NEPI tutorial

Conducting CCNx experiments on PlanetLab Europe

INRIA
PLANETE team

Alina Quereilhac

Version 0.1

May 16, 2012

Contents

CHAPTER

1 Introduction

In this tutorial we will specifically focus on describing the use of NEPI for conducting experiments involving CCNx and PlanetLab Europe.

NEPI [4, 5], the *Network Experimentation Programming Interface*, is a life-cycle management tool for network experiments. The idea behind NEPI is to provide a single tool to design, deploy, and control network experiments, and gather the experiment results. Going further, NEPI was specially conceived to function with arbitrary experimentation platforms, so researchers could use a single tool to work with network simulators, emulators, or physical testbeds, or event a mixture of them.

NEPI supports conducting hybrid-experiments, by deploying overlay topologies across different experimentation platforms and communicating them through special tunneling components. To accomplish this, NEPI provides a high-level interface to describe experiments, that is independent from any experimentation platform, but is able to capture platform specific configurations. Experiment definitions can be stored in XML format to be later reproduced, and modified according to experimentation needs. Experiment execution is orchestrated by a global experiment controller, which is platform independent, and many platform-dependent controllers, creating a control hierarchy that is able to adapt to platform specific requirements while providing an integrated control scheme.

Currently NEPI provides support for three different experimentation platforms: The *ns-3 network simulator* [2], the *netns emulator* [1], and the *PlanetLab Europe* [3] distributed testbed.

NEPI is licensed under GPLv2 and is implemented as a Python library, for those who prefer scripting, but it also provides a Graphical User Interface called NEF.

CHAPTER

2 Getting started

To start using NEPI to conduct CCNx experiments on PlanetLab Europe, you will first need to get a copy of the source code and get a PlanetLab Europe account.

2.1 Resources

Before starting consider visiting NEPI wiki page at:

`http://nepi.pl.sophia.inria.fr`

There you will find publications on NEPI and more information on how to run experiments.
For any questions regarding NEPI please join the users mailing list at *nepi-users@nepihome.org*. To join the list send an email to:

sympa@lists-sop.inria.fr with subject *SUBscribe nepi-users <your-username>*.

2.2 The Planetlab Europe account

If you do not already have a PlanetLab Europe account, you will need to get one. Please contact the Principal Investigator (PI) registered for PlanetLab Europe in your institution.

Once you have your account, you will need to log into the PlanetLab Europe site at the url below, and upload you *public ssh key*.

`https://planet-lab.eu`

Your account should have a PlanetLab *Slice* associated to it upon creation.

For basic experiments involving PlanetLab Europe, such as the first experiment shown in Chapter 3, there are no further requirements on the *Slice* configuration.

For experiments involving automatic deployment of overlay topologies, using tunnels and virtual interfaces, or other advanced features of NEPI, you will need to request your PlanetLab Europe administrator to add some extra *vsys tags* to your *Slice*. We will revisit the specific configuration requirements later on.

2.3 System requirements

NEPI was tested mainly in Linux systems. So we strongly recommend to use a Linux box to implement the examples detailed in this tutorial.

To use NEPI you will need the install the following dependencies:

- python >= 2.6
- python-ipaddr >= 2.1.5

To use NEF, NEPI's graphical user interface, you will also need to install:

- libqt4 >= 4.6.3
- python-qt4 >= 4.7.3

Additionally you will need to install the *Mercurial* version control system, to get a copy of NEPI's source code.

To run the experiment cases explained on this tutorial, you will also need to have *VLC* < 2.0 installed in your system. [1]

2.4 Installation

To get a copy of NEPI it is enough to clone the mercurial repository.

```
$ hg clone http://nepi.pl.sophia.inria.fr/code/nepi
```

If you wish to have NEPI installed in your system, you can use the *setup.py* script in NEPI source directory. This step is completely optional and you can skip it, if for instance you do not have root access on your machine.

```
$ sudo python setup.py install
```

Additionally, you can get a copy of NEF, NEPI's graphical user interface.

```
$ hg clone http://nepi.pl.sophia.inria.fr/code/nef
```

NEF provides as well a *setup.py* script for installation.

2.5 Running an experiment

NEPI provides several example experiments in the *examples* directory. Specific CCNx on PlanetLab Europe examples can be found in the "examples/ccnx" directory.

