# An emulation tool for PlanetLab

Marta Carbone[a], Luigi Rizzo[a]

[a] *Dipartimento di Ingegneria dell'Informazione, Università di Pisa*
*Via Diotisalvi 2 - 56122 - Pisa, Italy*

**Abstract**

Network testbeds have become very popular to support research on network protocols and distributed applications. To reproduce network behaviour, testbeds range between two extremes: use a fully emulated network, or distribute nodes on the real Internet. The former approach yields very reproducible results but might be a poor representation of reality; the latter gives more realistic but less reproducible scenarios.

In this paper we present an emulation solution for the PlanetLab testbed, and provide a detailed description of its feature and performance. Our system gives researchers the advantages of emulation while not giving up the opportunity of running experiments in a large and heterogeneous testbed with realistic network conditions. The work is based on a Linux version of the Dummynet network emulator, extended with specific features to improve efficiency on PlanetLab, and with custom configuration mechanisms to simplify its use.

The system described in this paper, developed as part of the Onelab2 project, has been deployed on a large subset of PlanetLab nodes. The emulation code itself is also available for ordinary Linux systems.

*Keywords:* Internet, network testbeds, emulation, PlanetLab, performance evaluation

## 1. Introduction

In recent years there has seen a significant growth in the deployment of testbeds to support research on network protocols and distributed applications. The primary motivation for most of these projects is to make available to researchers a system that, for its size and features, would not be affordable for individuals or even single institutions. Depending on the case, testbeds are built as a result of a community effort, where each participant contributes computing and networking resources; or thanks to the support of funding agencies, which sponsor strategic initiatives such as GENI [1] and FIRE [2].

Testbeds are generally made of a large number of computing nodes, managed by a central authority, and equipped with various storage and communication devices. Depending on the circumstance, the interconnection network (and the testbed itself) can be concentrated in a single location, or distributed across a large geographical area.

The target of each testbed varies. Some, such as Emulab [3], are focused on providing a very reproducible environment, and often make heavy use of virtualization and emulation techniques to present configurable and predictable node capacity or network resources to researchers.

Other testbeds address specific research topics, such as the study of wireless networks (ORBIT [4]), or routing protocols (VINI [5]). In these cases, the testbed includes components to address the particular problem domain.

Finally, testbeds such as PlanetLab [6] try to provide a realistic snapshot of the real Internet. The heterogeneity of nodes and network links that are part of a PlanetLab instance is a feature of the platform: it helps exposing applications to the same conditions that they would experience when deployed, but carries with it some lack of control on the reproducibility of experiments, because network conditions between nodes are typically unknown and variable over time.

This paper addresses the latter problem by extending PlanetLab with an emulation system that complements the features of the platform, and permits researchers to switch easily between two extremes: fully reproducible or completely realistic but uncontrolled network conditions.

The main contribution of this paper is a system

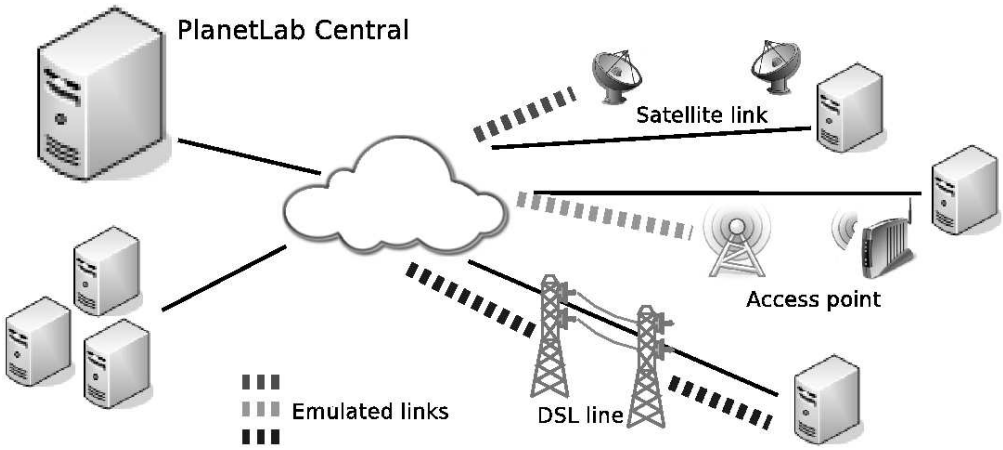



Figure 1: Applications of our PlanetLab extension. Different types of links (dashed) can be emulated on top of existing, physical links (solid).

that lets PlanetLab users configure, independently of each other, multiple emulated links for their experiments (Figure 1). Our system is made of several components, specifically tailored to the platform's needs: an emulation engine, implemented with a Linux port of the Dummynet [7] emulator; the addition of specific packet filtering and demultiplexing features (Section 4.2), to support concurrent users in a robust and efficient way; and a carefully designed user interface (Section 4) that permits an easy use of the emulation features.

The rest of the paper is structured as follows. Motivations for this work are presented in Section 2. Section 3 describes the overall architecture of our system, and provides a description of the components involved in our work: the PlanetLab testbed in Section 3.1, and the Dummynet emulator in Section 3.3. Section 3.4 documents how we ported Dummynet to Linux, while Section 4 shows how emulation is made available to PlanetLab users. Experimental results, including performance data, are presented in Section 5. Finally, Section 6 gives an overview of related work.

## 2. Motivations

PlanetLab [6] is a network testbed made by roughly a thousand of nodes distributed throughout the world and contributed by participating organizations. This testbed offers users and researchers a realistic snapshot of the Internet, where they can deploy new protocols, run experiments and measure network performance. PlanetLab is widely used and interesting due to its size and heterogeneity of network links and node hardware. On the weak side, the lack of any control on the conditions of the network makes it hard to obtain reproducible experiments, and even harder to run tests under controlled conditions. Reproducibility is a feature that we consider highly desirable, even more so if we can achieve it without giving up the existing features of PlanetLab: this constitutes the main motivation for the work presented here.

A common approach to achieve reproducible network behaviour is the use of emulation. As an example, in Emulab [3] nodes are colocated and connected by configurable switches, with FreeBSD machines interposed on the links and running the Dummynet [7] emulation software to provide the desired network features.

