Adding emulation to Planetlab nodes*

Paper ID: 137 – 12 pages

Marta Carbone
Dip. di Ingegneria dell'Informazione
Universita' di Pisa, Italy
marta.carbone@iet.unipi.it

Luigi Rizzo
Dip. di Ingegneria dell'Informazione
Universita' di Pisa, Italy
rizzo@iet.unipi.it

ABSTRACT

Network testbeds have become very popular to support research on network protocols and distributed applications. When it comes to reproducing network behaviour, testbeds range between two extremes: use a fully emulated network, as in EmuLab, which yields very reproducible experiments but might be a poor representation of reality; or communicate through the real Internet, as in PlanetLab, resulting in more realistic but less reproducible scenarios. Having both features available in the same testbed, and being able to choose and mix the two at will, is clearly interesting for researchers.

In this paper we make two contributions. First, we show how we ported the Dummynet emulator to Linux, making the tool available on that platform. Second, and more importantly, we present an extension of the PlanetLab testbed to add emulation capabilities to all nodes. Our extension uses Dummynet as the basic emulation engine, and provides mechanism to let PlaneLab users independently and concurrently configure emulated links on which to run their experiments. This gives users the advantages of emulation while not giving up the opportunity of running their tests in a large and heterogeneous testbed with realistic network conditions.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Emulation, experimentation, performance

*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.

ACM CoNEXT 2009, December 1–4, 2009, Rome, ITALY.

Keywords

Internet, network testbeds, emulation, performance evaluation

1. INTRODUCTION

In recent years we have seen a significant growth in the deployment of testbeds to support research on network protocols and distributed applications. The primary motivation behind most of these testbeds is the same: make available to researchers a system that, for its size and features, would not affordable for individuals or even single institutions. Depending on the case, the scale of the testbed is achieved as a result of a community contribution, where each participant contributes computing and networking resources; or thanks to the support of funding agencies, which sponsor strategic initiatives such as GENI [3] and FIRE [2].

Such 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 actual target of each testbed varies. Some of them, such as EmuLab [1], are focused on providing a very reproducible environment, in terms of node capacity or network resources. Here, the system makes heavy use of virtualization and emulation techniques to configure the environment so that researchers will not suffer from external interference while running their experiments.

Other testbeds address specific aspects, such as the study of wireless networks (ORBIT [9]), or routing protocols (VINI [14]), or mesh networks (roofnet [16]), or sensor networks (MoteLab [25]). In these cases, the testbed includes components to address the specific problem domain. As an example, ORBIT has well defined placement of nodes so that one can correlate radio behaviour with distance or position; it also has programmable radio sources to create specific interference patterns, and test even low-level aspects of the

behaviour of the MAC protocol. VINI provides support for creating tunnels and virtual interfaces, so one can create his own overlay to experiment with routing protocols.

Finally, testbeds such as PlanetLab [20] are more oriented towards providing a realistic snapshot of the real Internet. In this case, heterogeneity is a desirable feature of the platform, even if the price to pay is some unpredictable network behaviour. In fact, even unpredictability can be seen as a feature rather than a bug of the platform, because it exposes applications to the same conditions that would experience when deployed.

This paper focuses on PlanetLab, which is a large distributed testbed, and on its software, MyPLC [7] that lets users create their own instance of the testbed.

PlanetLab owes much of its popularity to the fact that institutions (especially academic ones) can become part of a large and growing platform with only a modest contribution in terms of resources. In return, users gain access to hundreds of nodes distributed across the Internet. The heterogeneity of systems and network locations, together with the size of the testbed, permits experimenting with network dynamics in a way that would be difficult to achieve within a laboratory or in a concentrated testbed.

A side effect of this heterogeneity is some lack of control on the reproducibility of experiments, because network conditions between nodes are typically unknown and variable over time. Emulation techniques are indeed well known and widely used to produce controlled environments, but PlanetLab nodes do not have this feature.

The main contribution presented in this paper is an extension that we developed to add emulation capabilities to PlanetLab. With this work, PlanetLab users gain the ability to configure, independently of each other, the actual features of the network. The extension has been implemented by porting the Dummynet [18] emulator to Linux, and this is a second contribution presented in this paper. Dummynet is extremely popular among researchers, and over time we have received many requests to make it available to other platforms than FreeBSD. We took the chance of the OneLab2 project [12], in which this work is being done, to address this request and make Dummynet available to Linux users.

The rest of the paper is structured as follows. Goals and motivations of this work are presented in Section 2, followed by a description of the components involved in our work: the PlanetLab testbed, and the Dummynet emulator. Section 5 documents how we ported Dummynet to Linux, while Section 6 shows how emulation is made available to PlanetLab users. Experimental results, including performance data, are presented in Section 7. Finally, Section 8 gives an overview of related work.

2. GOALS AND MOTIVATIONS

PlanetLab [20] is a network testbed made by roughly a thousand of nodes distributed throughout the world and contributed by participating organizations. This distributed testbed gives users and researchers a realistic snapshot of the Internet where they can deploy new software, run experiments and measure network performance.

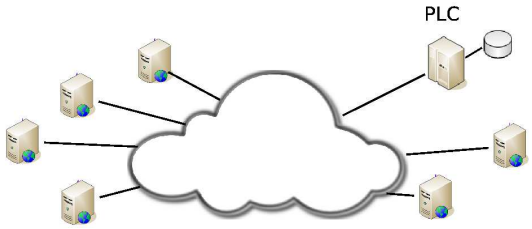

Figure 1: The PlanetLab architecture. Nodes are spread among participating sites, and connected to the Internet. The PLC is the central management site for the testbed.