Before running PlanetLab experiments with NEPI, it is advisable to make sure that there is a *ssh-agent* running.

```
$ ssh-agent
```

[1] Early tests with VLC 2.0 on MAC OS have exhibited some issues.

To run experiments with NEPI you only need to invoke the NEPI scripts using the python interpreter.

```
$ python my-nepi-script.py
```

If you did not install NEPI in your system, you will need to specify the path to NEPI's source code in the *PYTHONPATH* environmental variable.

```
$ cd <path-to-nepi-source>
$ PYTHONPATH=$PYTHONPATH:src python my-nepi-script.py
```

If NEF is installed in your system, you can launch it by typing the command *nef*.

```
$ nef
```

To use NEF without installing it in your system, you will need to run the executable *nef* file in the source of the project, adding the paths to NEPI and NEF source code directories to the *PYTHONPATH* environmental variable.

```
$ cd <path-to-nef-source>
$ PYTHONPATH=$PYTHONPATH:src:<path-to-nepi-source>/src ./nef
```

CHAPTER

3 CCNx experiments on PlanetLab Europe

In this chapter we will discuss in detail two example experiment scripts that can be found in NEPI source directory, in the “examples/ccnx” folder.

3.1 Deploying CCNx on PlanetLab from local source code

This first experiment example corresponds to the “examples/ccnx/planetlab_ccnx_unicast.py” script. This experiment has for objective to observe the effects of CCNx caching when streaming video over a series of PlanetLab Europe nodes associated in series though CCNx FIB entries.

A complete demonstration video for this experiment is available at:

`http://nepi.pl.sophia.inria.fr/wiki/nepi/CCNxOnPlanetLabEurope`

At the begining of the experiment execution, NEPI will automatically provision the PlanetLab Europe nodes, and deploy a CCNx daemon in all the allocated nodes.

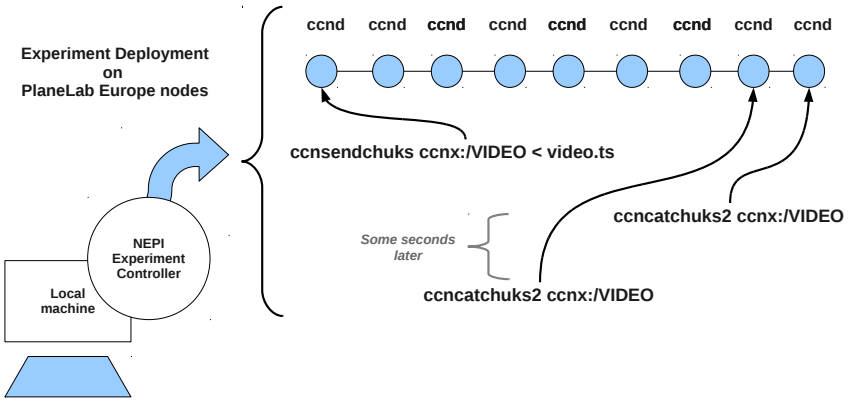

Figure 3.1: CCNx on PlanetLab experiment diagram

In the first node, a video will be published, by issuing the command *ccnsend-chunks ccnx://VIDEO < video.ts*. Then, the video will be retrieved on the last node of the series with *ccncatchunks2 ccnx://VIDEO*. After a few seconds, the video will again be retrieved in the same way, but this time on the previous node.

Because of the caching that occurs along the CCNx interest path, it is expected that the second video stream will not present any of the problems that might be visualized on the first stream (visual artifacts and freezing of scenes), until both streams are synchronized. After this point, as no interest cache exists for the arriving content, the two streams should be affected in a similar way.

In this example case, we exploit NEPI's ability to compile and install CCNx source code from a local tarball. This feature allows researchers to test new ideas on PlanetLab Europe by directly modifying CCNx source code.