A centralized emulator cannot be used in PlanetLab because there are no controlled devices on the path between nodes and the rest of the network. However, we can achieve a similar result implementing emulation directly within the nodes. Traffic will traverse both emulated and real links, and will be subject to the limitations imposed by the two. Depending on their features, we can try to make one of the two components dominating over the other, and achieve a reasonable amount of control over the features of the communication network. This approach is made easier by the fact that clusters of PlanetLab machines reside within the same lab or data center, and PlanetLab nodes

are generally well connected to the rest of the Internet. As a consequence, in many cases it it possible to neglect the unpredictable component introduced by the physical links.

## 3. Architecture

Important design decisions for our emulation system are how to implement the emulation engine; how to integrate it within PlanetLab nodes; how to provide safe operation and insulation among users; which emulation features to expose to users; how to design a user-friendly interface, to avoid distracting researchers from their main goals; and finally, what is the impact of emulation on the nodes' performance. The rest of the paper will address these questions.

Figure 2 gives a snapshot of the architecture of our emulation system. Users issue commands through a simplified user interface (Section 4) to configure the desired features of the emulated links. Using the Vsys service (Section 3.1.2), commands are passed to the "root context" of the PlanetLab node, which can issue actual requests to configure the emulation engine (Section 3.3). The latter includes specific features (Section 4.3) to improve the performance and robustness of operation in a shared and heavily loaded system such as a PlanetLab node.

In the next Sections we describe the various components involved in our architecture, and show how we use them to achieve our goals.

### *3.1. PlanetLab*

PlanetLab is made of two types of components: a central controller, called PLC (PlanetLab Central), and several computing *nodes*, where users can run their experiments.

The PLC is the core of the system. It runs the testbed management software and acts as a server for nodes and users. Nodes willing to be part of the testbed must download from the PLC, and install on their disks, a custom version of Linux together with a set of management programs.

PlanetLab *users* also register with the PLC to gain access to the system. Once registered, they are allowed to create one or more *slices*, the administrative entities used to account for resource usage. A slice's instantiation on a node is called *sliver*, and it is essentially a user account running in a protected environment on the node.

#### *3.1.1. Node and sliver management*

Users log into the nodes and run their experiments in a virtualized environment which provides resources isolation between the slivers. This is implemented by the Linux-Vservers [8] system, providing a private filesystem namespace to each sliver, while still allowing all slices to access the full set of devices available on the node. Each sliver runs within a dedicated *vserver context* with limited root permissions, meaning that the sliver can only execute a subset of the system calls. Operations that require real root access (i.e. must run in the so-called *root context*), are controlled by the *vsys* service described next.

#### *3.1.2. The vsys service*

Users are king (root) in their Vserver, but their privileges on operating on the root context are limited and strictly controlled using the vsys service. The vsys service is the mechanism used by slivers to perform certain privileged operations that may affect the whole node (as an example, configuring interfaces or packet filters). The service works by creating one or more file descriptors accessible to the sliver (see Figure 2, left). These descriptors communicate with backend programs that run in the root context and are, as a consequence, able to invoke any system call without restrictions. Vsys backends are installed by the system administrator, who can also specify which backends are available to each sliver. The identity of the invoking sliver is available to the backend at runtime, so that specific access policies can be implemented.

In our system, we use the vsys service to authenticate and pass to the root context all user requests for configuring emulation.

### *3.2. Choice of emulation engine*

PlanetLab nodes run a custom version of Linux, and there are several existing emulation packages already available for that operating system, including NISTnet [9], *tc* [10], and netpath [11] (the latter is based on Click [12]).

We dismissed NISTnet because it is not available as a standard component in PlanetLab, and is not actively developed. *tc*, which is a traffic shaper and link scheduler, was not a suitable choice for at least two reasons. First, it is not an exact match for our requirements, as it requires an external package, netem [13], to model features such as propagation delays and reordering. Second, and most important, *tc* is already used within PlanetLab nodes to

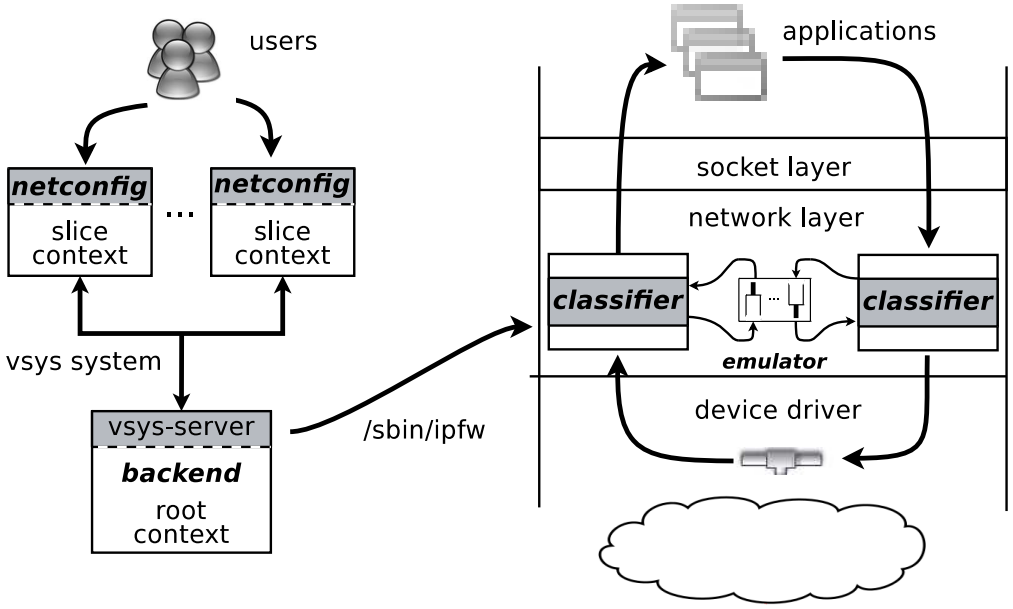



Figure 2: On the left, the interaction between users, vsys frontend and backend (Section 3.1.2). On the right, the flow of packets through network stack, packet classifier and pipes (Section 3.3.3).

enforce traffic limitations, so its use for emulation would interfere with the existing configuration, and require a lot of care to ensure a safe coexistence.

netpath is interesting in terms of performance when used as a standalone emulator, because it uses its own device driver hooks and packet processing stack. However, much of netpath's performance comes from an aggressive use of polling and busy wait loops, which are a bad fit with PlanetLab nodes, already heavily loaded with user programs.

