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 are very popular tools for 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 features 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, largely extended with specific features to improve efficiency on PlanetLab, and to emulate wireless links 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 the whole PlanetLab-Europe testbed. The emulation code itself is also available for all popular operating systems (FreeBSD, Linux, Windows, OS X).

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.

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.

Depending on the case, testbeds are built as a result of a community effort, where each participant contributes computing and networking resources; or they are supported by funding agencies which sponsor strategic initiatives such as GENI [1] and FIRE [2].

The goals of these projects 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], CMU wireless emulator [5]), or routing protocols (VINI [6]). In these cases, the testbed includes components to address the particular problem domain.

Finally, testbeds such as PlanetLab [7] 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 contributions of this paper are i) a system that lets PlanetLab users configure, independently of each other, multiple emulated links for their experiments (Figure 1), and ii) extensions to the Dummynet [8] emulator, which provide improved support for wireless emulation. In addition to this, we added specific packet filtering and demultiplexing features to the emulation engine, to support concurrent PlanetLab users in a robust and efficient way; and a carefully designed user interface that eases the use of emulation within existing experiments.

The rest of the paper is structured as follows. Section 2 briefly gives the motivation for this work. Section 3 presents our first contribution, namely the architecture of our PlanetLab extension and its components: the PlanetLab testbed (Section 3.1), the Dummynet emu-

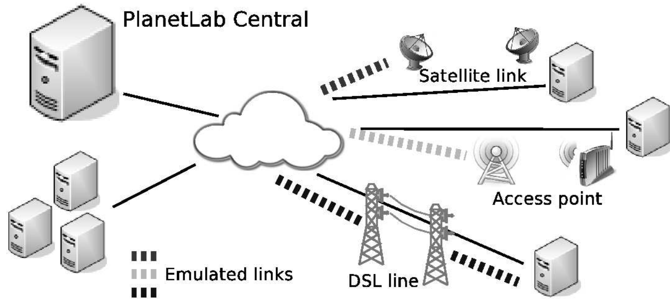


Figure 1: Applications of our PlanetLab extension. Different types of links (dashed) can be emulated on top of existing, physical links (solid).

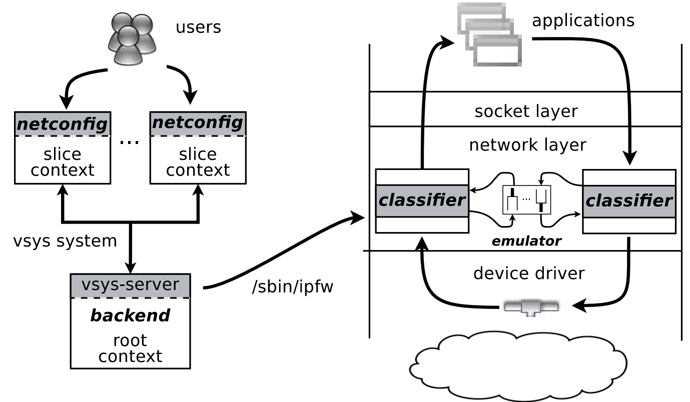


Figure 2: On the left, the interaction between users, vsys frontend and backend (Section 3.1.2). On the right, the flow of packets through network stack, packet classifier and pipes (Section 3.2.3).

lator (Section 3.2 and 3.3), the user interface (Section 3.4) and performance improvements (Section 3.6). Section 4 presents the second contribution of the paper, i.e. emulator extensions to model wireless networks. Experimental results, including performance data, are presented in Section 5. Finally, Section 6 gives an overview of related work.

2. Motivations

PlanetLab [7] 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 in this paper.

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 [8] 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 our goals by 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 dominate 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, we can be confident that in many cases the physical links will not act as communication bottlenecks.

3. Architecture

This Section addresses some important design decisions for our emulation system, namely: 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 we should 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 overall architecture of our emulation system is shown in Figure 2. Users issue commands through a simplified user interface (Section 3.4) to configure the desired features of the emulated links. Requests are then passed to the management code of the PlanetLab node, which processes them and configures the emulation engine making use of specific features (Section 3.6) to improve the performance and robustness of operation in a shared and heavily loaded system such as a PlanetLab node.

Before describing the various components in the system we give a brief description of how PlanetLab works.

3.1. PlanetLab

A PlanetLab instance 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* 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 Vserver [9] 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 rights to operate on the root context are limited and strictly controlled using the *vsys* service, which controls how sliver issue requests for 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 and communicating with backend programs that run in the root context (see Figure 2, left). Vsys backends are installed by the system administrator; they receive requests from the slivers together with the identity of the invoking sliver, analyse them, and possibly invoke any required system call in the root context to perform the desired action.

In our system, the vsys backend authenticates user requests to configure emulation, and extends them as described in Section 3.5 to avoid interfering with the traffic belonging to other users.

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 [10], *tc* [11], and netpath [12] (the latter is based on Click [13]).

We dismissed NISTnet because it is not available as a standard component in PlanetLab. *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 [14], to model features such as propagation delays and reordering. Second, and most important, *tc* is already used within PlanetLab nodes to 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 [12] 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.

