NEPI: An experiment plane for experimentation test beds

Martin Ferrari

MASTER II IFI (Informatique: Fondements et Ingénierie)
Parcours UBINET — Ubiquitous Networking and Computing

— Rapport de stage de fin d'études 2010 —

NEPI: an experiment plane for experimentation test beds

MARTÍN FERRARI

Maître de stage: Mathieu LACAGE
Tuteur enseignant: Fabrice HUET

Projet PLANETE — INRIA Sophia Antipolis

April 1, 2010 - September 30, 2010

Contents

CHAPTER 1

Introduction

Because the day to day work of the researchers of the Planete Team involve the design and the validation of new protocols and applications, they have considerable interest in tools that can improve the quality and the efficiency of their daily work flow. For a few years, now, they have invested a lot of efforts in building next-generation simulation tools [1] and wireless [2] or internet-scale [3] experimentation testbeds but have so far been unable to make it easy to use these tools together.

In a previous internship, we built the first prototype of NEPI[4], an experimentation description and control library that intends to simplify the description and the deployment of network experiments that involve these different experimentation tools. In this internship, we joined again the engineering team responsible for the development of the simulation tools used by Planete and we focused on the integration of a new backend for NEPIthat is based on Linux network namespace containers and that intends to demonstrate the feasibility and the usefulness of using different experimentation tools together.

In chapter 2, we first provide some context for this work and we attempt to explain the rationale for the objectives we pursued in this internship. Chapter 3 first presents the performance and scalability validation that was necessary to validate the domain of applicability of the Linux virtualisation technology the Planete team chose to use. Then, we highlight the most challenging problems we had to tackle to simplify the creation and the management of Linux containers and finally present our ongoing work to add support for this new test bed to NEPI.

CHAPTER 2

Motivation

2.1 Network experimentation for research

One of the objectives of network research is to design new protocols or modify existing systems to add new user-visible functionality or simply improve the performance of these networks. One of the major problems that network researchers must deal with is the high heterogeneity of existing networks and the potential interactions between the many existing layers of the protocol stack and the new mechanisms being designed.

To make it easy to deploy new protocols in production systems without impacting existing applications and without using undue amounts of resources in the network, these new protocols must thus be validated before they are deployed in the wild.

Fuelled by this need, experimentation tools of different characteristics, targeting different kinds of scenarios, have appeared in the last years. Nowadays, the evaluation of a new protocol or application can make use of multiple experimentation environments: field testing, simulators, test beds, or complex combinations of those.

2.1.1 Field experiments

One of the most natural way to test a new protocol before deploying it in a production network on a large scale is to conduct a small-scale field experiment: the low cost of hardware makes this a viable option in many cases. By following this approach, realistic and detailed data about the idea being tested is assured, but it suffers from a number of real problems. Firstly, although the good data is there to be picked, it is very hard to instrument and gather it. Second, because the purpose of these field tests is to observe what happens when the protocol

is exposed to the natural variability of the real world, these experiments are inherently difficult to reproduce.

2.1.2 Simulation experiments

Another approach to experiment with a new protocol is to build instead a simulation model of this protocol and of the target deployment environment. Network simulations usually focus on simulating the complex interactions between packet events, for example: the MAC layer of device X receives a packet, forwards it up to the upper layers, etc. These models are typically implemented in discrete event-driven simulators that keep track of the set of modelled events and execute them sequentially until no more events are left.

The major upside of this simulation mechanism is that it is perfectly reproducible and that it is possible to control and inspect the value of each state variable across the entire simulated system. However, the modelling process inherent to the use of these simulators introduces an extra level of abstraction that is hard to quantify: a simulation is much easier to carry out than a field experiment but it is very hard to know how realistic its results are, since those depend critically on the quality of the models employed.

2.1.3 Test beds

A common alternative to deal with the limitations inherent to simulations and field experiments is to build test beds: a test bed is a hardware and software platform that is usually very similar to a field experiment, with much improved control and reproducibility characteristics.

For example, hiring cheap students to walk around with ten or twenty laptops exchanging data through wireless cards would provide good insight on how variable the wireless medium is but it will not be easy to reproduce the results of that experiment. The corresponding simulation will be fully reproducible but will likely give very different results that do not take into account typical background radio interference. On the other hand, a test bed that sets these laptops in a room protected from external radio interference by a Faraday cage will make it possible to run the same experiment multiple times more easily, will likely give very similar results from one run to the other, and will be much more realistic than a simulation.

2.1.4 Mixed experiments

While test beds represent a great compromise between the controllability and reproducibility of a simulation and the realism of a field experiment, they still suffer from one major problem: large-scale experiments that require a lot of hardware become too expensive, making simulations the only viable alternative.

To improve the scalability of a test bed and the realism of a simulation, a rarely-pursued alternative is to set up a mixed environment that interconnects a real-time simulator and a test bed: the simulator makes it possible to conduct an experiment that scales to many more network devices than what is physically available in the test bed, while the test bed introduces more realistic and higher variance in the experimental conditions.