Eventually we decided to select Dummynet as our emulation engine. We have a significant experience with it, and have already used it as an external emulation solution in PlanetLab. Also, Dummynet is used on Emulab, which means that researcher may be already familiar with it. Dummynet was not natively available on Linux when we started this work but the port to the new operating system required a manageable amount of effort, and was a useful contribution in its own.

### *3.3. Dummynet*

Dummynet [7] is a network emulator developed under FreeBSD several years ago [14], later imported into other BSD-derived operating systems, including Mac OS X, and currently also available on Linux, OpenWrt and Windows.

Dummynet is a component of the operating system that can intercept network traffic and manipulate it, emulating the behaviour of one or more network links with programmable features. It is made of three parts: the emulator itself, *dummynet*; a packet classifier, *ipfw*; and a user interface, `/sbin/ipfw`. The first two parts run in the kernel of the operating system, and communicate with the user interface through a control socket. A full description of Dummynet is in [7]; the next Section reports only the details (including newly introduced features) that are relevant for this work.

#### *3.3.1. The emulator*

*dummynet* (the emulator) can create multiple instances of an object called *pipe*, which in its basic version shown in Figure 3 models a network links with programmable bandwidth, delay and queue size.

Other pipe configuration options exist to specify different queue management policies (e.g. RED), to model some MAC layer effects such as variable transmission times and link level overheads, and also to simulate very simple packet drop patterns.

More advanced features allow connecting multiple queues to a packet scheduler running one of several scheduling algorithms, and then sending packets through a link with configurable features.

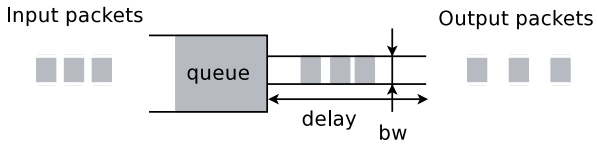



Figure 3: The basic components of a Dummynet pipe.

In this project we use pairs of pipes to emulate the features of the two directions of a communication link.

*3.3.2. Extended emulation features*

Part of the focus of the Onelab project, in which this work has been developed, was on wireless networks. As a consequence, we have extended the basic pipe model of Figure 3 to support better emulation of wireless and other channels with peculiar MAC protocols, variable transmission rates or channel errors. Rather than dealing with the complexity of modeling all the details of specific MAC protocols, we introduced two features: *delay profiles* and *varying links*.

*Delay profiles* support the definition of additional MAC overheads (such as contention, framing, retransmissions) through empirical profiles: the transmission time is extended by a random time, computed according to the distribution provided by the user (e.g., Figure 4). This way we can achieve a better match of the transmission times with those on real wireless links.

*Varying links* serve to model the variability of wireless channels (including loss rates and bandwidth) over time due to e.g. external interference or mobility. The pipe can be in one of many states,

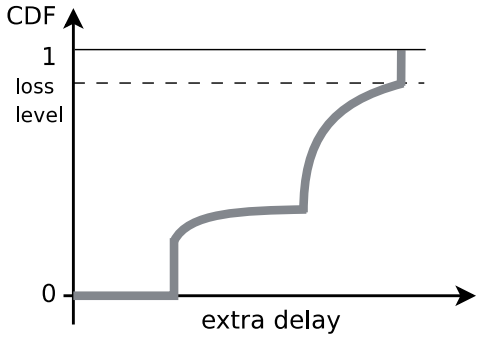



Figure 4: A sample *delay profile* describing the distribution of MAC overheads (Section 3.3.2).

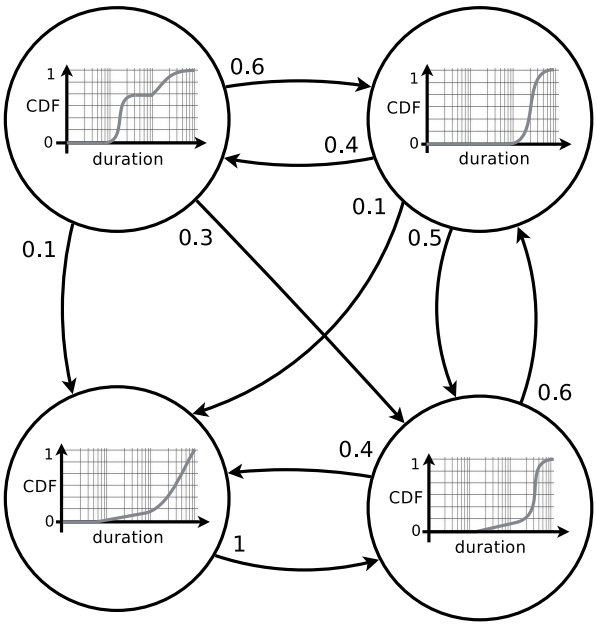



Figure 5: An example of the information used to implement *varying links* (Section 3.3.2). The link remains in each state for intervals of time with the given distribution, and then moves to a new state with the probability specified on the arcs.

each with its own set of parameters. Arcs connect states into a graph which specifies possible transitions. For each state we can specify, once again using empirical distribution curves, the amount of time spent in the state before moving to a new one, and the probability associated with each of the outgoing arcs (Figure 5). The system will then randomly switch between states in a way that yields the same distribution as programmed by the user.

*3.3.3. The packet classifier*

Dummynet works in close cooperation with a programmable packet classifier, *ipfw*, that intercepts packets in various points of the protocol stack, and decides of their fate. The packet's flow through the network stack, packet classifier and pipes is represented on Figure 2, right.

*ipfw* is programmed by writing a set of numbered *rules*, each containing zero or more *options* used to match packets, and one *action* specifying what to do with matching packets. Matching options include addresses, ports, protocols, protocol flags and various packet's metadata including the virtual server that the packet is associated to. to provide insulation between the different users.