The testbed is widely used and interesting due to the number of hosts and to the relative heterogeneity of network links and machines it provides. On the weak side, the lack of any control on the conditions of the network make it hard to obtain reproducible experiments, and even harder to run tests under specific conditions. This calls for the introduction of some form of configuration of the network features. Emulation is one possible approach, that is used successfully in other platforms such as EmuLab where the emulation is implemented by a configurable network interconnection, using FreeBSD machines running the Dummynet software to provide the desired emulation of network features.

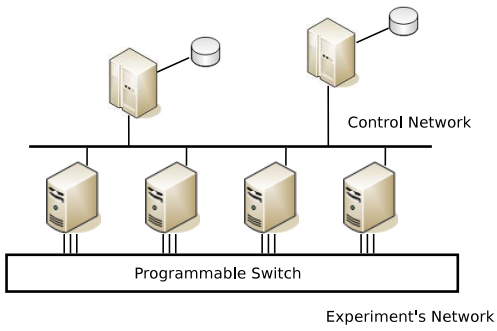

Figure 2: The Emulab architecture. A programmable switch is used to build the user-defined topology, including passing traffic through nodes dedicated to emulation. Other nodes are available for running user experiments.

In PlanetLab, the use of a centralized emulator is

not possible because there are no controlled devices on the path between nodes and the rest of the network. In previous work [17], we did propose and implement a PlanetLab extension that added external devices in charge of emulation. These devices, called Dummynet-Boxes, were placed as shown in Figure 3 on the path between nodes and the rest of the network, and were running a standalone version of FreeBSD, equipped with Dummynet, and configurable under user control to emulate the desired network features.

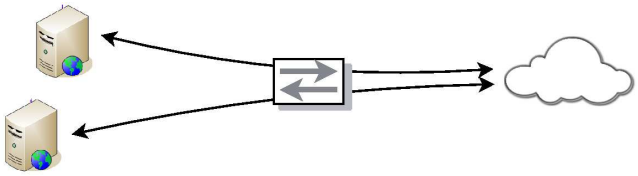

Figure 3: Emulation using Dummynet in an external device.

While available now as part of the OneLab testbed, the solution based on external emulation boxes has a disadvantage in terms of deployment. Users are reluctant to add new devices to their network, so the availability of this feature is limited.

In response to this problem, we opted for a different approach, namely implementing emulation directly within the nodes. The issue now becomes how to implement the desired functionality in an effective way.

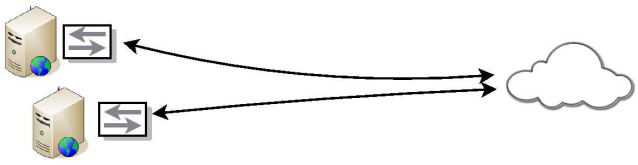

Figure 4: In-node emulation

One possibility was to create a new, custom emulator to be installed on the nodes. While doable in principle, this option was immediately dismissed given that multiple emulation software already exist and could be used without having do create a new one from scratch.

Given that PlanetLab nodes run Linux kernels, a second option was to use one of the emulation packages available on that platform: among them, there are *tc* [5] and NISTnet [19]

tc is already available on PlanetLab nodes. However, it is designed as a traffic shaper and link scheduler, so its native emulation capabilities are not an exact match for our requirements: in particular, *tc* cannot model propagation delays and reordering, features that can be added with the use of the netem [21] package.

At least two reasons discouraged us from using *tc*. First, the need to use an additional module (*netem*) to implement, and only in a partial way, the desired functionality, puts *tc* in no better position than any other solution. Second, *tc* is already used on PlanetLab nodes to control the traffic generated by the various users. Using it for emulation as well would require a lot of effort to avoid undesired interactions between the emulator and the traffic control policies, which in turn would make integration and deployment harder.

The NISTnet alternative was dismissed mostly because of our lack of experience with the tool. NISTnet is not available as a standard component in PlanetLab, so again we should rely on a third party module, and our learning curve would have been somewhat steep.

On the contrary, we do have significant experience with the Dummynet emulator, which has most of the functionalities we need, and is very easy for us to extend as we do maintain the code. The fact that the tool is not natively available on Linux is not a major drawback because the porting effort would be relatively small, and well compensated by the advantages, which include the following:

- our previous emulation solution [17] is based on Dummynet, so we can reuse the configuration tools already developed for it;
- researchers are also familiar with Dummynet and its features, because of its use on Emulab and FreeBSD. As a consequence, it will be easy for most of them to make an effective use of this new facility;
- a port of Dummynet to Linux was often requested to us by researchers who use this platform for their work, and making a port would also address this request;
- Dummynet already has a large number of features, combined with a flexible packet classifier. Also, we are planning several extensions to the tool, which would be harder to integrate in a system such as *tc* or NISTnet that we do not maintain directly;
- Dummynet is not used by other components in the testbed, and this removes the risk of interferences that we would have if we decided to rely on an already used component.

As a result, we decided to provide in-node emulation using a Linux port of the Dummynet emulator.

3. PLANETLAB

In order to describe how we added emulation to PlanetLab, it is useful to spend a few words on the architecture of the testbed.

PlanetLab is an initiative of a group of researchers interested in planetary-scale network services, where participating institutions contribute computing and network resources to build a distributed testbed. This plat-

form, depicted in Figure 1, is made of two types of components: one central controller, called PLC (PlanetLab Central), and several computing *nodes* which is where users can run their experiments.

3.1 The Planetlab Central (PLC)

The PLC is the core of the system. It runs the testbed management software and acts as a server for nodes and users. It contains a database with all relevant information on nodes and users of the system, accessible through a web interface or an XML-RPC API. It also contains a file server, used by the nodes in the system to download their initial software, additional packages, and software updates.