3.1.1 How to run it

There are several arguments that can be passed to the script. To get the full list of arguments you can execute the script passing *--help*.

```
$ python examples/ccnx/planetlab_ccnx_unicast.py --help

Usage: planetlab_ccnx_unicast.py -s <pl_slice> -H <pl_host> -k <ssh_key>
                           -u <pl_user> -p <pl_password> -v <vsys_vnet>
                           -N <host_names> -c <node_count> -d <delay>
                           -P <ccn-local-port>

Options:
  -h, --help     show this help message and exit
  -s SLICENAME, --slicename=SLICENAME
                        PlanetLab slicename
  -H PL_HOST, --pl-host=PL_HOST
                PlanetLab site (e.g. www.planet-lab.eu)
  -k PL_SSH_KEY, --ssh-key=PL_SSH_KEY
                Path to private ssh key used for PlanetLab
                authentication
  -u PL_USER, --pl-user=PL_USER
                PlanetLab account user (i.e. Registration email
                address)
  -p PL_PWD, --pl-pwd=PL_PWD
                PlanetLab account password
  -v VSYS_VNET, --vsys-vnet=VSYS_VNET
                Value of the vsys_vnet tag addigned to your slice.
                (e.g. 192.168.3.0/16)
  -N HOSTNAMES, --host-names=HOSTNAMES
               Comma separated list of PlanetLab hostnames to use
  -c NODE_COUNT, --node-count=NODE_COUNT
                Number of nodes to use
  -d DELAY, --delay=DELAY
                Time to wait before retrieveing the second video
                stream in seconds
  -P PORT, --ccn-local-port=PORT
                Port to bind the CCNx daemon
```

Pay special attention to the *-P* argument, it specifies the port to use as the value for CCN_LOCAL_PORT. If this argument is not passed, the CCNx daemon will use

the default port 9695. The use of this default port, might cause problems if different CCNx daemons are run at the same time in the same nodes, for example if this example script is executed in parallel with different slices on the same nodes. For this reason, choosing a random port to pass as argument *-P* is a good practice.

To run the experiment, execute the ssh-agent command followed by the experiment script with at least the following arguments:

```
$ ssh-agent
$ PYTHONPATH=$PYTHONPATH:src python examples/ccnx/planetlab_ccnx_unicast.py
    -s slicename -u planetlab-user@domain -p password -k /home/user/.ssh/
    id_rsa
```

3.1.2 The code

There are some portions of the code of this example experiment that are worth explaining. One of these parts is the instantiation of the *CCNxDaemon*, which is done by the *create_ccnd* function.

```
1  def create_ccnd(pl_node, routes, slice_desc):
2      pl_app = slice_desc.create("CCNxDaemon")
3
4      # Build path to local CCNx source tarball
5      path_to_source = os.path.join(os.path.dirname(os.path.abspath(__file__)),
6          "ccnx-0.6.0rc3.tar.gz")
7      pl_app.set_attribute_value("sources", path_to_source)
8
9      # Build instructions for the CCNx source code.
10     pl_app.set_attribute_value("build",
11             "tar xzf ${SOURCES}/ccnx-0.6.0rc3.tar.gz && "
12             "cd ./ccnx-0.6.0rc3 && "
13             "./configure && make ")
14
15     # Install instructions for the CCNx binaries
16     pl_app.set_attribute_value("install",
17              -r ./ccnx-0.6.0rc3/bin ${SOURCES}")
18
19     # Build the string with the routes to add as FIB entries.
20     # We use a wildcard to replace the public IP address of the
21     # node during runtime, once this IP is known
22     routes = "|".join(map(lambda route:
23             "udp {#[iface_%s].addr[0].[Address]#}" % route, routes))
24
25     # Add unicast ccn routes
26     pl_app.set_attribute_value("ccnRoutes", routes)
27
28     # Enable traces
29     pl_app.enable_trace("stdout")
30     pl_app.enable_trace("stderr")
31
32     # Associate application to node
33     pl_app.connector("node").connect(pl_node.connector("apps"))
```

This function begins by creating a *CCNxDaemon*. During deployment this will be translated into the execution of a *ccnd* command in the PlanetLab node.

A *CCNxDaemon* is configured by default to use the sources for CCNx version 0.6.0, published on the CCNx official releases page.

`http://www.ccnx.org/releases`

The attribute *ccnxVersion* holds a list of the official releases that can be automatically installed by NEPI. However, if the user requires to use a modified version of the CCNx, he/she can specify a local tarball to be installed in the PlanetLab nodes. The local path to this tarball is specified using the *sources* attribute. When a custom source is employed, the user also needs to set the attributes *build* and *install*, to allow NEPI to execute the correct build and installation instructions (see lines 10 and 16).