Traffic selection is performed by testing a packet against each of the rules, in numeric order, and performing the action associated to the first matching rule. For our purposes, the actions of interest are sending the packet to a pipe, which will in turn delaying or dropping the packet as appropriate, emulating the behaviour of the attached link. After the emulation, non-dropped packets are sent back into the network stack for their regular processing.

In this project we use classifier rules to dispatch traffic to the appropriate pipes, according to the configuration requests issued by users.

### *3.4. Porting Dummynet to Linux*

The port of Dummynet to Linux has applications even outside this specific project, because it makes the emulator available on a much wider set of systems, including embedded devices running OpenWrt [15], which is more and more used in various research prototypes as well as actual deployments.

The porting of the user interface, `/sbin/ipfw`, was trivial and just required to provide replacements or wrappers for library functions that differ between FreeBSD and Linux. The adaptation of the kernel subsystem was instead a lot more challenging, due to the lack of cross-platform standards in terms of programming interfaces (APIs), headers, kernel services, and even naming conventions.

Having performed similar work in the past, we found that a very effective strategy in these cases is to keep the original source code unmodified as much as possible (but within reason). This approach has the double benefit of pointing out platform-specific assumptions (with the opportunity to fix them in the mainstream code), and making it easier to keep the port up to date over time.

The most time-consuming parts of this specific work were related to the design of the adaptation infrastructure, and specifically i) identify the best location to add missing definitions and headers, ii) decide where to apply the "within reason" principle and which changes to the original source were acceptable, and iii) identify a good replacement for the kernel subsystems used by Dummynet.

#### *3.4.1. Hooking into the Linux network stack*

A first issue was to identify how to hook the classifier and the emulator into the network stack. Our two requirements are to intercept traffic in two points (one upstream, one downstream) and to reinject packets back into the stack after some delay.

Many operating systems support the insertion of generic *packet filter* functions on the packets' path. This mechanism is called *pfil* on FreeBSD, *netfilter* on Linux, and *miniport drivers* on Windows. All these subsystems allow us to manipulate packets without modification to the rest of the Operating System's kernel.

For our purposes we used the Linux *netfilter* system, which can divert packets from the normal processing path and put them into a queue. From there, packets are passed to a user-specified *queue handler* function, which can delay or manipulate packets at will, and finally call `nf_reinject()` to reinject them into the stack and possibly drop them if needed.

In our system, the queue handler calls the classifier, and depending on the outcome either passes the packet back to the network stack, or calls the emulator, which in turn calls `nf_reinject()` after a suitable delay. Figure 6 shows how the mechanism works in case of an incoming packet.

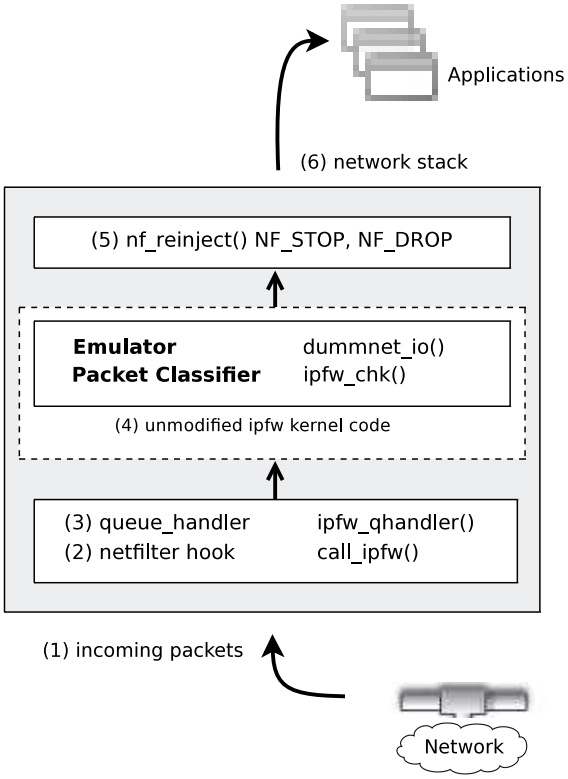



Figure 6: The interaction between netfilter hooks and the classifier (ipfw_chk() and emulator (dummynet_io()) code.

*3.4.2. In-kernel packet representation*

Across the various operating systems, the representation of network packets within the kernel varies in the details but not much in the approach. Typically, the data portion is stored in one or more linked buffers, and an external descriptor is used to store metadata. such as packet length, a pointer to the actual data, direction, related interfaces, flags.

In FreeBSD and other BSD-derived systems, metadata are stored in a structure called `mbuf`. In Linux, there is a similar arrangement except that the container for metadata is called `sk_buff`.

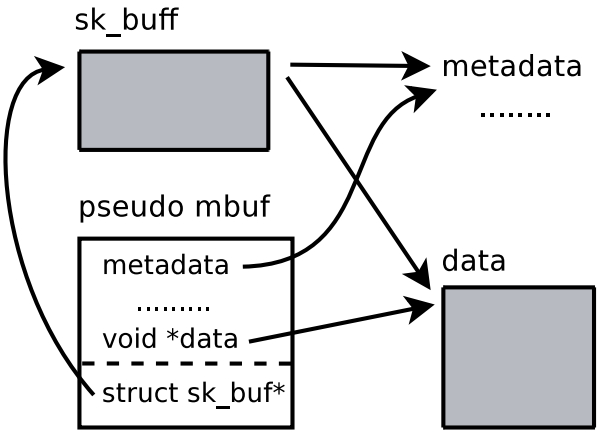



Figure 7: The mapping between `sk_buff`s and `mbuf`s.

In our port, whenever we receive an `sk_buff` representing a packet to process, we create a stripped-down `mbuf` structure, initialized with relevant fields fetched from the `sk_buff`. Figure 7 represents the linkage between the original `sk_buff` and the new `mbuf` representation. This way, the code to access the packet data or metadata can remain unmodified and simply refer to the usual `mbuf` fields. On return, the pseudo `mbuf` descriptor is destroyed, and the packet is reinjected into the network stack completely unmodified.

## 4. Usage model and user interface

One of the main features of Dummynet is the ease of use, and we tried to preserve this simplicity also in the integration into PlanetLab.