These mixed experiments need to be split into multiple smaller components, each one of them describing the topology and behaviour of a partition of the network, and being assigned to the test bed or the simulator. This introduces a number of problems:

- users must learn how to use each tool and its programming interfaces, to deal with separate authentication and authorization mechanisms, with the deployment, monitoring, etc.;
- it is hard to collect in a central location the description of the entire experiment together with the results of the sub-experiments that are scattered in many places;
- it becomes much harder to ensure that each partition is configured in a way that is compatible with the other partitions. For example, we need to make sure that the routing tables set up in each simulated or real network device are consistent and complete, to ensure full application-level connectivity within the entire network.

2.2 NEPI: Network Experiment Programming Interface

To deal with the difficulty of setting up and conducting mixed experiments, during a previous internship, we worked on the design and implementation of NEPI. NEPI intends to simplify the use of different network experimentation tools at the same time. To achieve this goal, we designed a unified API and object model that can be used to describe network experiments independentely from the underlying interfaces and idioms specific to each experimentation tool and that can do so without without abstracting and hiding their functionality.

NEPI is presented in the form of a Python library that manages the experiment description, its deployment on remote servers, its monitoring, and finally, the clean-up and collection of traces

Some of the design goals of NEPI were:

- to be able to run mixed or traditional experiments,
- ease of use,
- unified programming interface,

- single description for the whole experiment,
- single point of control, from which the deployment, control, and clean up takes place in an automated fashion.
- automated IP address assignment and routing table configuration.

At the core, an object model based on boxes and connectors was created. Diagrams based on this model are commonly used to design Integrated Circuit board layouts, but also are often the basic building blocks of graphical programming languages. This box metaphor is very simple, but it is powerful enough to describe arbitrarily-complex systems, especially when composite objects are used to describe hierarchical systems with varying levels of detail.[1]

To support each specific experimentation tool, Nepi requires a plug-in —called a *backend*— to provide the glue code and object definition for interfacing with the tool. Currently, the only backend available wraps the ns-3[1] simulator.

2.3 Goal of the internship

To demonstrate the feasibility and the usefulness of running mixed experiments, this internship focused on filling the gap left by the previous one; that is, create a second backend for Nepi to demonstrate that it is possible to fully automate the deployment and configuration of mixed experiments, hence paving the way to make them more widely adopted among network researchers.

2.3.1 Test bed

To do so, the Planete team chose to build a test bed platform based on a combination of multiple Linux kernel technologies (netns, netem, tbf, veth, tap, and bridging) that make it possible to spawn, on the same host system, multiple containers with independent kernel-level network stacks; to interconnect them with kernel-level emulated links, and to execute instances of any unmodified network application within each container.

netns: Netns is one of the components of the lightweight container-based Linux virtualisation technology called Linux Containers[5] (LXC). Netns refers to the ability of the Linux kernel to provide isolated network stacks to different groups of processes. This isolation comprises the visibility of network interfaces, IP addresses, routing tables, sockets, etc. In a nutshell, two processes running in different namespaces will see a completely different and independent network configuration. At the same time, they share the kernel, file system, IPC, and non-networking hardware, thus avoiding unnecessary overhead, and simplifying the sharing of information.

[1] More details about Nepi can be found in the paper[4] published last year on the topic.

veth: Originally developed for OpenVZ[6] —a project similar to LXC—, but now integrated in the main-line kernel, the *Virtual Ethernet device (veth)* is a simple but efficient mechanism for inter-namespace communication. Veths exist as pairs of coupled virtual network devices that simulate two Ethernet cards connected by a cross-over cable: what is sent on one interface is received on the other. These devices make it possible to create virtual links connecting different namespaces, as if two computers were connected by a Ethernet cable.

netem: Another recent addition to the Linux kernel is the *network emulation queueing discipline (netem)*[7], which provides the ability to emulate real-world link conditions (delay, jitter, packet loss, etc.) on any network device, including the veth device which normally works as an ideal no-delay, no-loss connection.

tbf: The *token bucket filter queueing discipline (tbf)* is the other key component to emulate link conditions, it is used to limit the bandwidth that a veth device can transmit.

bridge: The *bridge* device has been traditionally used to implement in software the forwarding of link-level frames between real physical network cards. In the context of our test bed, it is useful as a way to interconnect many veth halves as if they were connected to a network switch.

tap: The tap device creates a virtual network device which work just like an Ethernet card, but instead of sending and receiving electrical signals on a cable, it moves frames to and from user-space through a file descriptor. A user-space program can read from it to obtain its outgoing packets and write into it to enqueue a packet in its receive queue. The tap device has been used for many years to implement virtual private networks (VPN). In this project, it is the fundamental building block that allows a simulator and the netns-based test bed to communicate.