In order to configure the FIB entries on a CCNx daemon, it is necessary to specify them in the attribute *ccnRoutes* as a string, where different entries are separated by a "|" . We build this string in line 22.

We use a wildcard expression, *{#[iface_label].addr[0].[Address]#}*, to make sure NEPI replaces it for the correct network interface IP address of the target PlanetLab Europe node, which we do not know at design time. To reflect this configuration, NEPI will invoke the command *ccndc add* for each CCNx FIB entry.

Different CCNx configurations, where CCNx daemons form a circle or a tree, can be achieved by providing appropriate values to the *ccnRoutes* attribute.

For this example experiment we chose a configuration in series. We form the set of entries corresponding to each CCNx daemon, by traversing the list of PLanetLab Europe hostnames to be used in the experiment. Each CCNx daemon will have an entry pointing to the previous and the next hostname.

```
1   # Traverse the hostnames list to create a dictionary of FIB entries
2   # to configure the CCNx daemon on each PlanetLab node.
3   ccn_routes = dict()
4   prev_hostname = None
5   for hostname in hostnames:
6       ccn_routes[hostname] = list()
7       if prev_hostname:
8           ccn_routes[hostname].append(prev_hostname)
9           ccn_routes[prev_hostname].append(hostname)
10      prev_hostname = hostname
```

3.2 Using multicast FIB entries for CCNx on PlanetLab

The second experiment example corresponds to the script "examples/ccnx/planetlab_ccnx_multicast.py".

Before attempting to run this experiment, you will need to make sure your PlanetLab slice is appropiatelly configured. If the following *vsys tags* are not already added to your slice, you will need to request your PlanetLab Europe administrator to add them.

- **vsys ipfw-be**. The vsys ipfw-be tag enables PlanetLab nodes in a slice to use dummynet for emulating queue and bandwidth limitations, delays, packet losses, and multipath effects.

- **vsys vroute**. The vsys vroute tag enables the use of vroute script to manipulate routing table entries for virtual interfaces, in a secure manner without interfering with other slices.

- **vsys vif_up**, **vsys vif_down** and **vsys fd_tuntap**. The vsys vif_up, vsys vif_down and vsys fd_tuntap tags enable the use of special scripts for virtual interface and tunnel creation using TAP/TUN devices.

- **vsys_vnet <private network address>/<net prefix>**. The vsys_vnet tag associates a slice to an administrator-approved subnet segment, so addresses in that segment can be assigned to the virtual interfaces in an experiment.

This second experiment is generally similar to the previous one. The main difference between the two examples is that the second one uses multicast FIB entries to configure the CCNx daemons.

To achieve this, because the PlanetLab Europe nodes are connected through the Internet, we build an overlay network made up of tunnels, connecting the nodes.

The creation of the components that describe the tunnels between PlanetLab nodes is done by the function *create_tunnel*.

```
1  def create_tunnel(slice_desc, address1, prefixlen1,
2          address2, prefixlen2):
3      pl_tun = slice_desc.create("TunInterface")
4      pl_node.connector("devs").connect(pl_tun.connector("node"))
5
6      pl_tunpeer = slice_desc.create("TunInterface")
7      pl_peer.connector("devs").connect(pl_tunpeer.connector("node"))
8
9      pl_tun.connector("udp").connect(pl_tunpeer.connector("udp"))
10
11     ip = pl_tun.add_address()
12     ip.set_attribute_value("Address", address1)
13     ip.set_attribute_value("NetPrefix", prefixlen1)
14
15     peerip.set_attribute_value("Address", address2)
16     peerip.set_attribute_value("NetPrefix", prefixlen2)
```

To describe a tunnel between two PlanetLab nodes in NEPI, we need to create two components representing virtual interfaces. In this case, we create two *TunInterfaces*, and connect them in the *udp* connector.

Finally, we need to assign IP addresses to the virtual interfaces. These addresses must belong to the network segment associated to our PlanetLab Europe slice by the *vsys_vnet* tag.