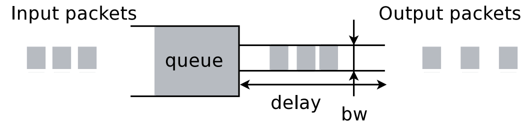


Figure 3: The basic components of a Dummynet pipe.

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 researchers 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.

3.2.1. Dummynet

Dummynet [8] is a network emulator developed under FreeBSD several years ago [15], 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 [8]; the next Section reports only the details (including newly introduced features) that are relevant for this work.

3.2.2. The emulation engine

dummynet (the emulator) can create multiple instances of an object called *pipe*, which in its basic version (Figure 3) models a network link 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.

In this project we use pairs of pipes to emulate the two directions of a point to point link, and a single pipe to emulate a shared media such as a wireless link.

3.2.3. 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.

The classifier is programmed through 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 sliver associated to a packet. A packet is tested against each of the rules, in numeric order, and the first matching rule terminates the search and causes the execution of the associated action. For our purposes, the action of interest is to send the packet to a pipe, which will in turn delay or drop 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.

3.3. Porting Dummynet to Linux

As part of this work we needed to build a Linux port of Dummynet. 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. As a result, we ended up providing wrappers that map FreeBSD data structures and kernel APIs into the equivalent components of the underlying operating system. In particular, `mbufs` (the packet representation in FreeBSD) were mapped to `skbufs`, and calls to the packet interception mechanism (`pfil` in FreeBSD) were mapped to `netfilter` calls.

Eventually, this led us to extend Dummynet availability even to Windows, as the porting approach described above made this possible with a relatively limited effort. As of this writing, the emulator is now available on all major operating systems, including Windows and OpenWrt [16], which is more and more used in various research prototypes as well as actual deployments.

3.4. 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 misconfigurations affecting other slivers. To solve these issues we offer users a simplified interface based on three types of link configurations: *server*, *client* and *service*.

A **client** configuration emulates a node hosting applications 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 pipes emulating the upstream and downstream links.

A **server** configuration 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.

Finally, a **service** configuration 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.

Users configure one or more emulated links, and define the traffic affected, with one-line commands such as those in Figure 4. Each line defines an emulated bidirectional link, indicating which traffic should be affected and the configuration of the pipes (after the `IN` and `OUT` keywords) emulating the downstream and upstream links. A special case is represented by shared media (e.g. WiFi networks) where the two directions share the same physical channel and are emulated using a single pipe, whose features are set using the `SHARED` keyword.

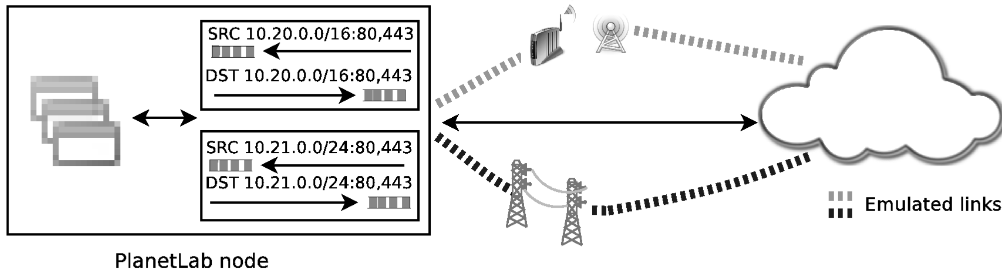
In the example in Figure 4, we emulate a multihomed client where selected HTTP/HTTPS traffic to a /16 subnet goes through an emulated 3G link, whereas other traffic for a /24 subnet goes through a link with ADSL-like features.

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, or the extended features described in Section 4) can be specified independently for the two directions of the communication, to cover the case of asymmetric links.

3.4.1. Under the hood

The `netconfig` program does nothing but pass the request to the root context, together with the identity of the sliver issuing the request. The ports/addresses specified as arguments are used to detect whether we are creating a new emulated link or modifying/deleting an existing configuration. A request normally installs a couple of rules for the classifier to select the desired inbound and outbound traffic, and configures one or two pipes with the specified features. Figure 5 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,




```
netconfig client 80,443@10.20.0.0/16 SHARED profile myconfigs_802.11b-11
netconfig client 80,443@10.21.0.0/24 IN bw 512kbit/s delay 8ms OUT bw 10Mbit/s delay 3ms
```

Figure 4: An example configuration to emulate a multihomed client. Two emulated links intercept traffic for two different subnets; the first one emulates an 802.11b network at 11 Mbit/s whose configuration is in the file passed as argument, whereas the second emulates an asymmetric DSL line.

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

ports and addresses, other filtering options) come from the user's request.

3.5. Isolation between users

In order to make sure that rules generated by one sliver cannot match traffic belonging to another sliver, the backend adds a `sliver X` option to all rules. The sliver ID, `X`, is extracted 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.

3.6. 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 3.2.3), 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 6: after a small number 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 N), followed by at most O(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.

Section 5.1.1 presents detailed performance measurements to quantify the per packet cost in the worst case, and show the effectiveness of our approach.