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. During regular operation, nodes periodically contact the PLC to fetch software updates, collect information on users allowed to access the platform, possibly report usage and other statistics.

3.2 Slices, users, slivers

The PLC gives participating organizations the right to create one or more *slices*, which are the administrative entities used to account for resource usage. A *slice* can be accessed by one or more users, whose credentials are stored on the PLC. In turn, information on existing slices and users is stored in the database on the PLC, and made available to nodes, which can use it to perform access control.

Using the web interface or the XML-RPC API, users can *instantiate* a slice on one or more nodes of the testbed. Instantiating a slice on a node means creating an *sliver* on the given node, i.e. an account and a set of resources accessible to all users belonging to the slice. The sliver is implemented using a "virtual server" (see next Section) which is the container used to confine operations performed by the sliver. Users can log into all nodes where their slice is instantiated.

3.3 Node and sliver management

Users connect to the nodes and run their experiments in a virtualized environment. The virtualization provides resources isolation between the slivers, and gives users the illusion of a node with dedicated resources. All this is done using two components, namely the Linux-Vservers [6] system and the VNET [15] system.

The Linux-Vservers is a virtualization system that provides a private filesystem namespace to each sliver, while still allowing all slices to see the full set of devices available on the node. Each sliver runs within a dedicated *vserver context* where it has limited root permissions, meaning that it can run a subset of the system calls. Operations that require real root access (i.e. must run in the so-called *root context*), are controlled through the vsys service described in the next Section.

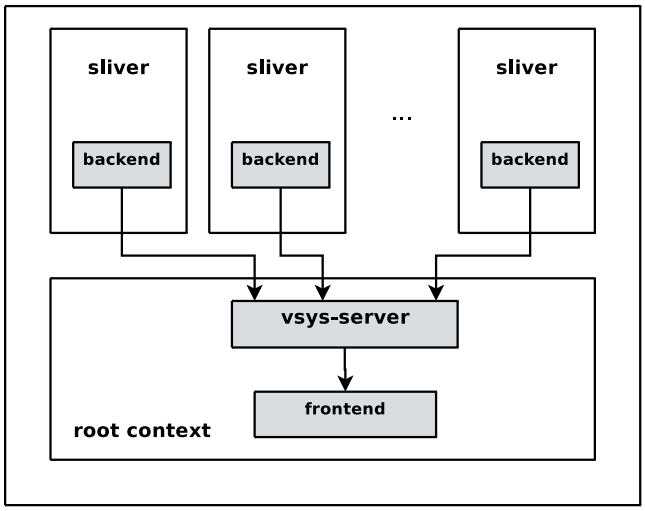

Figure 5: The structure of a PlanetLab node, and the interaction between slivers and root context.

The VNET system implements traffic isolation between slivers. It ensures that the first sliver that "claims" a local TCP or UDP port (by calling `bind()` on it) becomes the owner of the port, and will be the only one allowed to send or receive data on that local port. This feature is necessary to prevent interference between experiments that try to run servers on the same port.

Related to virtualization, nodes also use the *tc* traffic controller to limit the amount of traffic generated by each sliver so that none of them is able to monopolize the communication link.

3.4 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. The service works by creating one or more file descriptors accessible on the sliver that communicate with backend programs running in the *root context* and, as a consequence, able to run any command with no restrictions. The system lets the administrator specify which backends are available to each sliver, and also passes the identity of the sliver to the backend, so that specific access policies can be implemented.

4. DUMMYNET

The second component of our system is the Dummynet network emulator [18], developed under FreeBSD several years ago [23], and later imported

into other BSD-derived operating systems, including Mac OS X.

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.

4.1 The emulator

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

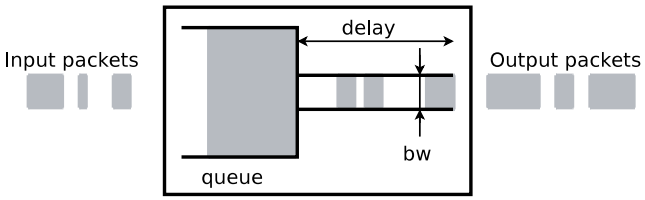

Figure 6: A Dummynet pipe.

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.

A second object implemented by the emulator is called *queue*, and it models just a single FIFO queue (the left part of a *pipe*). Multiple queues can be connected to the same communication channel (the right part of a *pipe*), and queues are scheduled for service according to a specific link scheduling algorithm (at the moment WF^{2}Q+). This part is actually very important because many MAC protocols can be modeled as schedulers, and this will make the system easily extensible.

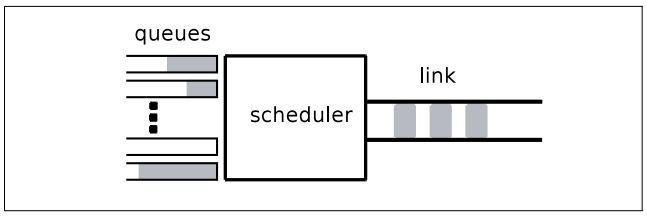

Figure 7: Dummynet queues and their use for scheduling or emulating MAC protocols.

Some emulators [21, 19] provide features to introduce specific loss or packet reordering patterns. In Dummynet, we use a different approach based on the following reasoning. In a real network, losses and reordering normally occur as a result of some specific traffic pattern (e.g. causing queue overflow), network configuration (e.g. routing flaps), or link conditions (e.g. poor SNR or excessive conflicts on a shared link). As a consequence, we try to emulate the mechanism that are the root cause of the phenomenon, and rely on users to drive the system so that the actual loss will occur as a result. This approach is especially important when testing applications that respond to losses by changing their traffic generation: in such cases, a model that generates losses irrespective of the actual traffic is likely to give inaccurate results.