CHAPTER

4 Problem resolution

4.1 Insufficient resources to grant request

NEPI deals with unresponsive PlanetLab nodes by adding them to a blacklist. Once a node is added to the blacklist, it will be discarded in the future when provisioning an experiment. This might lead to a *"Insufficient resources to grant request"* error, if a previously blacklisted node is explicitly requested in the experiment description.

```
ERROR:root:Exception occurred in asynchronous thread:
Traceback (most recent call last):
  File "/home/alina/repos/nepi/src/nepi/core/execute.py", line 328, in wrapped
    callable(*p, **kw)
  File "/home/alina/repos/nepi/src/nepi/testbeds/planetlab/execute.py", line
      186, in do_preconfigure
    self.do_resource_discovery()
  File "/home/alina/repos/nepi/src/nepi/testbeds/planetlab/execute.py", line
      307, in do_resource_discovery
    solution = resourcealloc.alloc(reqs, sample=pickbest)
  File "/home/alina/repos/nepi/src/nepi/testbeds/planetlab/resourcealloc.py",
      line 62, in alloc
    raise ResourceAllocationError, "Insufficient resources to grant request"
ResourceAllocationError: Insufficient resources to grant request
```

If the PlanetLab node is no longer unresponsive, the user can restore it to the available nodes, by editing the blacklist file.

```
$ vim ${HOME}/.nepi/plblacklist
```

4.2 Invalid GPG Key

For some PlanetLab nodes, it might happen that the gpg keys that get installed in /etc/pki/rpm-gpg/ are obtained from Fedora 14, while the sliver is based on Fedora 12. This causes the following error when attempting to "yum install" Fedora packages.

```
The GPG keys listed for the "Fedora 12 - i386 - Updates" repository are already
installed but they are not correct for this package.
```

To fix this problem log into the PlanetLab node and execute the following command:

```
$ sudo yum reinstall -y --nogpgcheck fedora-release
```

Another manifestation of this problem is to see a *RuntimeError: Failed install build sources* during the experiment deployment indicating that the *build* step failed for some application:

```
RuntimeError: Failed install build sources:
    ==> /home/inria_heartbeat/nepi-ccnd-25/installlog
        = cp: cannot stat `./ccnx-0.6.0rc3/bin': No such file or directory

==> /home/inria_heartbeat/nepi-ccnd-25/buildlog <==
    done
    /home/inria_heartbeat/nepi-ccnd-25/build/ccnx-0.6.0rc3/csrc/lib
    make[2]: Entering directory `/home/inria_heartbeat/nepi-ccnd-25/build/ccnx
         -0.6.0rc3/csrc/lib'
    cc -g -Wall -Wpointer-arith -Wreturn-type -Wstrict-prototypes -I../include -
        D_REENTRANT -fPIC -c -o ccn_client.o ccn_client.c
    make[2]: cc: Command not found
    make[2]: *** [ccn_client.o] Error 127
    make[2]: Leaving directory `/home/inria_heartbeat/nepi-ccnd-25/build/ccnx
         -0.6.0rc3/csrc/lib'
    make[1]: *** [default] Error 1
    make[1]: Leaving directory `/home/inria_heartbeat/nepi-ccnd-25/build/ccnx
         -0.6.0rc3/csrc'
    make: *** [default] Error 1
```

To verify that the problem is related to a package installation problem, log in to the node and inspect the *buildlog* file in the indicated deployment directory:

```
$ cat nepi-ccnd-25/buildlog
```

If you see the following message indicating that there was a problem with the public key of a package, you should also perform the installation of *fedora-release* as indicated before, and rerun your experiment.

```
downloading Packages:
warning: rpmts_HdrFromFdno: Header V3 DSA signature: NOKEY, key ID 4f2a6fd2


Public key for keyutils-libs-devel-1.2-2.fc6.i386.rpm is not installed
```