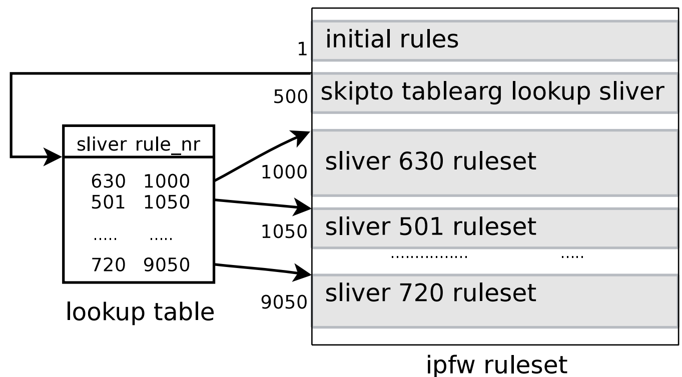


Figure 6: The structure of the ruleset used in the classifier, and the sliver table used to perform a fast dispatch of packets to the block of rules for each sliver.

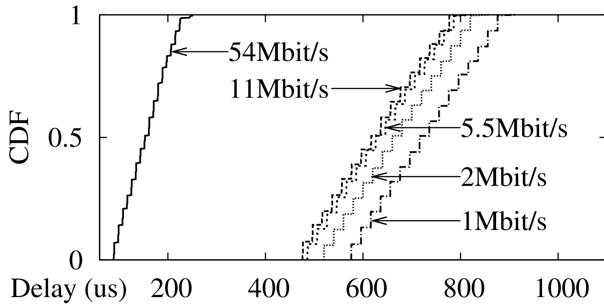


Figure 7: *delay profiles* describe the distribution of MAC-level overheads. The staircase comes from random backoffs (slot times differ in 802.11g and 802.11b), while the remaining part comes from other framing bits and MAC-level acks.

4. Extended emulation features

While dummynet pipes (Figure 3) model very closely a wired, point-to-point communication link, emulating a wireless channel must take into account many other phenomena, such as the effect of the MAC protocol, noise-induced errors, the sharing of the link with other stations, external interference, and variations of the channel conditions over time. Detailed emulators of such phenomena do exist [5] and [17] but in the PlanetLab context (where applications contend network and computing resources with many other applications) a high resolution emulation would be overkill. Besides, results would still depend on a very precise model of the system, which in practice is extremely hard to come up with. As a consequence, we introduce a few simplified mechanism to reproduce the wireless channel, as described in the following.

4.1. MAC overheads

MAC overheads are modeled using *delay profiles*: for each packet transmission, we extend the transmission time by a random amount whose distribution (CDF) is described by an empirical function, such as those shown in Figure 7. This extra time models the overhead (backoffs, preambles, MAC framing, link-level acknowledgements) incurred in transmitting a packet on a wireless channel. These values can be, in principle, derived from the relevant standards, though in practice it may be useful to extract these numbers from actual measurements because different hardware implements the standards in different ways [18] and [19]. As an example, typical 802.11b/g transmissions incur an overhead given by a fixed amount of time (framing plus MAC-level acknowledgement), plus a variable backoff, as shown in Figure 7.

4.2. Noise

Noise on the channel may cause erroneous *symbol*[1] decoding in the receiver, which in turn leads to packet drops and retransmissions. The relation between noise (or better, the Signal-to-Noise Ratio, SNR) and the Packet Error Rate (PER) depends on the modulation, channel width and data rate.

The error model used in our implementation is based on the formulas used in the Ns-3 simulator for the DSSS [20] and the NIST [21] models. For 802.11g modulations, the latter has been shown to be a better approximation of reality than the YANS model [22] previously used. The 802.11b channel model is based on the communication theory formulas described in [23] and validated against a clear channel by [20, Sec. 1.2.3].

As an example, the equation for DBPSK at 1 Mbps, and the approximation of [24] for the 2 Mbps DQPSK modulation are the following (at these rates, 1 symbol = 1 bit):

$$BER_{1M} = \frac{1}{2} e^{-\frac{E_b}{N_0}}$$

$$BER_{2M} = \frac{\sqrt{2}+1}{\sqrt{8\pi\sqrt{2}}} \frac{1}{\sqrt{\frac{E_b}{N_0}}} e^{-(2-\sqrt{2})\frac{E_b}{N_0}}$$

where E_b is the energy per bit and N_0 is the noise value at the receiver. Knowing the SNR, the channel's bandwidth B and the data rate R, we can easily compute $E_b/N_0 = \text{SNR}\frac{B}{R}$. The resulting BER curves for various rates are shown in Figure 8.

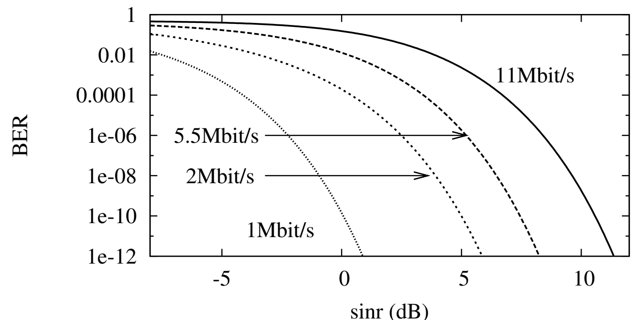


Figure 8: The relation between SNR and BER for different rates.