4.2 The packet classifier

Dummynet works in close cooperation with a programmable packet classifier called *ipfw*, that lets the user intercept selected traffic in various points of the protocol stack, and direct it to a *pipe* (or a *queue*), as shown in Figure 8. *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. 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. When using Dummynet, the typical action is to send the packet to a pipe or queue, which will in turn emulate the behaviour of the link, delaying or dropping the packet as appropriate. After the emulation, non-dropped packets are sent back into the network stack for their regular processing. By properly programming the system, traffic can be passed through multiple different pipes, thus permitting the emulation of moderately complex network topologies.

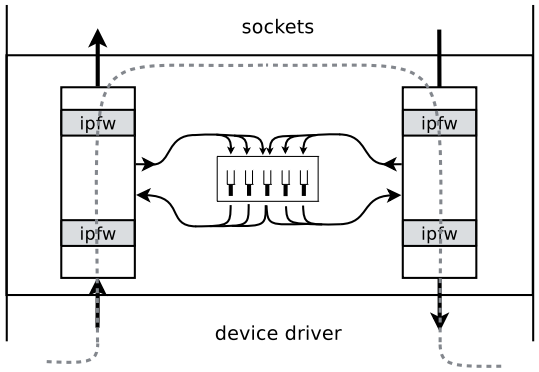

Figure 8: The flow of packets through network stack, packet classifier and pipes.

4.3 User interface

Users interact with Dummynet by deciding which traffic should be intercepted, and to which pipes it

should go. The configuration of the pipes (in terms of bandwidth, delay and so on) can be set or modified at runtime as needed.

Setting up a pipe, and passing traffic to it, is extremely simple, as in the following example:

```
# Configure dummynet:
# set bandwidth and delay of the emulated links
ipfw pipe 5 config bw 4Mbit/s delay 7ms
ipfw pipe 8 config bw 1Mbit/s delay 10ms
# Configure ipfw:
# pass selected traffic through the emulator
ipfw add 120 pipe 5 out dst-ip 10.2.0.0/24
ipfw add 130 pipe 8 out dst-ip 10.1.1.0/24
```

Here, we define two pipes with different features, and pass to each of them outgoing traffic for two different subnets.

The configuration of pipes and queues can be changed at runtime without disrupting the operation of the network. Similarly, *ipfw* rules can be added or deleted at runtime to modify the configuration of the system, and without disruption on traffic not affected by those rules. As a consequence, with a proper configuration of the classifier (e.g., by reserving certain sets of ports or addresses to each user), multiple users of a system can share the same instance of Dummynet without interfering with each other. This feature is used in this work to let different PlanetLab users share the emulator.

Multiple rules can send packets to the same pipe, which means that the user can create arbitrary aggregations of traffic sharing the same emulated link. This is important when, e.g., one wants to gradually add interfering traffic to an application under test, while passing other traffic (such as a control connection, DNS requests, remote disk accesses and so on) through a different pipe to avoid interference.

5. THE LINUX PORT

As mentioned before, a Linux port of Dummynet is interesting even outside the scope of this project, because of the popularity of the Linux platform. Also, a Linux version is also interesting because it lets people use Dummynet on embedded devices running OpenWrt [8], which is more and more used in various research prototypes as well as actual deployments.

Porting code that runs in user space is generally straightforward, and this case was no different. All we had to do was provide a replacement for some library functions (`strlcpy()`, `strtonum()`, `sysctlbyname()`) that were not present in Linux, and wrappers or renaming macros for some other functions (e.g. `heapsort()` or `setprogname()`).

Adapting a kernel subsystem to different operating systems is instead a lot more challenging, because of the lack of cross-platform standards in terms of programming interfaces (APIs), headers, kernel services, and even naming conventions. Having done 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 original version), and making it easier to repeat the work when changes are made in the base version of the code.

Overall, this particular port involved the following steps:

- adapting headers. This was done by creating a tree of additional "system headers", meant to add missing headers, or replace Linux headers that were in conflict with the FreeBSD ones, or were missing some required items;
- add small wrapper macros to disable or redefine certain identifiers. As an example, redefine the macros `DEFINE_SPINLOCK, LOG_SECURITY` and `LOG_NOTICE`;
- fix certain areas of code that made platform-specific assumptions. As an example, make explicit calls to to 64-bit division routines, provide a unique path for dropping packets, do not always put ip_len and ip_off fields of the IP header in network order. Such changes should be integrated in the original code to improve portability;
- disable parts of the code that were not interesting for this project and were too system-specific. As an example, we disabled approximately 10 matching options (out of 48) and 5 actions (out of 19) that were referring to FreeBSD-specific subsystems;
- add glue code to recreate the FreeBSD kernel APIs on top of the Linux ones. This will be described in more detail in the following Sections.

The most time-consuming parts of the work were related to the design, and specifically i) find the best location to put header information, ii) decide where to apply the "within reason" principle and when changes to the original source were acceptable, and iii) identify a good replacement for the kernel subsystems used by Dummynet.

5.1 Hooking into the 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, as in Figure 8), and to reinject packets back into the stack after some delay.

In the original version of Dummynet, packets were intercepted by modifying some functions

in the packets' path (`ip_input()`, `ip_output()`, `ether_input()`, `ether_output()`), making them call the classifier. The latter would return back those packets that were not dropped or delayed. Reinjection of delayed traffic was done by explicitly calling the same functions as above, with packets marked in a way to avoid further reinjection in the classifier.

Kernel modifications are not necessary anymore. Many operating systems now support the insertion of generic *packet filter* functions on the packets' path. This mechanism is called *pfil* on FreeBSD, and *netfilter* on Linux.