4.3 Connection closed by remote host

SSH defines limits to the number of connections that can be created over a period of time. If this number is exceeded, a new SSH connection will fail with the message *ssh_exchange_identification: Connection closed by remote host.* NEPI is susceptible to this problem when the experiment involves the creation of many tunnels between PlanetLab nodes.

```
Traceback (most recent call last):
  File "/home/alina/repos/nepi/src/nepi/util/parallel.py", line 39, in run
    rv = callable(*args, **kwargs)
  File "/home/alina/repos/nepi/src/nepi/util/parallel.py", line 226, in __run
    return fn(*args, **kwargs)
  File "/home/alina/repos/nepi/src/nepi/core/testbed_impl.py", line 244, in
      perform_action
    getattr(factory, action)(self, guid)
  File "/home/alina/repos/nepi/src/nepi/testbeds/planetlab/metadata.py", line
      499, in postconfigure_tuniface
    element.launch()
  File "/home/alina/repos/nepi/src/nepi/testbeds/planetlab/interfaces.py", line
      292, in launch
    self.peer_proto_impl.launch()
  File "/home/alina/repos/nepi/src/nepi/testbeds/planetlab/tunproto.py", line
      663, in launch
    super(TunProtoUDP, self).launch('udp')
  File "/home/alina/repos/nepi/src/nepi/testbeds/planetlab/tunproto.py", line
      292, in launch
    self._install_scripts()
  File "/home/alina/repos/nepi/src/nepi/testbeds/planetlab/tunproto.py", line
      184, in _install_scripts
    raise RuntimeError, "Failed to set up TUN forwarder: %s %s" % (out,err,)
RuntimeError: Failed to set up TUN forwarder:
    ssh_exchange_identification: Connection closed by remote host
```

When deploying experiments on PlanetLab, NEPI uses heavy task parallelization to decrease deployment time. This parallelization directly affects the number of SSH sessions that are required over a small period of time. In order to overcame problems related to SSH limits on the number of open connections, NEPI uses the 'ControlMaster' and 'ControlPersist' options, when starting a new SSH session. The 'ControlMaster' option instructs SSH to reuse a same network connection (master) for different sessions, decreasing the number of required connections. The option 'ControlPersist' allows to automatically start a background SSH multiplex master.

Since the 'ControlPersist' option is only available since OpenSSH 5.6+, on systems using an earlier OpenSSH version NEPI will not perform the mentioned SSH connection optimization. As a consequence, a *Connection closed by remote host* error might occur during deployment.

The solutions to this problem are either to update the system's OpenSSH version, or to decrease the number of PlanetLab nodes/tunnels in the experiment.

CHAPTER

5 Experiment life-cycle support

NEPI provides support for all stages in the life-cycle of a network experiment. It distinguishes two main stages *Design* and *Execution*, and three sub stages for the *Execution* stage, *Deployment*, *Control*, and *Results*.

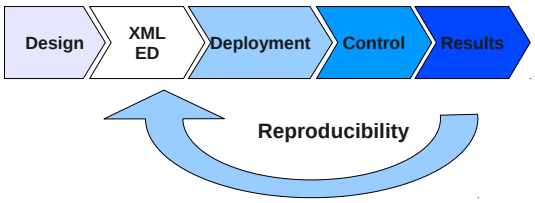

Figure 5.1: Stages in the experiment life-cycle of a NEPI experiment

Each stage is characterised by specific activities. During the *Design* stage the user constructs the *Experiment Description* (ED), defining both topology and application aspects of the experiment. At this stage NEPI performs the validation of the experiment configuration specified by the user. Once the design is finished, the **ED** is persisted to XML format. This XML **ED** will be used during the *Execution* stage by NEPI to automatically deploy the experiment.

The *Execution* stage begins with the instantiation of the NEPI *Experiment Controller* (EC). The **EC** is responsible for deploying and controlling all the components involved in the experiment. Deployment instructions are taken from the XML **ED** generated during design, and used to instantiate experiment components (e.g. provision nodes), configure them, and establish interconnections (e.g. tunnel creation). Through the *EC*, the user will be able as well to control these components during experiment execution, by changing their configuration parameters, or querying and changing their state. The *EC* makes it possible as well to retrieve the results generated during the experiment execution.

By providing a language to describe experiments with a fine grained level of detail, and putting in place all the necessary mechanisms to replay an experiment

description, NEPI allows to reproduce a same experiment as many times as needed.

A NEPI experiment can involve different experimentation platforms, namely simulations, emulations or physical testbeds. In order to hide the complexities of dealing with these different platforms from the user, NEPI provides a uniform API both for experiment design an execution.