In turn, for modulations using one symbol per bit, the relation between BER, the packet size in bits (L), and the Packet Error rate (PER), has the form

$$PER = 1 - (1 - BER)^L$$

For other modulation we should replace BER with Symbol error rates and L with the size in symbols. As an example, Figure 9 shows the packet error rate for different values of BER and L.

To avoid excessive overhead at runtime, for each modulation we compute offline the PER for the packet sizes and SNR values of interest, so that at runtime a simple table lookup will give us the error probability for the given packet, and we generate a random number to determine the fate of the packet. If a packet is lost, we consider the channel busy for the time required for the transmission, and emulate the link level retransmission protocol for the number of times specified by the protocol.

[1] Depending on the modulation, a symbol carries one or more bits.

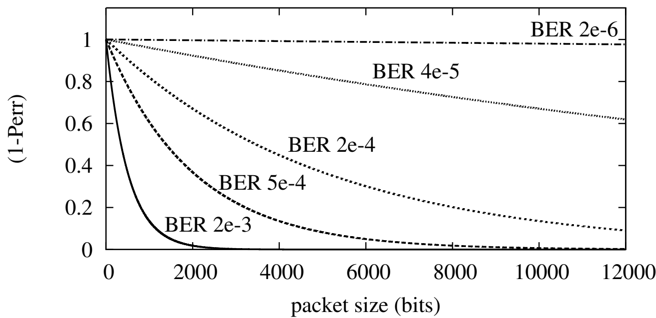


Figure 9: Transmission probability for different BER values.

4.3. Network load and competing stations

The presence of competing stations is modeled using two parameters. One, called *netload*, represents the fraction of airtime which is unavailable due to competing traffic (without accounting for collisions, which are handled separately). As an example, Beacon signals from an access point make consume approximately 1% of the airtime. Using a very coarse approximation, all packet transmission times are multiplied by $\frac{1}{1-n}$ so that, on average, we will see a throughput which is proportional to the available fraction of airtime.

A second parameter is called *collision* and indicates the probability of a collision. *collision* and *netload* are related, but the relation varies depending on the traffic patterns (see [25, 26]) so we really need two separate parameters to model the two phenomena. The *collision* is used to randomly force an error on transmission, which adds to those caused by noise. These errors are handled in the same way as those induced by noise, i.e., considering the channel busy for the actual transmission time and then doing a link level retransmission (and, eventually, reporting a transmit failure upstream).

4.4. Time-varying links

Time-varying links model the variability of links over time, e.g. due to mobility. When using time-varying links, the system dynamically alters the configuration of the pipes (including bandwidth, delay profiles, loss rates) according to a transition graph specified by users as in Figure 10.

Each node in the graph specifies a configuration (bandwidth, losses, SNR, netload, collisions) for the link. Each configuration lasts for a random amount of time, whose distribution is specified as an empirical curve associated to the node. Changes from one configuration (node) to the next one occurs with the probability specified on the arcs connecting the nodes. Due to the coarse timescales (compared to packet transmission times) involved in the process, reconfigurations are controlled by a user-space process without the need of additional mechanisms within the kernel.

5. Experimental results

The emulator described here has been integrated in the nodes of the PlanetLab-Europe platform since 2010, and it is being updated with new features as we develop them. In this Section we report various performance and accuracy metrics of our system, and discuss how they fit the requirements of the platform. In particular, we cover the following:

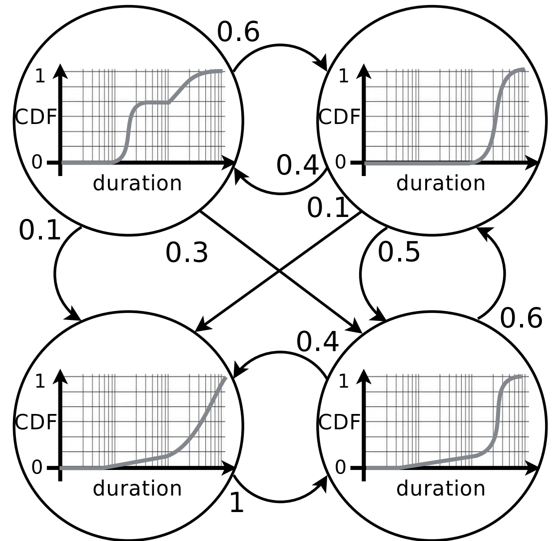


Figure 10: An example of the information used to implement *time-varying links* (Section 4). The link remains in each state for intervals of time with the given distribution, and then moves to a new state with the probability specified on the arcs.

- per-packet processing costs and scalability. This involves the classifier (affecting all packets), emulation, and reconfiguration costs;
- accuracy of the emulator, how it affects experiments compared to other "noise" sources existing in PlanetLab nodes;
- validation of the wireless emulation features, comparing emulation results with experiments in real settings.