The addition to the mainline kernel of these elements has suddenly made possible[2] to run applications in separated network stacks and to connect them with virtual network links that mimic conditions of real network connections. More concisely, to create a network test bed in a single computer and kernel.

2.3.2 Tasks

The first task we were assigned was to validate experimentally the robustness and the memory and CPU efficiency of the Linux technologies used for this test bed. The second task, in case these were deemed fit for the project, called for the design and implementation of a new backend for NEPI that would be functional by the end of this internship.

[2] It was possible to do this before, and there are emulation tools that use similar concepts, but they require heavy kernel patching, or offloading to user-space, which is much less efficient

CHAPTER 3

Internship

While the high-level objectives of this internship were well-defined, many low-level implementation and design details were left undecided. In this chapter, we first report in section 3.1 on the extensive benchmarks performed to assess the viability and efficiency of the system, discuss its implementation details in section 3.2, and finally highlight some of the problems encountered along the way in section 3.3

3.1 Validation

For the system to be useful, it needs to accurately meet the specifications the user defined for the emulated links. While non-realtime simulations advance the simulation clock as calculations are completed, a realtime test bed such as the one we consider here needs to avoid missing deadlines under heavy load to ensure that the network links it models match their intended behavior. Once the processor or the memory are exhausted, the characteristics of the virtual links start to vary, showing potentially decreased throughput, innacurate link delays, and possibly undesired packet losses.

In this section, we thus focus on determining through experimentation the boundaries in terms of the number of nodes and the number of packets to process per second beyond which the test bed gives unreliable results.

3.1.1 Methodology

To estimate the realism boundaries of this test bed, we consider the case of the simple linear topology shown in figure 3.1 with saturating UDP traffic: TCP is not used to avoid the interference of its congestion control algorithms with our benchmarks. The left-most node sends saturating UDP traffic with a user-specified packet size towards the right-most node while all other nodes forward

traffic from left to right. The right-most node keeps track of the number of packets received and reports it at the end of each run of a benchmark. Each benchmark was conducted once for each combination of the parameters we were interested in, that is, the size of the packets and the number of nodes.

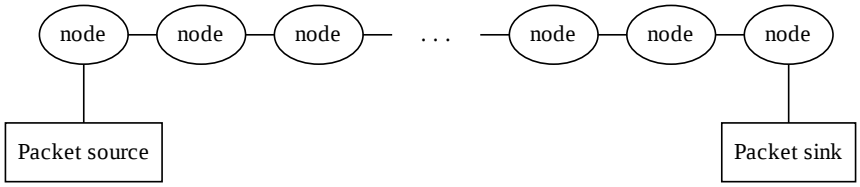

Figure 3.1: Configuration used to run the benchmarks

Due to the considerable runtime execution cost of these benchmarks, we have not yet been able to run each benchmark multiple times to average our measurements and collect confidence intervals. However, the following sections present and discuss the early results we obtained.

Because none of the existing network measurement applications (iperf, netperf[8, 9]) could really saturate our test bed links, we wrote and used instead `udp-perf` to generate and receive the application traffic. `udp-perf` is a small command-line C program that we wrote to use the Linux high resolution timer [10] API and thus be able to control accurately the transmission rate of small packets even when the target throughput is very high. `udp-perf` achieves typically much higher throughput than what a tool such as iperf[8] would report in this case because it does not use an accurate active loop that uses up cpu or an imprecise low-resolution timer API to wait between packet transmissions.

On top of this, ad-hoc scripts were used to construct the network layout and explore the parameter space, since the netns library was still not written.

3.1.2 Baseline: loopback transmission

To measure the efficiency of the Linux kernel in handling the IP stack, context switching, and memory copying to and from user space, the first batch of tests were run on the bare loopback interfaces.

This test displayed results matching our expectations, as per-packet processing costs varied in proportion to packet size (to account for memory copies) with a strong constant cost associated with context switching and packet handling in the kernel, as shown in figure 3.2.