5.1 Design

Experiment design in NEPI is based on a *Boxes and Connectors* modeling abstraction. Each supported experimentation platform defines a set of *Box* types, which represent the conceptual constructive blocks of an experiment (e.g. A *Node* box, an *Application* box, etc). These boxes can be associated to other boxes through named ports called *Connectors*. Each *Connector* represents a component characteristic, and associations between *Connectors* define the deployment instructions and runtime behavior of the experiment. As an example, a *Node* box could have a *apps* connector to which *Application* boxes can be attached. During experiment deployment, this association will be translated into the installation and execution of concrete application instances in the node.

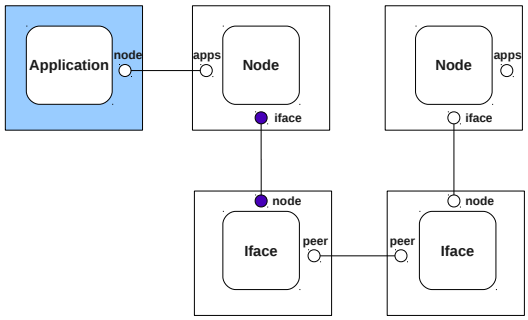

Figure 5.2: Modeling experiments with *Boxes and Connectors*

Each *Box* type exposes a set of *Attributes* which allow to define the experiment configuration. While this *Boxes and Connectors* abstraction is generic enough to be used to describe experiments for arbitrary experimentation platforms, platform specific characteristics and behavior can be expressed though the choice of *Box* types, and *Attributes* exposed.

Box types are also associated to a specific collection of *Traces*. A *Trace* defines experimental data to be generated and collected during experiment execution (e.g. a pcap file generated with tcpdump).

NEPI enforces experiment validation during the *Design* stage, by putting in place different validation criteria. Each experimentation platform defines specific *Box* types and the association rules to be enforced between boxes. These association rules define what *Connectors* can be interconnected and what is the minimum an

maximum number of connections required on a specific *Connector* (e.g. The *node* connector in an *Application* box must be associated with ath least one *Node* box on its *apps* connector). *Box Attributes* also define validation functions for their values, in order to prevent the specification of invalid experiment configuration parameters.

In order to identify each *Box* instance unequivocally throughout the experiment execution, and to be able to modify its configuration or state, each *Box* is associated to a Globally Unique Identifier (GUID), assigned during experiment design.

5.2 Execution

NEPI uses a hierarchical control structure to orchestrate experiment execution. It uses two levels of *Controller* entities, the *Experiment Controller* (EC), and the *Testbed Controllers* (TC). The **EC** is responsible for globally supervising the experiment execution, and there is only one entity of this type per experiment. The **TC** is responsible for supervising the experiment execution only on a specific experiment platform instance, and there can be many of these entities per experiment. The user communicates with the **EC** by means of the *Execution* API, and in turn the **EC** sends instructions to the corresponding **TCs**, to take specific actions (e.g. node provisioning). The **EC** is generic in the sense that it does not "know" how to perform operations on specific experimentation platforms. On the contrary, there is a specific **TC** for each supported experimentation platform which provides the necessary implementation to translate the generic **EC** instructions into concrete actions. As an example, an experiment description can require the use of a PlanetLab Europe node as well as the use of an ns-3 simulated Node. The **EC** will parse the instructions for provisioning both components from the XML **ED** provided by the user, and communicate the first instruction to a PlanetLab **TC** instance, and the second to a ns-3 **TC** instance. Each **TC** will then provision the nodes according to the requirements of the underlaying experimentation platform. The PlanetLab **TC** will identify a PlanetLab node that satisfies the user requirements and associate it to th user's slice, and the ns-3 **TC** will instantiate a C++ ns3::Node object in the ns-3 simulation.

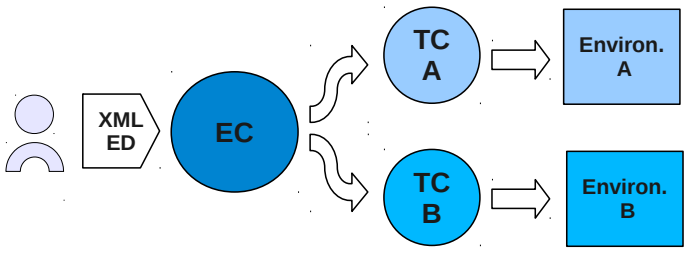

Figure 5.3: Experiment Controller hierarchy

All *Controller* entity can be executed separately, as independent processes, allowing to distribute experiment control to remote locations. A researcher could for example launch the *EC* in a remote dedicated server from his/her notebook. In this way the experiment can continue execution without requiring that the machine from where it was launched continues to be switched on. The default configuration however, is to execute all *Controller* entities in a single process in the local machine where the experiment is launched.