5.1. Emulator overhead

In Dummynet, the per-packet processing costs are made of two components: classification, which grows linearly with the number of rules, and emulation, which is essentially constant except for a small $O(\log \mathrm{L})$ component, where L is the number of active pipes. Measurements made by the authors in a previous work [8] further decompose cost components, with relevant results summarized in the first part of Table 1. These measurements were made by generating traffic from a local source, and dropping it at various stages of the network stack, as shown in Figure 11. The average cost of specific components (e.g. certain classifier instructions, or emulation costs, etc.) was determined by measuring the throughput (in packets per second, PPS) in various conditions and computing the difference between the

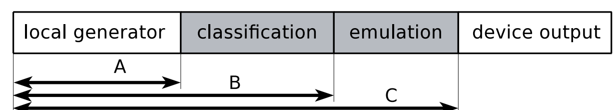


Figure 11: To evaluate the per-packet processing overhead, we measure the PPS rate with a local generator and traffic dropped in different points of the protocol stack. Differences between the measurements give the cost of each processing block.

Parameters (from [8])	Cost (ns)
Enter the classifier	400
Test one simple rule	36
Traverse a pipe	700-1300
Overhead for L pipes	$\log(L) \cdot 100$
New measurements (Sec. 5.1.1)	Cost (ns)
Table lookup, E entries	$220 + \log(E) \cdot 11$
Worst case no pipe, full table	< 1000
Worst case 1 pipe, full table	2150
Worst case 20 links/user	< 4000

Table 1: Summary of emulation costs, both individual components and worst case configuration.

inverse of the PPS rates. Absolute values change depending on the hardware, but the ratio between the components is approximately constant across different hardware platforms.

5.1.1. Per packet costs

The PlanetLab version of the classifier was extended with an optimization (based on a table lookup) to quickly jump to the user-specific part of a ruleset (Section 3.6). Using the same techniques (Figure 11) and the same hardware as in the previous section, we measured the cost of packet generation alone (case A in Table 2); packet generation plus classification through a single rule (case B_1); classification using a table with a single entry (case B_T) or a full table with 2^{16} entries (case B_{FT}); and finally, classification using a full table plus the traversal of a worst-case pipe[2] (case C). This really represents the worst case for a PlanetLab node with the maximum number of slivers (2^{16}), all using emulation.

Table 2 presents the results of the experiment, listing average and standard deviation for each test case (a 10-second run repeated 100 times). 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 2: Results of the measurement of per-packet overheads with different configurations of the classifier and emulator.

Starting from these measurements, individual cost components have been measured and are summarized in the bottom of Table 1. Table lookups take approximately 220 ns with a single entry, but then the variable cost scales logarithmically with the number of entries so even the worst case ($B_{FT} - B_1$) is only taking about 400 ns. Overall, the difference between cases A and C shows that in the worst case emulation consumes roughly 2150 ns with a full table and one active pipe. Considering that each emulated link introduces two extra rules in the per-user part of the ruleset, even allowing each user to define up to M emulated links, we need to add approximately $M \cdot 2 \cdot 36$ ns to this time, and another small $log(L)$ contribution coming from the presence of multiple, simultaneously active pipes.

In conclusion, the presence of the emulator increases the per-packet processing time by less than 1 μs for each packet not subject to emulation, and less than 4 μs for packets subject to emulation.

5.1.2. Throughput

To determine how the emulation overhead impacts the performance of a PlanetLab node, we have studied the network load on the platform nodes using the data reported by the CoTop monitor [27]. CoTop computes the average traffic of a node over a 1-minute interval. The highest values we saw were around 30 Mbit/s. We do not have data on the number of Packets Per Second (PPS) measured on the nodes, but 30 Mbit/s correspond to a value between 2500 and 58000 PPS, considering a packet size of 1500 and 64 bytes respectively.

Even taking the worst case, a 4 μs per-packet overhead at 60 KPPS 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.

5.1.3. 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, use the vsys system to communicate, and then makes a few calls to `/sbin/ipfw` to update pipes' and classifier configuration. Figure 12 shows the components involved.

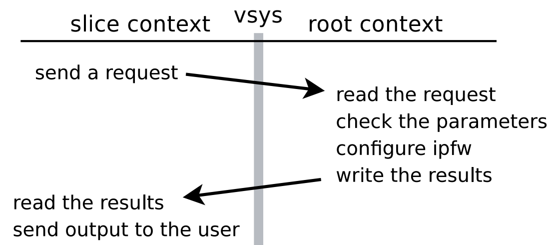


Figure 12: The operations involved in a reconfiguration of the emulator.

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 means that we can already handle tens of reconfiguration requests per second. The reconfiguration costs can be reduced by one order of magnitude by moving to a compiled version of the backend.

[2] in previous work we determined that the worst case for emulation costs is a link with only delay and no bandwidth limit.

5.2. Emulator accuracy

An important aspect to be evaluated in an emulator is how well it is able to reproduce the timing computed by the model. Once again, [8] makes a detailed analysis of this aspect, pointing out three main sources of inaccuracy: timer resolution, competing traffic, system load. Here we analyse the impact of these three factors in the context of PlanetLab.

The first source of inaccuracy, peculiar to the specific approach used by our emulator, is the timer resolution, which on PlanetLab nodes is 1 ms. This means that, apart from other sources of error, all packet transmissions and receptions will be subject to a 1 ms timing inaccuracy. The question is whether this error is acceptable for the type of measurements that we want to make, and how it compares to other timing errors that are induced by other parts of the system.