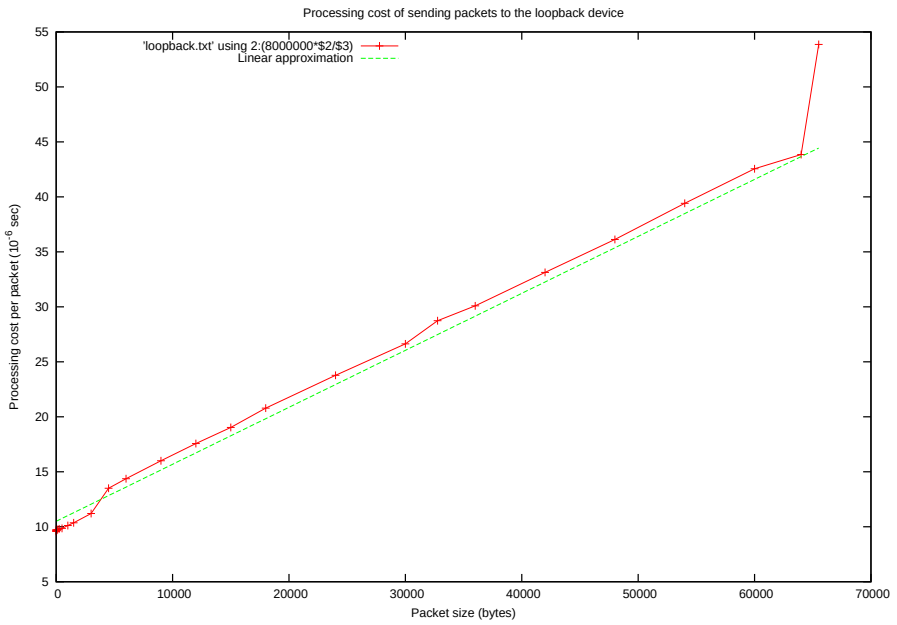

Figure 3.2: Test showing baseline costs of packet processing

3.1.3 Namespaces

Another round of tests was performed to compare the cost of forwarding packets between different namespaces, using a simple linear topology with and without bridges.

In figure 3.3, we can observe the per-packet cost growing almost linearly with respect of the number of nodes, but the change is much more significant that the proportion seen in the previous section.

This shows that both the routing code and the bridging code impose significant performance penalties. After further investigation using a system profiler, we found the routing overhead to be linked to an extra memory copy in a critical section of the routing code, caused by a suboptimal choice for the size of the SKB[1] headroom. A patch to solve this issue was created and offered to the kernel developers, but it still has not been accepted.

[1]The SKB structure is one of the most important elements in the Linux networking stack, it is a complex structure and an explanation of it is outside the scope of this report, but it is well documented elsewhere[11].

3.1.4 Comparison with ns-3

To have a reference value, the processing cost per packet was measured against an equivalent experiment ran with the ns-3 simulator. For improved fairness, the kernel was configured to run with a single processor, since the simulator can only use one.

In figure 3.3 it can be seen that the performance of the system compares similarly to that of the highly optimised ns-3, outperforming it when no link emulation is in use. It is important to note that, unlike the simulator, the kernel can take advantage of multi-core systems, and also that we have already detected many places in the kernel where performance is suboptimal, most surely related to the youth of the code base.

Observing this graph, it becomes clear that the technology is viable, and that there is still room for improvement.

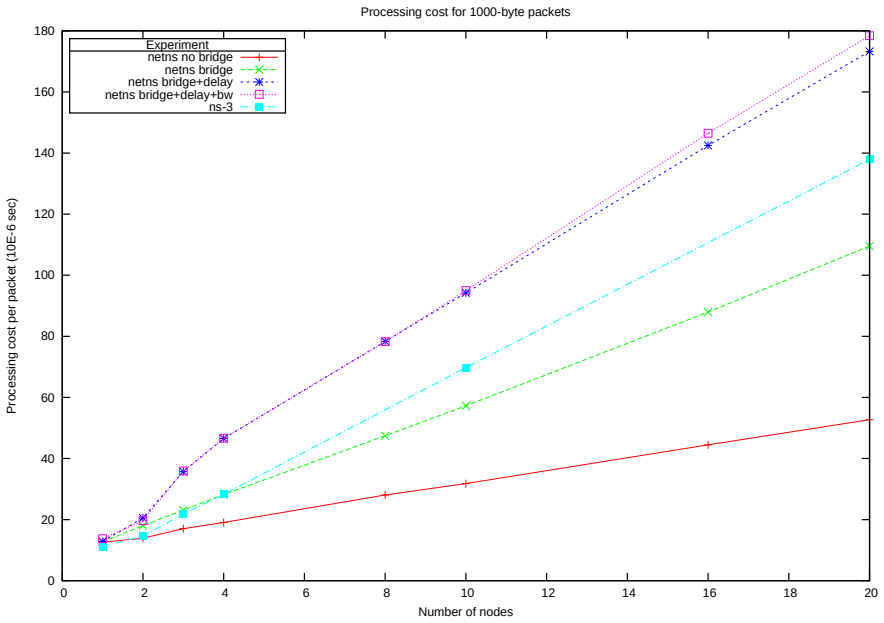

Figure 3.3: Processing cost as a function of nodes

3.2 The netns library

The main part of the work carried out consisted in creating a Python library that would execute the correct steps to set up network namespaces, create and

configure virtual network devices, install network link emulation filters, and start programs.

3.2.1 Components

Before describing the architecture, it is useful to describe the conceptual blocks on which the library is built, which mostly match the class hierarchy, and explain briefly how they are implemented.

controlling process The process calling directly the library becomes the controlling process, which is responsible for the creation of most of the objects described here, coordinating all the delegate processes, and cleaning up after the experiment has finished. It does not match any specific class.