When a packet filter function is registered, it gets called on each packet traversing the network stack, and it should tag the packet with an indication of its fate, which is normally *PASS* (let the packet progress through its destination) or *DROP* (drop the packet[1]).

The *netfilter* system is particularly interesting for our purposes because it supports a *QUEUE* tag, that causes such packets to be diverted from the regular path and passed to a *queue handler* function, together with appropriate metadata. The queue handler can delay or manipulate the packet at will, and must eventually call the function `nf_reinject()` which causes the packet to be finally go back into the stack after the point of intercept.

In the current Linux version of Dummynet, we register two netfilter functions on the `PRE_ROUTING` and `POST_ROUTING` hooks, and one queue handler function. The netfilter functions unconditionally[2] tag all packets as *QUEUE*, and leave to the queue handler the task to run the packet classifier and possibly the emulator, and reinject packets into the stack after a suitable delay.

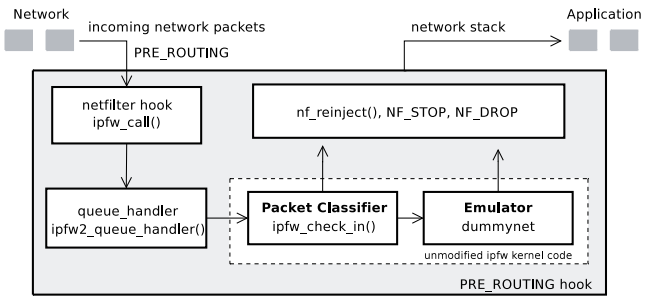

Figure 9: Netfilter hooks in dummynet

Figure 9 shows how the mechanism works in case of an incoming packet. In this example, the packet goes to the netfilter function `ipfw_call()` which unconditionally tags it as QUEUE. Then the packet goes to the netfilter queue subsystem, which in turn calls our queue handler, `ipfw2_queue_handler()`. The function is in charge of calling the packet classifier, and depending on the outcome, either pass the packet immediately to `nf_reinject()` or call `dummynet_io()`, which is the entry point of the emulator. Eventually, the packet is delivered back to the network stack calling `nf_reinject()`.

[1]The *DROP* tag can also be used when the filter wants to keep the packet for itself, and will reinject the packet in the stack at a later time.

[2]This will be optimized in future versions, avoiding unnecessary calls to the queueing subsystem.

5.2 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 (or a set of function arguments) is used to store metadata such as packet length, a pointer to the data, direction, related interfaces, flags.

In FreeBSD and other BSD-derived systems, metadata are stored in a structure called `mbuf`, which holds all the above information, including a pointer to an internal (to the `mbuf`) or external block of memory containing the actual packet data. Data is in turn stored in a linked list of buffers. In Linux, there is a similar arrangement except that the container for metadata is called `sk_buff`.

In our port, whenever we receive a packet to process, we first create a stripped-down `mbuf` initialized with relevant fields fetched from the `sk_buff`. 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 external `mbuf` descriptor is simply destroyed, and the packet is reinjected completely unmodified. Note that some emulation features (such as error injection) may involve modifying the data areas, in which case one has to be careful because the `mbuf`/`sk_buff` content is often shared by multiple parts of the kernel. Also note that one may need to modify the metadata as well. As an example, on Linux, when a packet is received and can be associated to an existing socket, the socket is timestamped with the reception time. When we delay incoming packets, we must also update the timestamp in the socket, or program such as `ping`, which use the socket's timestamp, would return invalid results.

5.3 Other system services

The adaptation of other system services has been relatively straightforward, and normally provided by writing wrappers around the Linux functions so that we could export a FreeBSD-compatible API.

In the original version of Dummynet, locking uses both mutexes and rwlocks. In the Linux port, we have mapped both types of locks to `spin_lock_bh()`. Optimizing the choice of the locking mechanism to improve concurrency has been postponed, because the original

code is undergoing similar changes.

The memory allocator just required some conversion macros to map FreeBSD calls (`malloc(), free()`) into the equivalent Linux functions (`kmalloc(), kfree()`). FreeBSD uses an optimized allocator called "UMA" for objects of fixed size; in this port, we simply remapped the `uma_*` calls into `malloc()/free()` calls. Again, the use of an optimized allocator (also available on Linux) will be done in future versions of the code.

Timer support on FreeBSD relies on a periodic system timer, running with a configurable rate, which is used to trigger pending timeouts. Kernel subsystems can schedule the invocation of a function after a certain timeout using the "callout" system. Linux offers a similar functionality, with only a different interface and naming. The conversion is again done by means of simple wrapper functions.

The typical value for the periodic timer is 1 KHz on FreeBSD, resulting in 1 ms accuracy of the timing. On some Linux versions, the default timer runs at 250 Hz, giving 4 ms accuracy. The resolution can be changed easily, but going below 100 μs is problematic because of the extra system overhead. Also, there are certain uninterruptible blocks of code (such as large memory-to-memory copies, or critical sections) which may well consume tens of microseconds, so even with higher timer resolution we cannot increase the accuracy by much.

The infrastructure for managing loadable kernel modules relies on a slightly convoluted set of macros, functions and compiler/linker support to create appropriate dependency lists at compile time, and enforce them at runtime. Also in this case we had to adapt the FreeBSD macros mapping them to Linux ones. In this case the conversion was not totally straightforward as it relies on a lot of preprocessor tricks, but the details are not particularly interesting.