Competing network traffic, even in absence of emulation, causes packets to be moved in time by at least the time to transmit one maximum-sized packet at line rate. According to the information reported by PlanetLab monitoring tools, most PlanetLab machines are attached to 100 Mbit/s switches despite being equipped with Gbit interfaces. As a consequence, this introduces timing errors of at least 1.2 ms, already larger than those induced by the timer resolution.

Finally, the largest and most unpredictable source of inaccuracy comes from competing system load. This affects both the kernel (which may delay processing of network traffic when the CPUs are busy) and the scheduling of user applications, which may be delayed by large amounts of time (up to hundreds of milliseconds) by other computationally intensive applications.

As an experiment to measure the first component (kernel load) we ran a series of ping tests between pairs of colocated PlanetLab nodes. Ping times are mostly unaffected by user process scheduling, because processing on the responding side occurs entirely within the kernel, and at the sending side packets are usually timestamped as soon as they reach the kernel, so measurements do not account for scheduling delays. Even in this favourable setting, the values measured on our PlanetLab nodes have large variations. Figure 13 plots the CDF of ping response times between some pairs of colocated PlanetLab nodes. 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.

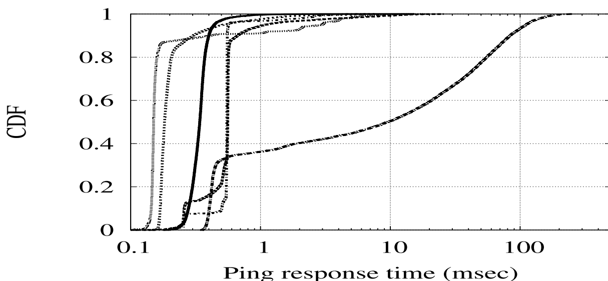


Figure 13: Distribution of ping response times between some pairs of colocated PlanetLab nodes.

From the above we can conclude that the emulation inaccuracy 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. Validation of wireless emulation

The wireless emulation features are entirely new, so we need to validate how well they reproduce the behaviour of a real wireless network. Our validation tests have been conducted by first running measurements in real wireless networks under different operating conditions, After that, we have run the same experiments on the emulator configured to model the original wireless network. Because we use a coarse model of the wireless channel, our main metric will be the throughput that we can achieve on the link (real or emulated one).

In general, our tests involve one station (STA) and one Access Point (AP) located in close proximity (Figure 14), and a server connected to the AP acting as the other endpoint of the communication. A nearby monitor node is used to collect traces of interesting events in the wireless medium, such as actual inter-packet delays, retransmissions and so on, and to help investigate possible differences between real and emulated settings.

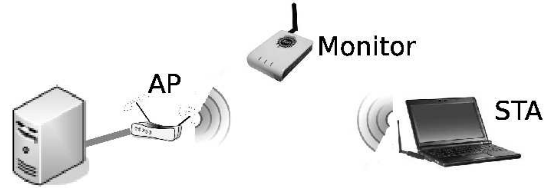


Figure 14: The experiment scenario.

The uncertainty on actual network conditions, especially for the presence of unknown interfering stations, makes it hard to get a close match between emulation and real world experiments. There are however some simple, baseline cases where we can expect very close results. One such example is when a node is transmitting a unidirectional stream of packets on an otherwise idle channel. While we cannot guarantee that a channel is exempt from interference, running many short (5-10 s) experiments on different channels and times of the day gives a reasonable probability that at least some of those experiments run with little or no interference.

With this in mind, a first set of experiments has been run to measure the maximum throughput achieved at different 802.11b and 802.11g rates, and then verify whether the emulator is able to reproduce them. We lock the access point at a fixed rate, and send a unidirectional stream of maximum-sized UDP packets from the server to the station, using `iperf` [28]. The choice of UDP is to avoid interference from TCP acknowledgements flowing in the reverse direction.

Figure 15 plots the distribution of throughputs obtained in a large (50 to 200) number of experiments at each different rate. The spread of values is mostly due to occasional interference on the channel. The column labeled *Real* in Table 3 reports max, average and standard deviation for these measurements. The maximum values should represent the maximum channel capacity at each rate (net of all overheads), and represent our target for the emulation results.

The emulated experiments were made by running the same set of tests over an emulated link. For the delay profile (Sec-

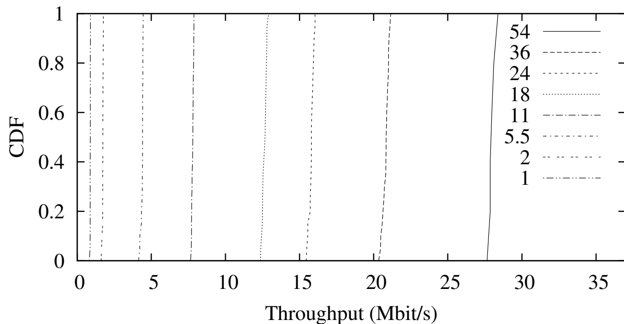


Figure 15: Throughput CDF for different rates, in Mbit/s.

tion 4.1), measurements showed that our access point does random backoffs using only 16 slots (instead of the 32 indicated by the standard), so we used a profile configured accordingly. Also, we took into account the fact that beacons (occurring 10 times per second, and consuming approximately 1.14 ms each in our network) consume about 1% of the airtime, and modeled that as an equivalent *netload*. We set the *collision* parameter to 0 because, in our case, there is no interference between beacons and data packets (they both originate from the same device).