The Dummynet's user interface, `/sbin/ipfw`, is too low level for most PlanetLab users due to the huge number of options available. Furthermore, we could not expose it to individual users because of the risk of unwanted misconfiguration affecting other slivers. We then decided to offer users a very simple usage model based on three types of configurations: *server*, *client* and *service*.

Users configure one or more links, and define the traffic affected, with individual commands such as those in Figure 8, which emulate a multi-homed client where selected HTTP/HTTPS traffic to a /16 subnet goes through an emulated 3G link, whereas other traffic for a /24 subnet goes through a link with ADSL-like features. Port lists and addresses/masks can be used to pass specific traffic through each of the emulated links. Parameters of the link (bandwidth, delays, loss rates, delay profiles or variable channel features as in Section 3.3.2) can be specified independently for the two directions of the communication, to cover the case of asymmetric links.

Replacing the *client* keyword with the desired configuration type will let us deploy one of the desired settings:

**client** emulates a node hosting clients that connect to external servers, whose ports and/or addresses are known. The classifier will intercept all traffic to/from those servers, and pass it through two emulated links with the specified parameters;

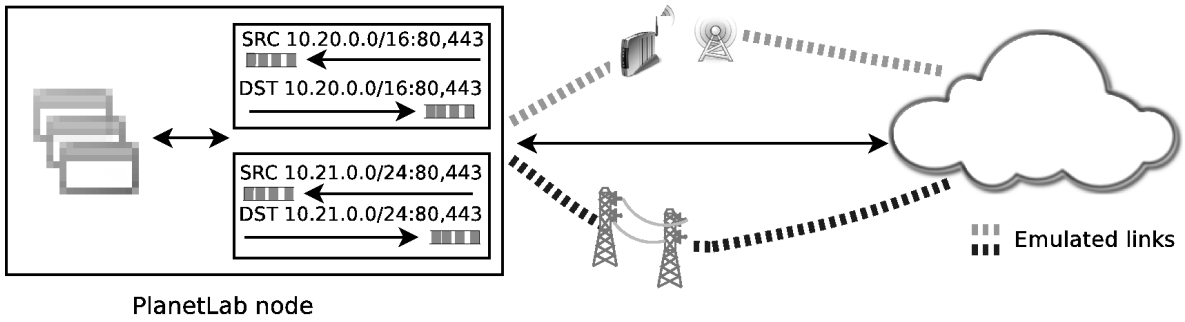
**server** is meant to emulate the case where the local node hosts a server on one or more well-known local ports. The user specifies the local ports, and possibly the addresses of remote clients/subnets if we want to differentiate the behaviour depending on whom is talking to the server;

**service** can be used when we have a distributed application, e.g. a P2P system, where nodes run both clients and servers on well known ports. In this case, the emulator will be configured to intercept traffic between parties of the same application – in practice, this represents a combination of a *client* and *server* configuration that share the same emulated links.

Note that a user can define several emulated at the same time, of course operating on different traffic.

*4.1. Under the hood*

The `netconfig` program does nothing but pass the request to the root context, together with the




```
netconfig client 80,443@10.20.0.0/16 IN bw 3.6Mbit/s delay 50ms OUT bw 1.2Mbit/s delay 80ms
netconfig client 80,443@10.21.0.0/24 IN bw 512kbit/s delay 8ms OUT bw 10Mbit/s delay 3ms
```

Figure 8: An example configuration to emulate a multihomed client. Two emulated links intercept traffic for two different subnets.

identity of the sliver issuing the request. The ports/addresses specified as arguments are used as a key to detect whether we are creating a new emulated link or modifying/deleting an existing configuration, and make the backend act accordingly. A request normally installs a couple of rules for the classifier to select the desired inbound and outbound traffic, and configures two pipes with the specified features, one for each direction of the emulated link. Figure 9 shows an example of a command and the configuration it generates.

In the translation, rule and pipe numbers are assigned by the backend. Most other parameters (bandwidth, delay, ports and addresses, other filtering options) come from the user's request. The `sliver X` options are inserted, as described in the next Section, to prevent interference between users' configurations.

### *4.2. Isolation between users*

We want to make sure that rules generated by one sliver cannot match traffic belonging to another sliver. To this purpose, we add a `sliver X` match option to all rules, where `X` is the sliver ID, derived automatically, and without chance of being overridden, from the identity of the sliver issuing the `netconfig` request. At run time, the packet classifier looks up the socket and sliver associated with each packet (either incoming and or outgoing), and the information is used to make sure that, irrespective of any other match option, rules will match only traffic for the sliver that requested this specific configuration.

### *4.3. Optimizing performance*

A naive implementation of the translation of requests into `/sbin/ipfw` rules would quickly lead to scalability problems. In fact, as we have seen, each emulated link implies the insertion of a couple of rules in the classifier's configuration. Rules are scanned sequentially (see Section 3.3.3), so even limiting the maximum number $M$ of emulated links that a sliver can define, the cost of scanning the ruleset for a system with $N$ slivers would grow as $O$(N*M). In PlanetLab, $N$ is already as large as a few hundreds, and this cost would be paid on each packet, which is clearly unacceptably high.

To reduce this complexity, we have structured the ruleset as shown in Figure 10: after a small number

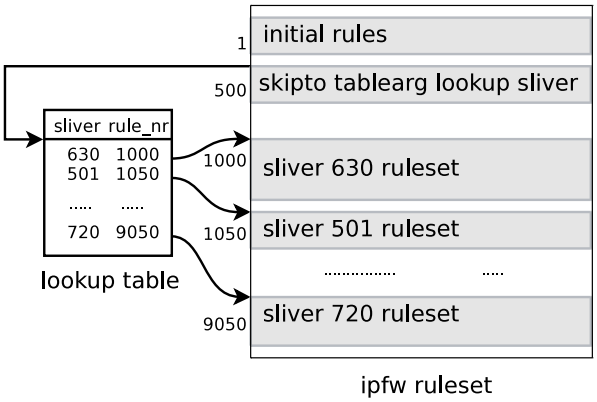



Figure 10: The structure of the ruleset used in the classifier, and the sliver table used to perform a fast dispatch of packets to the block of rules for each sliver.

```
netconfig config client 22,80@xyz IN bw 6Mbit/s OUT bw 256Kbit/s
ipfw pipe 10000 config bw 6Mbit/s
ipfw pipe 10001 config bw 256Kbit/s
ipfw add 1050 pipe 10000 in src-ip xyz src-port 22,80 sliver 50
ipfw add 1050 pipe 10001 out dst-ip xyz dst-port 22,80 sliver 50
```