Node A node is the equivalent of a network host, it contains a separate network stack, IP addresses, and network stack. The implementation is a child process with a new, separate network namespace, which executes commands on behalf of the controlling process received via an ad-hoc RPC protocol.

Switch The Switch object can model any network connectivity object, most notably, a network switch, but also long distance links, satellite or modem connections, etc. It provides the functionality to emulate non-ideal network conditions such as delay, jitter, limited bandwidth, packet loss, etc. It is implemented with the bridge device, to connect two or more interfaces. The `tc` subsystem in the Linux kernel allows to plug *queueing disciplines* like netem and traffic shapers, that emulate the link conditions.

NodeInterface The library provides various types of interfaces, the NodeInterface being the most important. NodeInterface models a normal network interface connected to a network switch. It is implemented as a pair of veth devices, one residing in the node's namespace and the other in the namespace of the controlling process, where is connected to a bridge device (the switch object).

P2PNodeInterface A simpler to use interface, the point-to-point interface is a direct, ideal connection between two nodes, no link emulation can be done on it. It is also implemented as a pair of veth devices, but each one of them is put into the namespace of a different node.

TapNodeInterface A special kind of interface, the TapNodeInterface sits inside a node, but instead of having a paired device connected to a bridge, it provides a stream of frames in a file descriptor. It is used to connect to the external world: it can connect two instances of the test bed, or to connect to any other real or simulated network, provided there is a compatible

device on the other side. For example, the `FileDescriptorNetDevice` object in the ns-3 simulator is meant to be connected to this kind of devices.

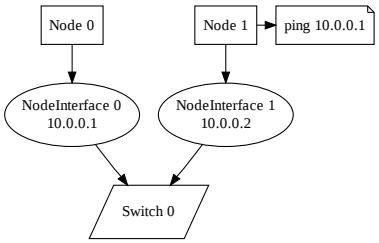

Figure 3.4: Logical view of a minimal configuration

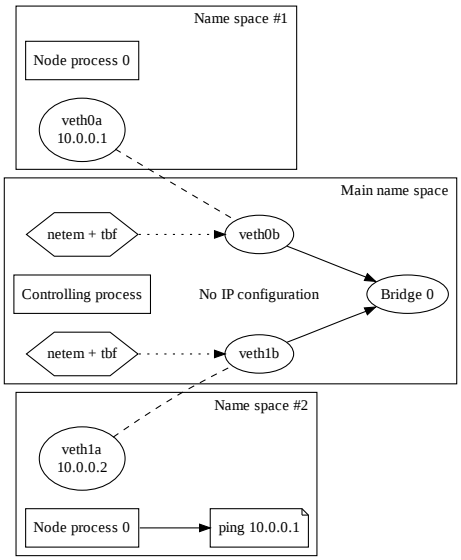

Figure 3.5: System view of the configuration from figure 3.4

A minimal network configuration is shown in figure 3.4. The set up consists of two nodes connected by a link with emulated delay and bandwidth limitation, one of the nodes is running the `ping` application to test connectivity. The real kernel objects involved in this scenario and their relationship can be seen in figure 3.5.

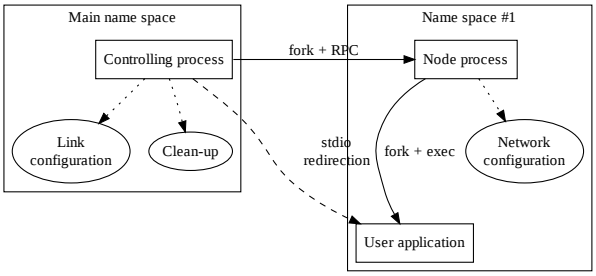

Figure 3.6: Relationship between processes and namespaces

3.2.2 Architecture

The architecture of the library is fairly simple. The program directly invoking the library becomes the controlling process. Each new node object creates a child process that is responsible for the corresponding namespace. Programs started by the user are children of the relevant node process.

Communication between processes is made by means of a simple text-based protocol that allows only a handful of specific calls. Since socket communication is not possible after namespace change, all communication is set up with `socketpair` before the new process is created.

In figure 3.6 a simple overview of the processes and namespaces created and the programs invoked by the library is shown.

A big part of the code and effort went into the seemingly mundane task of interfacing with the live system, which can in fact be daunting. One important choice regarding this, is that the library does not assume that is alone in the system: interfaces can appear or disappear, change configuration, routing tables be added automatically by the kernel, etc. To handle this, at the expense of some overhead, the code does not keep state information but asks the kernel each time, minimising in this way the probability of failure because of broken assumptions.

Non-visible code keeps track of objects created to properly clean-up when needed. Objects for which other objects have a dependency on them also carry references to ensure ordered tear down.

Python automatic destructor methods, combined with explicit calls for destruction and hooks installed in key parts of the code, assure that the system is cleaned before exiting the program. Proper clean up is a main goal of the library, as any object that is not properly removed, stays in the system cluttering its normal functioning.