Table 3 compares the throughput in real wireless network and in the emulated experiments. For what we said so far, we should compare the max values from both sides, and we see that there is a close match between the two. Of course, this result only tells us that we can reproduce very closely a specific configuration of the system. At the same time, Figure 15 reminds us that real world experiments are bound to have large variations due to unpredictable channel's conditions.

Nominal	Real			Emulator		
Rate	Max	Avg	sd	Max	Avg	sd
54000	28390	27970	281.4	28570	28500	46.6
11000	7879	7788	68.7	7831	7806	27.3
5500	4455	4377	85.4	4405	4400	3.4
2000	1767	1737	39.2	1772	1771	0.548
1000	894	881	16.7	900	899	0.448

Table 3: Real and emulated throughput for unidirectional UDP traffic at different 802.11b and 802.11g speeds. Rates are in Kbit/s.

5.3.1. Modeling competing traffic

To show the effect of competing traffic and how it can be modeled, we measure the throughput of a unidirectional TCP flow. In this case the reverse ACK traffic not only consumes airtime (which is already accounted for by the emulator), but also generates some amount of collisions because data and acks are somewhat synchronized with each other. The presence of random backoffs can reduce the chance of collisions, but not entirely eliminate them. Their probability depends heavily on the number of active stations, traffic patterns and overall network load [25, 29]. The number of slots, CW, used for random backoffs introduces a factor ϕ/CW in the probability of collision, but the exact proportionality factor needs to be determined experimentally.

In Table 4 we compare the throughput of a unidirectional TCP flow on a real, 11Mbps wireless link, and over emulated links with different settings for the collision parameter. As mentioned, our devices use $CW = 15$ so we have explored various values for the collision parameter in the vicinity of $1/CW$. Having collision set to 0 clearly overestimates the available bandwidth, and in these experiments it seems that a factor near $1/CW$ models the channel very closely. Due to the different conditions is not possible a direct comparison with the results reported in saturated [25] or non-saturated [29] networks, however they represents a good starting point in the evaluation of packet collisions.

Scenario	Throughput (Kbit/s)		
(real or emulated)	Max	Avg	sd
real	5729	5681	32.0
em, coll = 0	6266	6237	10.9
em, coll = 0.04	5936	5876	21.4
em, coll = 0.06	5733	5692	19.6
em, coll = 0.07	5657	5601	26.3
em, coll = 0.08	5546	5520	21.4

Table 4: Real and emulated throughput using an 11Mbps nominal rate and TCP. Emulated experiments were run using different values for the collision parameter, as shown in the first column.

6. Related work

We conclude the paper with an overview of related work. The research areas most related to this paper are network testbeds, and wireless emulation systems, including the modeling of wireless links.

6.1. Network testbeds

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.

Two of the most popular testbeds are PlanetLab [7], and Emulab [3]. Both are publicly available to researchers, but differ in several aspects. We have already described PlanetLab extensively 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 [30] (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 [31] 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 [31] 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. 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 [6] 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 area, 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.

The APE testbed [32] is a research project of the Uppsala University and allows to evaluate mobile and ad hoc routing protocol in a real-world environment. The testbed simplify the process of performing complex tests in real networks, providing a set of tools to define network scenarios, to collect data and to analyze the results. The mobility of devices is implemented by a feature called "virtual mobility", and it is implemented by SNR changes, making possible experiments reproducibility. The APE testbed is build by a set of public available tools, and it is able to support the protocols implemented on a Linux systems.

6.2. Emulators

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

Dummynet has been already described in detail in Section 3.2.1. Similar features are available in NISTnet [10], 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* [11] and *netem* [14], 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 [12] is a high-performance emulator based on the Click modular router [13]. 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 [33] 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 [34] 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 [31] and Ns-3 [17], which can drive the simulator with live traffic, and interact in this way with real traffic sources and links.

SatelliteLab [35] is a system with goals similar 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 probes 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.

6.3. Emulating wireless networks

Emulating wireless links used for data communication may be an extremely complex problem, as it has to deal with physical as well as MAC protocol aspects.

Channel models and the effect of noise on data communication, in presence of specific modulations, is the subject of communication theory research, see e.g. [23]. Some packet level simulators provide different options, according to the propagation model used; as an example, the latest Ns-3 version gives the choice between the YANS [22], DSSS [20] and NIST [21] error models.

Emulating the MAC protocol in presence of multiple stations is doable [36] but computationally intensive, as we need to reproduce the behaviour of individual stations. Several studies [25, 26] have tried to determine analytical models of aggregate parameters (e.g. probability of collisions, channel usage). These results can be of use in coarse-level emulations as the one we have implemented.

When the amount of computation involved in wireless emulation becomes too significant, e.g. because one wants to reproduce multipath, interference, etc., it might be useful to use

dedicated processing units for this purpose. As an example, the CMU wireless emulator [5] uses FPGA-based hardware to emulate signal propagation in space, enabling controlled and reproducible experiments.