Figure 9: A netconfig command and its translation in terms of classifier rules and pipes' configuration.

of rules used for housekeeping, we invoke a special classifier rule which jumps to a specific block of rules using the sliver number as the dispatching key. The cost of looking up the dispatch table, and jumping to a specific entry, is $O(\log \mathrm{N})$, followed by at most $O(\mathrm{M})$ steps for finding the right rule within the block. This makes the problem completely manageable because we can limit $M$ to a small value, and the logarithmic component never requires more than 16 steps, so it only causes a modest overhead.

The next Section presents detailed performance measurements to quantify the per packet cost in the worst case, and show the effectiveness of our approach.

## 5. Experimental results

Our main performance metric is the per-packet processing cost, because at least the classification, and possibly emulation as well, affects all packets flowing through a node. A second goal is to determine the accuracy with which emulation reproduces the configured parameters, and compare this accuracy with the variability of the parameters that exist in the node even before the introduction of the emulator. Finally, though less important, we need to determine the cost of configuring the emulator, an operation that involves running the vsys frontend and backend, and going through the vsys system itself.

The classification and emulation cost and accuracy depends almost entirely on the features of the emulation engine, for which a very detailed analysis is present in [7]. The following Sections will build on those results and deepen the analysis in the context of a PlanetLab node.

### *5.1. Emulator overhead*

In [7] we have shown that the per-packet processing costs are made of two components – classification and actual emulation. The classification cost grows linearly with the number of rules, whereas the emulation cost is essentially constant except for a small $O(\log \mathrm{L})$ component, where $L$ is the number of active pipes.

Measurements made in [7] on a low-end workstation reported approximately 400 ns to enter the classifier, 36 ns for each simple rule, 800-1300 ns for traversing a pipe, and an additional $\log(L) \cdot 100$ ns to account for the dependency on the number of active pipes. Absolute values change depending on the hardware, but the ratio between the components is approximately constant across platforms.

The PlanetLab version of the emulation engine uses a table lookup to quickly jump to the interesting block of rules (Section 4.3). The cost of this specific ruleset was not measured in [7], so we repeated the measurements here using the same approach, i.e., generating traffic from a local source and dropping it at various stages of the protocol stack (Figure 11). By taking the differences between the various measurements we can make an estimate of the classification and emulation costs. Specifically, the table lookup has a logarithmic cost in the number of entries (sliver IDs) so we make sure to trigger the worst case conditions by running some experiments with a table containing $2^{16}$ entries (sliver IDs are represented on 16 bits). Also, in previous measurements we determined that the worst case for emulation costs is a link with only delay and no bandwidth limit, so we use that configuration in our tests.

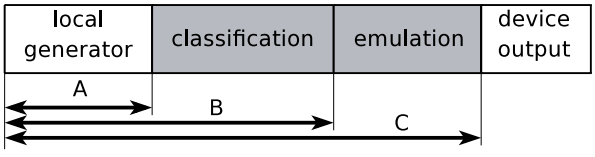



Figure 11: The configurations used to evaluate the per-packet processing overhead. We measure the PPS rate with a local generator and traffic dropped in different points of the protocol stack. Differences between the measurements give the cost of each processing block.

The results of our experiments are presented in Table 1. Each test case involves a 10-second run repeated 100 times. The per-packet processing time is measured by dividing the length of the experiment (10 seconds) by the number of packets generated. The hardware used for experiment is a desktop machine at the low end of the specifications for a PlanetLab node, so we can expect that existing nodes have the same or better performance.

| Case | avg/sd (ns) | 1 flow |
|---|---|---|
| $A$ | 1167 / 72.5 | Drop before classifier. |
| $B_1$ | 1525 / 51.2 | Drop in first rule. |
| $B_T$ | 1750 / 51.5 | Table with 1 entry. |
| $B_{FT}$ | 1911 / 60.8 | Table with $2^{16}$ entries. |
| $C$ | 3311 / 69.4 | $B_{TF}$ and worst-case pipe. |

Table 1: Results of the measurement of per-packet overheads with different configurations of the classifier and emulator.

Specifically, case $A$ represents the cost of packet generation alone. Case $B_1$ is the time consumed for packet generation plus classification with a ruleset containing a single rule. Case $B_T$ goes through an additional table lookup with a table containing a single entry. Case $B_{FT}$ does a lookup through a full table, and the final case, $C$ represents a lookup plus traversal of a worst-case pipe.

From the difference between $A$ and $C$ we see that in the worst case emulation consumes roughly 2150 ns with a full table and one active pipe. If we allow each user to define up to $M$ emulated links, we need to add approximately $M \cdot 70$ ns to this time, and another small $log(L)$ contribution comes from the presence of multiple, simultaneously active pipes.

Overall, we can safely claim that the presence of the emulator adds less than 4 $\mu$s to the per packet processing time. Note that packets not subject to emulation will be subject to a much lower overhead (less than 1 $\mu$s), as they are typically filtered out before or during the table lookup.

#### *5.1.1. Throughput*

To determine how this overhead impacts the performance of a PlanetLab node, we have studied the network load on PlanetLab nodes using the data reported by the CoTop monitor [16]. CoTop computes the average traffic of a node over a 1-minute interval. The highest values we saw were around 30 Mbit/s. Unfortunately we do not have data on the number of packets per seconds (PPS) measured on the nodes, but 30 Mbit/s correspond to a PPS load between 2500 and 58000 PPS, considering a packet size of 1500 and 64 bytes respectively.

Even taking the worst case, a 4 $\mu$s per-packet overhead corresponds to 25% of one CPU core, which is an acceptable value, considering that we are probably overestimating the PPS load on the node by almost one order of magnitude, and that the majority of PlanetLab nodes have much lower traffic.