5.4 Linux port summary

As a result of this work, we have created a Linux port of Dummynet which is made of a single kernel module, `ipfw_mod.ko`, containing both the packet classifier and the emulation module, and a control program, `/sbin/ipfw`, which provides the user interface. The two parts communicate through the `sockopt` mechanism, whereas packet are passed between the kernel and the module using the netfilter API. The current code has been tested on a wide range of Linux versions, including the 2.4 family (used in some OpenWrt distributions) and several versions of the 2.6 family (2.6.22 to 2.6.28). The code is available at
`http://info.iet.unipi.it/~luigi/dummynet/` .

6. ADDING EMULATION TO PLANETLAB NODES

Putting together all the components described so far, the architecture of our in-node emulator for PlanetLab becomes relatively simple.

At the lowest level, we use a Dummynet kernel module and its related control program `/sbin/ipfw` running in the root context to do the emulation. On top of this, we use the vsys service to control how slivers access the emulator. A vsys frontend, `netconfig`, is used to "claim" a TCP or UDP port and configure emulation on it. A vsys backend, `ipfw-be`, does the parameter checking and possibly configures Dummynet or updates the existing configuration.

No kernel or other system modifications are required to install the above components, so existing PlanetLab nodes can be updated by simply downloading and installing the required packages (one for the root context, one for the sliver) from the PLC. This will make the integration of emulation as easy and fast as any other software update.

6.1 Isolation between users

To avoid that users of the different slivers interfere with each other in configuring the emulator, we adopted a strategy similar to the one implemented by the VNET system described in Section 3.3: a sliver "claims" a network resource (in this case, a TCP or UDP port) as its own, and after that no other slivers are allowed to configure emulation on that port until it is released or the claim expires.

The claim is made the first time a sliver runs the frontend program to configure emulation on a given port:

```
./netconfig -p <port number> <parameters>
```

This results in the backend program being run, which in turn checks that the desired port is still available (or already in use by the same sliver), and then performs the desired configuration. The actual commands run by the backend depend on the parameters specified; the following is an example output corresponding to the use of port 5678 (here the port number is also used as the base for rule number and pipe number):

```
ipfw pipe 15678 config <parameters>
ipfw pipe 25678 config <parameters>
ipfw delete 5678
ipfw add 5678 pipe 15678 src-ip $ME src-port 5678
      // timeout sliver
ipfw add 5678 pipe 25678 dst-ip $ME dst-port 5678
```

As we can see, we configure two rules, one per direction, and one of them also stores, in the comment field, the identity of the sliver who claimed the port, and the timeout value for the rule. The sliver name is used to check that subsequent updates come from the same sliver, whereas the timeout is used by a separate

daemon to expire rules (and pipes) that are not in use anymore.

Modifications of the configuration, as well as release of the port(s) reserved by a sliver, can be made by simply reinvoking the `netconfig` program, with the same port number, and suitable parameters. Note that `netconfig` can be also invoked from within the code that is part of the experiment, so that variable network conditions can be obtained without direct user intervention.

6.2 Resource management and logging

It is useful, from an administrative point of view, and also for management purposes, to keep track of users of the emulation service and their requests. In fact, emulation consumes resources (ports, but also buffers for packets that are staging in the emulator) and we do not want users to abuse the system. Also, the total throughput of the system is limited by the available bandwidth on the physical link, so we should keep track of concurrent requests and at least warn users if they are making requests that cannot be fulfilled because they exceed available resources.

In our system, logging is done by calling external programs both in the vsys backend `ipfw-be`, and in the daemon in charge of cleaning up expired configurations.

6.3 Building and installing

The extension described here has been developed using a local instance of the MyPLC [7] software. The *build system* that is part of MyPLC allows an easy integration of new software into the platform.

Emulation extension is made of two software packages (RPMs):

- one for the root context emulator, which includes the `ipfw_mod.ko`, the `/sbin/ipfw` control program, and the periodic cleanup program. It must be installed in the root context;
- one for the frontend, to be installed in the vserver context by all slivers who want to use the extension;

The vsys backend is packaged together with other vsys backends in the vsys-script package.

7. EXPERIMENTAL RESULTS

Dealing with an emulation system, the performance evaluation refers mostly to finding the limits of applicability of the system itself. In particular, we want to find out:

- how much the introduction of the additional kernel module impacts the performance of the node;
- what is the accuracy of the emulation system;
- how frequently we can tolerate reconfigurations of the emulator.

To measure the overhead introduced by Dummynet, and the accuracy of the emulator, we have run a number of tests where the machine under test receives ping requests from an external machine, connected through a full-duplex, 100 Mbit ethernet switch. We measure ping response times machine with and without Dummynet, and with different load conditions. We use ping because ICMP requests are processed entirely within the kernel, so there is no unwanted interaction with user space process scheduling. Absolute performance numbers are not that important, but we do want to see if there are statistically significant deviations from the behaviour without the emulator (our *baseline*).

In our tests we used the following load conditions:

- IDLE. Completely idle system, no extra process is running except the basic system services;
- KERNEL. A number of process are accessing devices and memory filesystems, continuously issuing system calls that cause heavy kernel load;
- USER. Several processes run the loop `extern volatile a; for(;;) a++;`, consuming the full CPU available, and accessing the memory bus, in user space;

While not exhaustive, these conditions try to reproduce a reasonable subset of the load conditions that a node can experience.

In all conditions, we measure the ping response times without the Dummynet module, and with the module loaded using various configurations. The results (both average and standard deviations) are presented in Table 1.

	IDLE		KERNEL		USER	
	avg	sd	avg	sd	avg	sd
without Dummynet						
default	82.0	5.93	103.0	8.75	76.9	4.16
with Dummynet						
IPFW-1	79.1	3.91	107.5	6.25	77.1	5.18
IPFW-100	98.5	15.1	119.6	5.74	97.5	14.4
Dummynet and two 10ms pipes (times in ms)						
HZ=1000	19.63	0.310	19.71	0.319	19.62	0.260
HZ=250	21.99	1.148	22.05	1.152	22.06	1.166

Table 1: Average and standard deviation of response times in the various tests. All times are in microseconds except for the bottom two rows, where they are in milliseconds. Please refer to Section 7.1 for an explanation of the apparently surprising results with and without load.

