable details about the resources used to run the experiment. All this results in very strong reproducibility guarantees.

More details, and instructions on how to reproduce this experiment, can be found at: `http://www.nepihome.org/wiki/nepi/popiExperiment`.

6 Conclusions

In this paper we have presented a framework that increases control over experiments conducted using PlanetLab, allowing researchers to go beyond what previously available tools permitted.

Our solution is based on building flexible routing overlays on top of PlanetLab. We provide the ability to automate deployment of whole overlay experiment scenarios, and improve the flexibility and scalability of the tunnelling and routing techniques compared to previous approaches used in PlanetLab. Additionally, our solution supports custom traffic shaping and different traffic encapsulation and transmission methods, enabling easy customization of the overlays.

As part of this work, we extended PlanetLab's *vsys* interface to include scalable and secure mechanisms for routing table manipulation, and creation of GRE links. With these tools, we have shown how to implement low-overhead yet highly customizable routing overlays.

In our use case "POPI", we demonstrated the relevance of our contribution by showing how it enables the researcher to gather experimental information that was previously unavailable to him or her. By choosing to reproduce a previously published experiment case, we proved that our solution is relevant to real research cases, while providing additional value.

A comprehensive technical evaluation of our solution is needed to further validate the extent and limitations of our work. In future work, we will focus on evaluating metrics related to scalability, resource usage, and performance. To this end, we will consider node specific metrics, such as per node maximum bandwidth and resource consumption, as well as global performance and scalability metrics, such as maximum number of concurrent overlays and maximum number of nodes per overlay.

References

1. Subramanian, L., Stoica, I., Balakrishnan, H., Katz, R.: OverQoS: An Overlay based Architecture for Enhancing Internet QoS. In: NSDI 2004 (2004)
2. Jannotti, J., Gifford, D.K., Johnson, K.L., Kaashoek, F.M., O'toole, J.W., Frans, M., James, K.: Overcast: Reliable Multicasting with an Overlay Network. In: OSDI 2009 (2009)

3. Andersen, D., Balakrishnan, H., Kaashoek, F., Morris, R.: Resilient overlay networks. In: SOSP 2001 (2001)
4. Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., Bowman, M.: PlanetLab: an overlay testbed for broad-coverage services. In: SIGCOMM 2003 (2003)
5. Bavier, A., Feamster, N., Huang, M., Peterson, L., Rexford, J.: In VINI veritas: realistic and controlled network experimentation. In: SIGCOMM 2006 (2006)
6. Lischka, J., Karl, H.: RiaS: overlay topology creation on a PlanetLab infrastructure. In: SIGCOMM VISA 2010 (2010)
7. Mark, H.: VNET: PlanetLab Virtualized Network Access. PlanetLab Consortium (2005)
8. Bhatia, S., Motiwala, M., Muhlbauer, W., Mundada, Y., Valancius, V., Bavier, A., Feamster, N., Peterson, L., Rexford, J.: Trellis: a platform for building flexible, fast virtual networks on commodity hardware. In: CoNEXT 2008 (2008)
9. Leonini, L., Rivière, E., Felber, P.: SPLAY: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze. In: NSDI 2009 (2009)
10. Muir, S., Peterson, L., Fiuczynski, M., Cappos, J., Hartman, J.: Privileged operations in the PlanetLab virtualised environment. SIGOPS Oper. Syst. Rev. 40, 75–88 (2006)
11. Bhatia, S., Di Stasi, G., Haddow, T., Bavier, A., Muir, S., Peterson, L.: Vsys: A Programmable sudo. In: USENIX ATC (2011)
12. Lacage, M., Ferrari, M., Hansen, M., Turletti, T., Dabbous, W.: NEPI: using independent simulators, emulators, and testbeds for easy experimentation. SIGOPS Oper. Syst. Rev. 43, 60–65 (2010)
13. Quereilhac, A., Freire, C., Lavage, M., Turletti, T., Dabbous, W.: NEPI: An Integration Framework for Network Experimentation. In: SoftCom 2011 (2011)
14. Albrecht, J., Tuttle, C., Snoeren, A.C., Vahdat, A.: PlanetLab application management using plush. SIGOPS Oper. Syst. Rev. 40, 33–40 (2006)
15. CoMon, `http://comon.cs.princeton.edu/`
16. Linux-VServer, `http://linux-vserver.org`
17. User-mode Linux, `http://user-mode-linux.sourceforge.net/`
18. Linux network namespaces, `http://lxc.sourceforge.net/index.php/about/kernel-namespaces/network/`
19. Lu, G., Chen, Y., Birrer, S., Bustamante, F.E., Li, X.: POPI: a user-level tool for inferring router packet forwarding priority. IEEE/ACM Trans. Netw. 18, 1–14 (2010)
20. Botta, A., Canonico, R., Di Stasi, G., Pescape, A., Ventre, G.: Providing UMTS connectivity to PlanetLab nodes. In: CoNEXT 2008 (2008)
21. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F.: The click modular router. ACM Trans. Comput. Syst. 18, 263–297 (2000)
22. Floyd, S., Paxson, V.: Difficulties in simulating the internet. J. IEEE/ACM Transactions on Networking (TON) 9 (2001)