7. Conclusions

In this paper we have presented 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 support wireless emulation, and improve scalability and performance in the PlanetLab environment. In addition to the emulation engine, we have designed and implemented a simple and intuitive user interface, so that researchers 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 the PlanetLab-Europe testbed, and is being updated with new features as we develop them.

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] K. Borries, G. Judd, D. Stancil, P. Steenkiste, Fpga-based channel simulator for a wireless network emulator, in: Vehicular Technology Conference, 2009. VTC Spring 2009. IEEE 69th, pp. 1 –5.

[6] A. Bavier, N. Feamster, M. Huang, L. Peterson, J. Rexford, In vini veritas: realistic and controlled network experimentation, in: Proc. of SIGCOMM 2006, ACM, New York, NY, USA, 2006, pp. 3–14.

[7] 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.

[8] M. Carbone, L. Rizzo, Dummynet revisited, SIGCOMM Comput. Commun. Rev. 40 (2010).

[9] Linux Vservers, http://linux-vserver.org/, 2006.

[10] M. Carson, D. Santay, Nist net: a linux-based network emulation tool, SIGCOMM Comput. Commun. Rev. 33 (2003) 111–126.

[11] Linux Advanced Routing and Traffic Control, http://lartc.org/, 2002.

[12] S. Agarwal, J. Sommers, P. Barford, Scalable network path emulation, in: MASCOTS '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 219–228.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. F. Kaashoek, The click modular router, ACM Trans. Comput. Syst. 18 (2000) 263–297.

[14] S. Hemminger, Network emulation with NetEm, Linux Conference, Canberra, Australia (2005).

[15] L. Rizzo, Dummynet: a simple approach to the evaluation of network protocols, SIGCOMM Comput. Commun. Rev. 27 (1997) 31–41.

[16] OpenWrt, http://openwrt.org/, 2008.

[17] NS-3, http://www.nsnam.org/, 2006.

[18] G. Bianchi, A. Di Stefano, C. Giaconia, L. Scalia, G. Terrazzino, I. Tinnirello, Experimental assessment of the backoff behavior of commercial ieee 802.11b network cards, INFOCOM 2007. IEEE (2007) 1181–1189.

[19] A. Di Stefano, A. Scaglione, G. Terrazzino, I. Tinnirello, V. Ammirata, L. Scalia, G. Bianchi, C. Giaconia, On the fidelity of ieee 802.11 commercial cards, Proceedings of the First International Conference on Wireless Internet (2005) 10–17.

[20] T. Pey, G. Henderson, Validation of ns-3 802.11b PHY model, http://www.nsnam.org/∼pei/80211b.pdf, May 2009.

[21] T. Pey, G. Henderson, Validation of OFDM error rate mode in ns-3, http://www.nsnam.org/∼pei/80211ofdm.pdf, 2010.

[22] M. Lacage, T. R. Henderson, Yet another network simulator, in: Proceeding from the 2006 workshop on ns-2: the IP network simulator, WNS2 '06, ACM, New York, NY, USA, 2006.

[23] J. Proakis, M. Saleh, Communication systems engineering (1994).

[24] G. Ferrari, G. Corazza, Tight bounds and accurate approximations for dqpsk transmission bit error rate, Electronics Letters 40 (2004) 1284 – 1285.

[25] G. Bianchi, Performance analysis of the ieee 802.11 distributed coordination function, Selected Areas in Communications, IEEE Journal on (2000) 535 –547, vol.18.

[26] D. Malone, P. Clifford, D. Leith, Mac layer channel quality measurement in 802.11, Communications Letters, IEEE (2007) 143 –145 vol.11.

[27] CoTop, http://codeen.cs.princeton.edu/cotop/, 2009.

[28] C. Hsu, U. Kremer, P. P. Models, Iperf: A framework for automatic construction of performance prediction models, In Workshop on Profile and Feedback-Directed Compilation PFDC, Paris, France (1998).

[29] D. Malone, K. Duffy, D. Leith, Modeling the 802.11 distributed coordination function in nonsaturated heterogeneous conditions, Networking, IEEE/ACM Transactions on 15 (2007) 159 –172 vol.15.

[30] USRP: Universal Software Radio Peripheral, http://www.ettus.com/, 2009.

[31] The ns-2 Network Simulator, http://nsnam.isi.edu/nsnam/index.php, 2005.

[32] H. Lundgren, D. Lundberg, J. Nielsen, E. Nordstrom, C. Tschudin, A large-scale testbed for reproducible ad hoc protocol evaluations, in: Wireless Communications and Networking Conference, (WCNC) 2002 IEEE, volume 1, pp. 412 – 418 vol.1.

[33] 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.

[34] M. Zec, M. Mikuc, Operating system support for integrated network emulation in imunes (2004).

[35] M. Dischinger, A. Haeberlen, I. Beschastnikh, K. P. Gummadi, S. Saroiu, Satellitelab: adding heterogeneity to planetary-scale network testbeds, in: SIGCOMM '08:, ACM, New York, NY, USA, 2008, pp. 315–326.

[36] G. Bianchi, L. Fratta, M. Oliveri, Performance evaluation and enhancement of the CSMA/CA MAC protocol for 802.11 wireless LANs, in: Personal, Indoor and Mobile Radio Comm. PIMRC'96, volume 2, pp. 392–396.