7.1 Dummynet overhead

The row labeled "default" represent the baseline case. Note that the USER column, in this and other rows,

reports faster response times than the IDLE column. This is not a mistake, and while apparently surprising, it has a very clear explanation which is useful to give, also to show how careful one should be when doing certain measurements.

On many operating systems, the CPU enters a power saving state when there is no work to do. The exit from this power saving state, generally triggered by an interrupt, requires a significant amount of time. We are precisely in this situation during our IDLE tests, where requests are infrequent enough to cause this phenomenon. In the USER case, the CPU is fully used, and no power saving mode is entered, reducing response times because our workload did not involve system calls or other uninterruptible tasks.

The same explanation applies when comparing the baseline with the case when Dummynet is loaded but it only has one rule installed (IPFW-1). Dummynet causes the kernel to run a function at every timer tick. This extra load is enough to reduce the use of power saving states, and explains why the case IPFW-1/IDLE seems faster than the case default/IDLE.

A reasonable estimate of the Dummynet overhead can be given by the KERNEL column. In this case, the CPU is fully busy with kernel tasks, some of which are not interruptible. This causes an increase of the response time compared to the IDLE case, and also a larger difference between the "default" and "IPFW-1" cases, approximately 4.5 μs on our hardware. This difference can be reasonably charged to the setup of the two invocation of the packet classifier (done through the netfilter hooks and the queue handler), and is an estimate of the minimum overhead caused by the emulator.

The cost of the classifier processing also depends on the number of rules in the configuration. The row IPFW-100 presents the response times with 100 matching rules, which is representative of a large classifier configuration. In this case we see that 200 matches (100 in each direction) cause an additional 20 μs delay in the response, or 100 ns per rule. Figure 10 shows the distribution of response times with 100 rules. As we see, in this case not only the average times increase, but also the variability becomes large.

7.2 Emulator accuracy

As discussed in Section 5.3, the accuracy of our emulator depends on the resolution of the timer tick. The value used on PlanetLab nodes gives 1 ms granularity. This roughly corresponds to the duration of a maximum-size ethernet frame at 12 Mbit/s, or to a minimum-size frame at 512 Kbit/s. The granularity affects the error on the timing of individual events (packet transmissions or receptions), but does not accumulate over multiple events, so we can still simulate links at higher rates. Table 1 also shows two experiments with

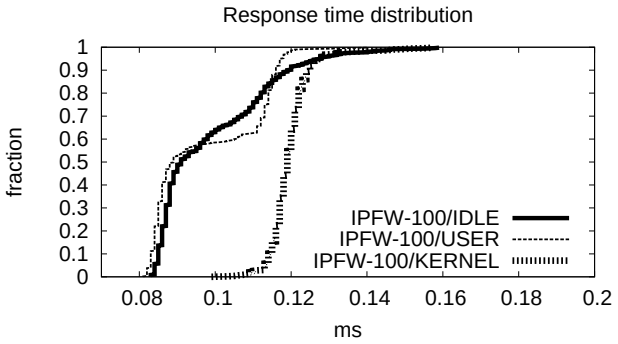

Figure 10: Distribution of response times with 100 matching rules and variable load.

a 10 ms pipe in each direction, on a kernel with 1 ms timer. The same experiment is repeated on a kernel with the timer running with a 4 ms granularity. In both cases, the timer resolution reflects directly into the standard deviation of the test.

The timing accuracy on individual events is also affected by the jitter introduced by the kernel in responding to interrupts. We can estimate a lower bound for this value comparing the the IDLE (or USER) and KERNEL columns. In our experiments, we see as much as 30 μs of difference in the response times, which tells us that it would be unreasonable to increase the timer resolution much beyond the 200 μs range.

7.3 Reconfiguration cost

Reconfiguring the emulator involves calling the frontend, which in turn uses the vsys service to call the backend program. This results in a few calls to `/sbin/ipfw` to update pipes and classifier configuration.

A single invocation of `/sbin/ipfw` takes less than 10 ms to run even on a very slow system (a PlanetLab node running in qemu). The entire process, even in the current, non-optimized form (both the backend and the frontend are implemented by shell scripts) can run in 150 ms. Optimizing the two programs, and running the code on a real machine should easily allow at least 10 reconfigurations per second, which is well beyond what a normal user should expect to do.

8. RELATED WORK

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

8.1 Network testbeds

We have already introduced PlanetLab in Section 3, and discussed its features through this paper.

EmuLab is another popular network testbed, also publicly available to researchers, but differing from PlanetLab in several aspects. EmuLab is a public facility, with nodes mostly concentrated in a single location and interconnected through a programmable switch that is used to create user-specified topologies. EmuLab's strength is the availability of a wide range of experimental environments such emulation, simulation, real wireless testbed involving radio and sensors network. Recent work [22] add to the platform the so called "virtualized emulation", where different virtualization techniques are introduced in order to best exploit the physical resources. In EmuLab, each experiment requires to define a topology, which can be done using the Ns-2 [10] 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.

EmuLab provides a large building with fixed and mobile nodes. Some devices are equipped with 802.11 a/b/g wifi interfaces. Mobile nodes have wireless card attached to robots able to moving around a small area in the lab and can be controlled and programmed by the user. An additional feature of EmuLab is the hardware support for Universal Software Radio Peripheral [13] (USRP) devices, (and related software GNU Radio [4]) connected to some EmuLab nodes. The integration of the Ns-2 [10] in EmuLab makes simulation capabilities available to the platform.