3.3 Challenges

Although the library has a simple structure, there were many challenges to overcome; some of them were known in advance, many others were discovered in the way, and some design decisions had to be reconsidered in light of these problems.

3.3.1 Links and link emulation

Initially, the library was going to offer only point to point links, since they are simpler to set up and control (no bridging, only a pair of veth devices instead of two, simpler clean-up) and for most experiments they are enough. But at the beginning of the performance evaluation tests, it was discovered that the current namespace implementation in the Linux kernel does not support adding queueing disciplines to devices outside of the main namespace, rendering the links worthless for the main purpose of emulating real world conditions.

3.3.2 Kernel bugs and language support

Most of the project dealt with cutting-edge technology, and as such, many rough edges in the software. The Linux kernel started supporting network namespaces in version 2.6.26, but we found many different problems —from missing, slow, or incorrect functionality to kernel crashes— that required profiling and debugging of kernel code, and finally raised the version requirement to 2.6.35, which was officially released less than a month ago[12]. For such a new technology, there is still no support in the programming language (Python), so we had to write support libraries for the recent user-space interfaces to the netns-enabling system calls[13].

3.3.3 Application IO capture

Another interesting problem posed was that of running user applications inside the virtual node. Since it is essential to be able to capture the output of programs in real time, and a much desired feature is to be able to interact with them with a regular terminal emulation program, the standard streams need to be available to the user of the library, which resides in a different network namespace from the application being run.

The initial, naïve approach was to create named sockets before executing the program, so the user would be able to connect to them and talk with the application. But UNIX domain sockets are also isolated when changing namespaces, so this method did not work. The second idea was to use named pipes, which provide a similar concept, but pose problematic synchronization issues when input and output streams are to be redirected.

Finally, it was decided to take advantage of a mostly unknown feature of UNIX systems: the ability to pass open file descriptors among different

processes using a special message called `SCM_RIGHTS`[14, pages 601-614]. This allowed creating the communication channels in the user-facing namespace, and to transfer them to the node namespace for redirection. The mechanism requires the use of UNIX domain sockets which, as said before, cannot be used to connect two distinct namespaces. So, a second trick is needed: said UNIX socket is created before the node controlling process is forked or the namespace is separated. This way, the child process inherits the already connected socket, and once connected it remains functional even with the namespace change. Implementing this feature required developing a second support library[15], due to Python not supporting the special system call.

3.4 NEPI backend

With the netns library ready to be used, the next task was to prepare the wrapping code for it to be used in NEPI. From the point of view of NEPI, this library is quite simple to adapt, which is not surprising knowing that it was thought from the beginning to be used in that context.

The biggest problems posed by this subject were related to the very nature of the system: there is no concept of a stop time, nor the start time can be synchronised. Simulators, by contrast, have a precise moment in which all the experiment comes to life, and another exact moment when everything finishes. To solve this, the backend is careful to spawn in the correct order every object, leaving the applications for the last moment, so the start of the experiment is as synchronised as possible. For stopping the experiment, an internal timer is used, and then the applications are killed as fast as possible when the time comes.

3.5 Mixed experiments

The final objective of this work, not yet complete, is to be able to perform experiments in which more than one tool is involved, relying on NEPI to take care of the deployment, address assignment, interconnection of the components, and finally, result gathering and clean up.

We are very close to this objective, some preliminary tests have been already executed successfully, and we expect to have some concrete experiments up and running by the end of September. In figure 3.7, the proof of concept test is shown as NEPI objects. On the left, the ns-3 backend contains only the minimum objects to obtain a working node, plus a "Ping" application; the special `FileDescriptorNetDevice` is responsible of converting Ethernet frames to and from the file descriptor into ns-3 packet objects. On the right side, the netns object `TapNodeInterface` represents a tap device, which performs a similar function, but instead injecting those packets into the network stack, so they finally reach the node and we get a ping reply.

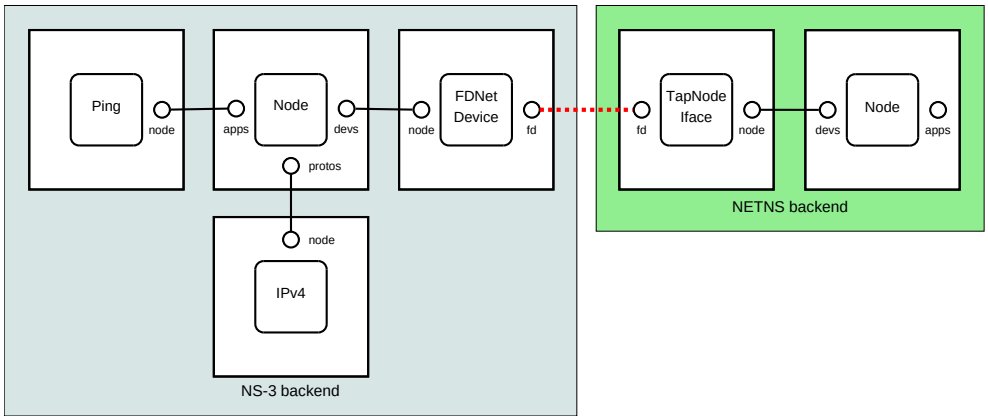

Figure 3.7: Proof-of-concept mixed experiment

By the end of the internship, we expect to be able to run large-scale, mixed experiments, with different configurations. One of the use-cases marked as a deveolpment objective is the one shown in figure 3.8, in which the network is partitioned into manageable chunks, and then fed to different instances of the experimentation tools.