### *5.2. Emulator accuracy*

An important aspect to be evaluated in an emulator is how much it is able to reproduce the timing computed by the model. Once again, [7] makes a detailed analysis of this aspect, pointing out three main sources of inaccuracy: timer resolution, competing network traffic, and operating system interference. We want to estimate the impact of these factors on PlanetLab nodes.

The timer resolution is 1 ms in the kernels used on PlanetLab nodes, and this adds an equivalent error on timings.

The presence of different emulated links sharing same output interface can cause additional delays when two or more packets become ready for transmission at the same time. In this case the delay is proportional to the number of competing flows and maximum packet length and inversely proportional to the link speed of the physical interface. In practice, we saw that a large number (70-80%) of PlanetLab nodes are equipped with 1 Gbit/s interfaces, but in most cases they are attached to a 100 Mbit/s switch. This implies an error of the order of 1.2 ms, comparable to the one introduced by timer resolution.

The largest source of inaccuracy, however, is the operating system load, and its impact can be easily measured with simple experiments such as measuring the distribution of ping response times from a nearby host (or any other interaction that causes a process to suspend and be woken up). In an ideal situation, we expect constant response times, but interference due to other OS activities (e.g. processes for the same or other slivers) may cause additional delays.

As an experiment we measured the distribution of ping response times between pairs of colocated PlanetLab nodes. Not surprisingly, the values have large variations, as shown in Figure 12 which plots the CDF of the measurement on a number of node

pairs. The median values range between 150 and 500 ns in most cases, but almost invariably we see a long tail, with the top 5-10% of the values going up to several milliseconds. In one case (rightmost curve in the figure, corresponding to a heavily loaded node), delay of tens of milliseconds and more have been measured.

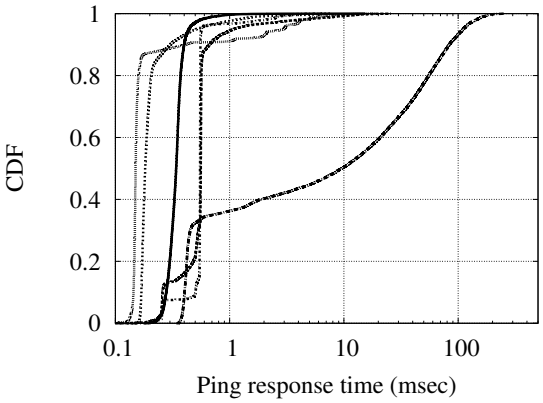



Figure 12: Distribution of ping response times between some pairs of colocated PlanetLab nodes.

From the above we can conclude that the emulation (in)accuracy is no larger than the timing uncertainties normally experienced by applications running on the nodes and due to competing processes and network activity.

### *5.3. Reconfiguration cost*

As a final part of the performance evaluation, we have measured the cost of adding, modifying or removing an emulated link. Though infrequent, these requests can be sent concurrently by all slivers, so we want to make sure that they don't cause an excessive overhead on the system. Reconfigurations, triggered by explicit user requests, uses the vsys system to communicate, and then makes a few calls to `/sbin/ipfw` to update pipes' and classifier configuration. Figure 13 shows the components involved.

Even with the current, non-optimized structure of the backend (the configuration database is stored in a text file and manipulated by a shell script) the entire backend runs in less than 30 ms, which is an acceptable value. It is easy to reduce this cost by almost one order of magnitude by moving to a compiled version of the backend.

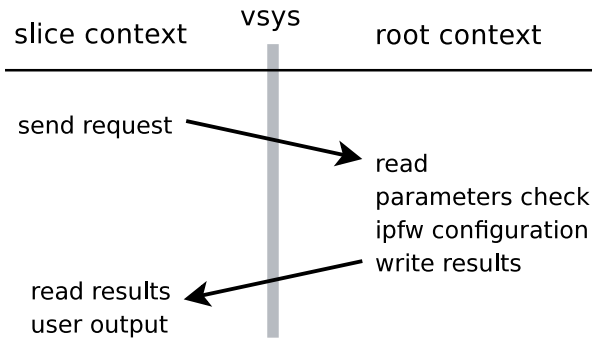



Figure 13: The operations involved in a reconfiguration of the emulator.

## 6. Related work

The two research areas most related to this work are network testbeds and network emulation systems. As mentioned in the Introduction, network testbeds have been an active research subject in recent years, resulting in the development and availability of several testbeds addressing different needs.

### *6.1. Network testbeds*

Two of the most popular testbeds are PlanetLab [6], and Emulab [3]. Both are publicly available to researchers, but differ in several aspects. We have already described PlanetLabextensively in this paper, so we refer the reader to Section 3.1.

Emulab is a public facility whose nodes are mostly concentrated in a single location, and interconnected through a programmable switch to create user-specified topologies. Emulab's strength is the availability of a wide range of experimental environments such as emulation, simulation, real wireless links, and sensor networks. The initial version of the platform relied on Dummynet instances placed between processing nodes to create emulated links with the desired features. Subsequent additions to the testbed include wireless interfaces, Universal Software Radio Peripheral [17] (USRP) devices, and some mobile nodes placed on robots that can be driven around a lab. Emulab users can define the desired topology using the Ns-2 [18] syntax or by a Java GUI. This configuration also covers the definition of hardware and software features of the nodes, wireless capabilities, and mobility. After this stage, the platform maps virtual requirements on physical resources, trying to minimize the use of the physical

resources. The integration of Ns-2 [18] in Emulab makes simulation capabilities available to the platform.

ORBIT [4] (Open Access Radio Grid Testbed) is a testbed based on a large indoor grid of around 400 radio nodes, which can be dynamically interconnected to create arbitrary topologies and wireless channels behaviour. Each node is connected to the network by one wired link, used as a control channel, and two wireless cards, normally used to run experiments. The features of the wireless link, such as transmission power, transmission rate and other high level parameters can be configured. By placing the nodes involved in the communication in different points of the grid, and possibly using other nodes or signal generators as sources of interference, one can study the effect of varying channel characteristics on the communication.