ORBIT [9] (Open Access Radio Grid Testbed) is a testbed based on a laboratory equipped with a large indoor radio grid emulator 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 three cards, two wireless and a wired one. The first is used to perform experiments, the latter is usually used as a control channel. ORBIT provides a useful environment for wireless application testing, where wireless capabilities such the channel, transmission power, transmission rate and other high level parameters can be configured. In this way it is possible to change the node behaviour, making possible to configure the nodes acting like access points, pure wireless nodes or any other kind of device. This grid of nodes provides a very flexible testing environment for wireless research.

VINI 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.

8.2 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, and software-based solutions that run in standalone devices or within standard operating systems.

The latter category includes some very popular solutions, already mentioned in this paper, and variants thereof. Again, we have described Dummynet in depth in this paper. A tool with similar features is NISTnet [19], which runs on Linux and also supports the emulation of multiple links with programmable bandwidth and features. Another solution that is sometime used under Linux is the combination of *tc* [5] and *netem* [21], where the former does the 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.

Several researchers have extended Dummynet to provide additional features such as programmable packet dropping. Other works, such as Modelnet [24], have used modified versions of Dummynet as the basis for a larger emulation system.

The basic emulation features of the above packages can be used to build complex topologies, by using multiple physical or virtual instances of the emulator, and interconnecting them with the nodes in the network through a programmable switch (again, a real one or a virtual one).

Dummynet makes this possible through the reinjection of traffic in pipes multiple times. The switching in this case occurs by an appropriate programming of the packet classifier. Imunes [26] is a system based on FreeBSD which supports multiple, virtual network stacks within a single instance of the operating system. Each virtual stack can implement a node in the emulated topology, and connect to other nodes through its own instance of Dummynet. The obvious extension of this concept is to run multiple emulator instances within virtual machines (Xen, VMWare, VirtualBox, Qemu) and connect them as required.

Emulation features are also present in network simulators such as Ns-2 [10] and Ns-3 [11], which can drive

the simulator with live traffic, and interact in this way with real traffic sources and links.

9. CONCLUSIONS

We have presented how the Dummynet emulator has been ported to Linux, and how the emulator has been used to add emulation capabilities to PlanetLab nodes. The port of Dummynet to Linux makes the emulator available to a large set of users who rely on Linux as their platform of choice. It also makes the emulator available on OpenWrt, which runs on a large variety of low cost devices.

The PlanetLab extension presented here is also useful for researchers, because it complements the features of the testbed with a useful mechanism to achieve more reproducible experiments. Our measurements show that we can run the emulator in a node without a significant impact on performance, and with reasonable accuracy at least at low or medium data rates. We expect significant improvements in both performance and accuracy as we optimize the port to use more efficient primitives available on Linux.

As of this writing, the extension presented here are being integrated in the OneLab version of PlanetLab, and work is in progress for the inclusion into the main testbed. PlanetLab nodes will be able to exploit the platform by simply installing a couple of RPM packages, without the need for a full update.

The system described in this paper is under active development. In addition to some performance optimization to the emulator, and more powerful configuration options, we are developing some extensions to Dummynet which include better emulation of wireless MAC, as documented in [18]. These will become readily available to PlanetLab users as they are integrated in the system.

10. REFERENCES

[1] EmuLab - total network testbed. http://www.emulab.net/.

[2] FIREWORKS. http://www.ict-fireworks.eu/.

[3] GENI: Exploring Networks of the Future. http://www.geni.net/.

[4] GNU Radio. http://gnuradio.org.

[5] Linux Advanced Routing & Traffic Control. http://lartc.org/.

[6] Linux Vservers. http://linux-vserver.org.

[7] MyPLC. http://www.planet-lab.org/doc/myplc.

[8] OpenWrt. http://openwrt.org/.

[9] Orbit. http://www.orbit-lab.org/.

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

[11] The NS-3 Network Simulator. http://www.nsnam.org/.

[12] The Onelab2 Project. http://www.onelab.eu/.

[13] USRP: Universal Software Radio Peripheral. http://www.ettus.com.

[14] VINI, A virtual network infrastructure. http://www.vini-veritas.net/.

[15] VNET: PlanetLab Virtualized Network Access. http://www.planet-lab.org/doc/vnet.

[16] J. Bicket, D. Aguayo, S. Biswas, and R. Morris. Architecture and evaluation of an unplanned 802.11 b mesh network. In *Proc. of the 11th Int. Conference on Mobile computing and networking*, pages 31–42. ACM New York, NY, USA, 2005.

[17] M. Carbone, G. Cecchetti, L. Rizzo, F. Checconi, and A. Ruscelli. Wireless link emulation in OneLab. *2nd International Workshop on Real Overlays And Distributed Systems (ROADS) Warsaw (Poland)*, July 2007.

[18] M. Carbone and L. Rizzo. Dummynet revisited. Technical Report, May 2009. Available at http://info.iet.unipi.it/∼luigi/dummynet/dummynet09.pdf.

[19] M. Carson and D. Santay. Nist net: a linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, 2003.

[20] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[21] S. Hemminger. Network emulation with NetEm. In *Linux Conf Au*, 2005.

[22] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX 2008 Annual Technical Conference*, pages 113–128, 2008.

[23] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, 1997.

[24] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36:271–284, 2002.

[25] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: a wireless sensor network testbed. In *IPSN '05: Proc. of the 4th International Symposium on Information processing in sensor networks*. IEEE Press, 2005.

[26] M. Zec and M. Mikuc. Operating system support for integrated network emulation in imunes. In *Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS), Boston, MA*, 2004.