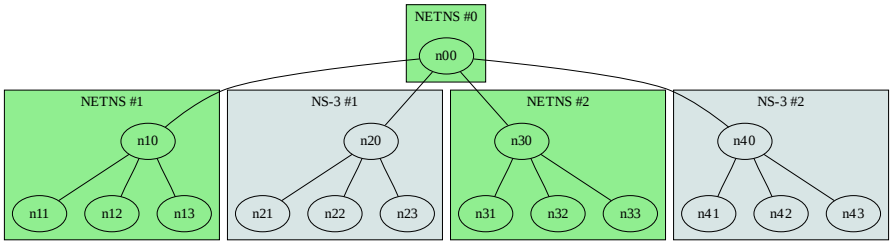

Figure 3.8: A larger experiment

3.6 Timeline

The following table shows the organisation of work during the past months, and also the planning for the remaining of the internship.

Period	Description
	Viability and performance analysis
April	Rough scripts written to perform tests Baseline measurements Cost analysis for each functionality Kernel profiling to find bottlenecks First kernel problems detected, patch proposed
May	Stress testing triggers kernel panics
	Design and implementation
May	API defined Project structure, build system Tests and coverage automatization Coding starts Unit tests for basic functionality Creation of the python-unshare library
June	Full-blown coding Namespace creation IPC protocol Creation of the python-passfd library Support for executing processes inside a namespace
July	Handling of interfaces, creation and destruction Modelling of different interface configurations Bridging support Routing table configuration support Handling of the traffic control system Link characteristics support
August	Bug fixing Comprehensive tests Reproduction of performance tests with the new code Write NEPI backend
September	Code to set up communication channels between backends Run and test different mixed experiment scenarios Complete tests, documentation Improve clean-up, robustness

CHAPTER 4

Conclusion

An important part of the research activity consists in conducting experiments. In the case of network research, the experimentation can be carried out using different schemes which mainly include three elements: simulations, emulation and physical deployment. Each of these elements can be used independently, but the most interesting results are obtained when they are combined.

Recently, support for network namespaces has been added to the Linux kernel opening a new opportunity for virtualisation-based network testing. An important part of the work I did during my internship lay in implementing the netns network emulator based on this new technology. My final goal consisted in using the netns emulator in conjunction with the ns-3 network simulator to realize a mixed experiment and thus both prove the feasability of the approach, and show the advantages this can bring to network research.

After completing the development of the netns emulator, I worked on adding new capabilities to the ns-3 network simulator project to adapt it to my requirements. Finally, I needed a way to easily describe and deploy mixed experiments so I extended the Network Experimentation Programming Interface, NEPI, which I developed in my previous internship, to achieve this. The outcome has been greatly successful so far, although there is still work to be done during the next month to fully support mixed experiments and reach my objectives.

As in every ambitious project, a number of things are yet to be improved and supported. Large scale experiments need to be tested and more performance evaluation of all results would be necessary. Also, in order to fully explore the possibilities provided by mixed network experiments, new backends for different experimentation platforms should be added to the NEPI project, especially for interfacing with physical networks. After this, a thorough comparison of the overall performances on the different experimentation schemes can provide new insights on the way researchers could profit from mixed exper-

iments. Finally, a Graphical User Interface would allow researchers to easily learn and use all the developed tools enabling them to quickly obtain results.

To conclude, I would like to say that I consider the work I did during these five months as truly relevant to networking research and I believe myself and others will be able to continue it for the benefit of the research community. I feel this internship has given me an opportunity to contribute to the research world. In a personal aspect, this experience has paid greatly, having had the opportunity of working in a highly collaborative environment that favoured team work and knowledge exchange. I have also seen my coding and software design skills improved. Taking all this into account, it has been one of the most gratifying learning experiences for me so far, and this is why I considered I have accomplished the objectives of this internship.

APPENDIX A

Sample scripts

A.1 A simple test using the netns library

This script demosntrates the basic usage of the netns library. It creates three nodes in a linear topology, showing the usage of two different kind of interfaces, link emulation, and IP configuration. Then, it tests connectivity and link conditions, showing how processes are started in the nodes.