VINI [5] is a testbed platform aimed to test lower layer software, such as routing protocols. VINI provides a wide, shared physical infrastructure where researchers can define arbitrary network topologies and test protocols and applications. Using the VINI platform it is possible for researchers to run their conventional routing software, in a wide environment, exposed to real network conditions and real traffic. Researchers are allowed to control the network behaviour too, reproducing particular network events or injecting controlled failures in the network, in order to test and measure their software in every possible situation.

### *6.2. Emulators*

The second related work area refers to network emulators. Here the spectrum of available solutions ranges from dedicated hardware solutions, generally targeted to the evaluation of MAC protocols, to software-based solutions that run in standalone devices or within standard operating systems.

An in-depth description of Dummynet is already present in Section 3.3 so we will not repeat it here.

A tool with similar features is NISTnet [9], which runs on Linux and also supports the emulation of multiple links with programmable bandwidth and features. Another option for link emulation under Linux is the combination of *tc* [10] and *netem* [13], where the *tc* is in charge of classification and traffic shaping, whereas the *netem* part is in charge of simulating propagation delays and reordering. A significant drawback of *tc* is that it cannot do shaping on the incoming path, which limits its usefulness when the data source is not on a machine equipped with the emulator.

NetPath [11] is a high-performance emulator based on the Click modular router [12]. A custom program is used to create a proper Click configuration with user-defined classifier, delay elements, queues and traffic shapers. NetPath is especially interesting for building dedicated emulation systems, because, borrowing from Click's use of custom device drivers and busy wait techniques, it makes a more effective use of the hardware than a generic operating system.

Modelnet [19] uses a modified version of Dummynet as the basis to build larger emulation engines. In this case a cluster of computers is used to host multiple emulator instances, and a programmable switch takes care of connecting end nodes with the proper emulator instances, compiling a topology description into a proper configuration of switches and emulator instances.

Emulation of networks involving multiple cascaded links can be done with most of the systems described above. Dummynet makes this possible through the reinjection of traffic in pipes multiple times, using classifier rules to model routing decisions. In NetPath and Modelnet, this is achieved by compiling the topology description. Modelnet offers some scaling capabilities because the emulation can be mapped on multiple nodes, and intermediate nodes only need to exchange metadata and not the entire payload of packets.

Imunes [20] is a system based on FreeBSD which supports multiple, virtual network stacks within a single instance of the operating system. The emulated topology is build by different nodes, each one is based on a virtual stack. Nodes are connected through Dummynet instances. This concept can be extended by using virtual machines (Xen, VMWare, VirtualBox, Qemu) to run multiple emulator instances.

Emulation features are also present in network simulators such as Ns-2 [18] and Ns-3 [21], which can drive the simulator with live traffic, and interact in this way with real traffic sources and links.

#### *6.2.1. SatelliteLab*

SatelliteLab is a system with very similar goals to the work presented here, though it uses a completely different approach. SatelliteLab reproduces the behaviour of a link (such as a DSL line connecting a residential customer) by passing traffic

(or an equivalent replica) through the actual physical link involved in the experiment, and using the measured behaviour to artificially delay or drop the traffic subject to emulation. With SatelliteLab one does not need to model a link, but also has no control on the experimental conditions. In this respect, SatelliteLab is more a testbed extension than a real emulator.

## 7. Conclusions

In this paper we have presented in detail the architecture of an emulation extension that we have designed and deployed on PlanetLab, and given an extensive report on the performance and scalability of the tool.

The core of the system is based on the Dummynet emulator, which has been ported to Linux and extended with specific features to improve scalability and performance in this context. In addition to the emulation engine, we have designed and implemented solutions to provide researchers with a simple and intuitive user interface, so that they can focus on their main work without being distracted by the complexity of configuring emulation.

As part of the Onelab2 project, the emulation support has been already deployed on a number of PlanetLab nodes, and we plan to make it available to the entire platform.

The system described in this paper is under active development, especially for the emulation engine which is currently being extended with more and more emulation features, performance optimizations, and more powerful configuration options.

## Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n.224263 – Onelab2.

## References

[1] GENI: Exploring Networks of the Future, http://www.geni.net/, 2009.
[2] FIREWORKS, http://www.ict-fireworks.eu/, 2009.
[3] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar, An integrated experimental environment for distributed systems and networks, SIGOPS Oper. Syst. Rev. 36 (2002) 255–270.
[4] Orbit, http://www.orbit-lab.org/, 2006.
[5] VINI, A virtual network infrastructure, http://www.vini-veritas.net/, 2006.
[6] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, M. Bowman, Planetlab: an overlay testbed for broad-coverage services, SIGCOMM Comput. Commun. Rev. 33 (2003) 3–12.
[7] M. Carbone, L. Rizzo, "dummynet revisited", SIGCOMM Comput. Commun. Rev. 40 (2010).
[8] Linux Vservers, http://linux-vserver.org/, 2006.
[9] M. Carson, D. Santay, Nist net: a linux-based network emulation tool, SIGCOMM Comput. Commun. Rev. 33 (2003) 111–126.
[10] Linux Advanced Routing & Traffic Control, http://lartc.org/, 2002.
[11] S. Agarwal, J. Sommers, P. Barford, Scalable network path emulation, in: MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, IEEE Computer Society, Washington, DC, USA, 2005, pp. 219–228.
[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. F. Kaashoek, The click modular router, ACM Trans. Comput. Syst. 18 (2000) 263–297.
[13] S. Hemminger, Network emulation with NetEm, Linux Conf Au (2005).
[14] L. Rizzo, Dummynet: a simple approach to the evaluation of network protocols, SIGCOMM Comput. Commun. Rev. 27 (1997) 31–41.
[15] OpenWrt, http://openwrt.org/, 2008.
[16] CoTop, http://codeen.cs.princeton.edu/cotop/, 2009.
[17] USRP: Universal Software Radio Peripheral, http://www.ettus.com/, 2009.
[18] The ns-2 Network Simulator, http://nsnam.isi.edu/nsnam/index.php, 2005.
[19] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, D. Becker, Scalability and accuracy in a large-scale network emulator, ACM SIGOPS Operating Systems Review 36 (2002) 271–284.
[20] M. Zec, M. Mikuc, Operating system support for integrated network emulation in imunes (2004).
[21] The NS-3 Network Simulator, http://www.nsnam.org/, 2006.