The script is also included in the library distribution, under the name `sample.py`, it needs to be run as the root user.

```
#!/usr/bin/env python
import netns, subprocess

# each Node is a netns
node0 = netns.Node()
node1 = netns.Node()
node2 = netns.Node()

# interface object maps to a veth pair with one end in a netns
if0  = netns.NodeInterface(node0)
if1a = netns.NodeInterface(node1)
# Between node1 and node2, we use a P2P interface
(if1b, if2) = netns.P2PInterface.create_pair(node1, node2)

switch0 = netns.Switch(
        bandwidth = 100 * 1024 * 1024,
        delay = 0.01, # 10 ms
        delay_jitter = 0.001, # 1ms
        delay_correlation = 0.25, # 25% correlation
```

```
        loss = 0.005) # 0.5% packet loss

# connect the interfaces
switch0.connect(if0)
switch0.connect(if1a)

# bring the interfaces up
switch0.up = if0.up = if1a.up = if1b.up = if2.up = True

# Add IP addresses
if0.add_v4_address(address = '10.0.0.1', prefix_len = 24)
if1a.add_v4_address(address = '10.0.0.2', prefix_len = 24)
if1b.add_v4_address(address = '10.0.1.1', prefix_len = 24)
if2.add_v4_address(address = '10.0.1.2', prefix_len = 24)

# Configure routing
node0.add_route(prefix = '10.0.1.0', prefix_len = 24,
        nexthop = '10.0.0.2')
node2.add_route(prefix = '10.0.0.0', prefix_len = 24,
        nexthop = '10.0.1.1')

# Test connectivity first. Run process, hide output and check
# return code
null = file("/dev/null", "w")
app0 = node0.Popen("ping -c 1 10.0.1.2", shell = True,
        stdout = null)
ret = app0.wait()
assert ret == 0

app1 = node2.Popen("ping -c 1 10.0.0.1", shell = True,
        stdout = null)
ret = app1.wait()
assert ret == 0
print "Connectivity IPv4 OK!"

# Now test the network conditions
# When using a args list, the shell is not needed
app2 = node2.Popen(["ping", "-c1000000", "-f", "10.0.1.2"],
        stdout = subprocess.PIPE)

out, err = appt2.communicate()

print "Ping outout:"
print out
```

Bibliography

[1] The ns-3 network simulator.
URL http://www.nsnam.org/

[2] B. Ben Romdhanne, D. Dujovne, T. Turletti, W. Dabbous, Efficient and scalable merging algorithms for wireless traces, Tech. rep.
URL http://hal.archives-ouvertes.fr/inria-00397832/en/

[3] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, M. Wawrzoniak, Operating system support for planetary-scale network services, in: NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, 2004, pp. 19–19.

[4] M. Lacage, M. Ferrari, M. Hansen, T. Turletti, W. Dabbous, Nepi: using independent simulators, emulators, and testbeds for easy experimentation, SIGOPS Oper. Syst. Rev. 43 (4) (2010) 60–65. doi:http://doi.acm.org/10.1145/1713254.1713268.
URL http://roads.mytestbed.net/papers/roads09_paper_1.pdf

[5] LXC maintainers, Linux containers project page (2009).
URL http://lxc.sourceforge.net/

[6] SWSoft, Openvz.
URL http://www.openvz.org

[7] The Linux fundation, Netem project page (2009).
URL http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

[8] A. Tirumala, M. Gates, F. Qin, J. Dugan, J. /Ferguson, Iperf - the tcp/udp bandwidth measurement tool.
URL http://iperf.sourceforge.net/

[9] R. Jones, Netperf.
URL http://www.netperf.org/netperf/

[10] T. Gleixner, D. Niehaus, Hrtimers and beyond: Transforming the linux time subsystems, in: Ottawa Linux Symposium (OLS'06), 2006.

[11] D. Miller, How SKBs work.
URL http://vger.kernel.org/~davem/skb.html

[12] L. Torvalds, Linux kernel version 2.6.35 announcement (2010).
URL http://www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.35

[13] M. Ferrari, python-unshare project page (2010).
URL http://yans.pl.sophia.inria.fr/trac/nepi/wiki/netns/python-unshare

[14] R. W. Stevens, S. A. Rago, Advanced Programming in the UNIX® Environment (2nd Edition), Addison-Wesley Professional, 2005.

[15] M. Ferrari, python-passfd project page (2010).
URL http://yans.pl.sophia.inria.fr/trac/nepi/wiki/netns/python-passfd

Summary

To demonstrate the feasibility and the usefulness of conducting mixed network experiments that integrate real-time network simulations together with virtualisation-based test beds, we focused in this internship on the task of adding support for a light-weight Linux virtualisation-based test bed to Nepi. To do so, we first validated the robustness and the efficiency of the Linux network namespaces virtualisation technology and then designed and implemented a python-based library to create and manage the lifetime of Linux network namespace containers. We finally integrated this library within Nepiand we plan to demonstrate a running mixed experiment before the end of this internship.