5.2.1 Deployment

The *Deployment* sub stage consists in provisioning, installing, instrumenting, and launching the experiment components specified in the **ED**. This sub stage is divided into well defined steps to allow global synchronization of the state of the components. After each step, all components will be in a same deployment state (e.g. Created, Configured, etc).

The steps are the following:

1. **Experimentation environment set-up and configuration**. During this step, the **EC** instantiates the **TC** entities, one per experimentation platform instance present in the **ED**. Each **TC** will take care of performing the environment initialization and configuration (e.g. A ns-3 **TC** will launch a ns-3 simulation and set all the global simulation parameters).

2. **Component instantiation**. During this step, the **TCs** will receive instructions to instantiate the experiment components, which will be under their control throughout the experiment execution. For example, a PlanetLab **TC** that is instructed to instantiate a PlanetLab Europe node, will automatically provision the node according to the requirements specified in the **ED**. Instantiation of other components can be differed until the necessary conditions are met. This conditions will depend on the particularities of each experimentation platform. In the case of a ns-3 simulation, instantiating a components means creating a ns-3 C++ object, and usually it does not need to be differed. In the case of PlanetLab however, instantiating a component might mean for example executing a command in a PlanetLab node. In this case, the actual execution of the command will be differed until the last step of the *Deployment* sub stage.

3. **Component configuration**. Each **TC** will receive configuration information from the **EC**, for every subordinate component. Configuration information is specific to each component, and can include the traces to be activated, routes or addresses to be configured, etc.

4. **Connection of components on a same TC**. In this step, each **TC** receives the connection information for the components under its control. This connection information is extracted from the associations between *Connectors* defined in the **ED**. In the case of an ns-3 simulation, connecting components

means invoking methods in ns-3 C++ objects, to associate them to other objects. In the case of PLanetLab, connecting components might mean performing specific actions in a node, such as creating a virtual interface by executing the corresponding *vsys* commands.

5. **Connection of components from different TCs**. Once all components under the control of each **TC** are instantiated and internally connected, connection to components controlled by other **TCs** can occur. During this step, for example, tunnels between components belonging to different experimentation platforms are created.

6. **Component start**. In the last step of the *Deployment* sub stage applications will be launched, trace generators started, interfaces activated, and in general, all actions to start the experiment will be performed.

5.2.2 Control

The *Control* sub stage starts after deployment, when the experiment is running. During this sub stage, the user can interact with the **EC** and modify experimental parameters in real-time. NEPI *Control* API provides methods to query and modify the state of the running applications and other components.

5.2.3 Results

After the experiment starts running, results associated to the *Traces* specified in the **ED** will start being generated. These results are stored in files locally to where they were generated. The **TCs** keep track of the location of these files, allowing the user to retrieve their contents from a remote location by issuing a request to the **EC**, which will transmit it to the corresponding **TC**.

CHAPTER

6 Python API

NEPI is implemented as a Python library. It provides *Design* and *Execution* API's to manipulate experiments.

6.1 Design

6.1.1 Object model

There are several classes that participate in the design of an experiment.

- **ExperimentDescription** class. This class represents the experiment itself. It groups experiment components used to represent the different experimentation platforms instances involved in the experiment. It also provides the API to obtain the serialized XML **ED**.
- **TestbedDescription** class. The instances of this class are used to represent platform instances (e.g. A PlanetLab Europe slice or a ns-3 simulation). They expose the methods to instantiate other experiment components.
- **FactoriesProvider** class. Provides the box types definitions for a concrete experimentation platform.
- **Box** class. Instances of this class represent the functional units that describe an experiment. (Ex: Node, Interface, Application, Channel).

6.1.2 API

This section describes the basic steps to design experiments using NEPI. The NEPI classes that are used to design experiments can be found in the *nepi.core.design* module. We need to start our Python script by importing these classes.

```
from nepi.core.design import ExperimentDescription, FactoriesProvider
```

We then need to instantiate the *ExperimentDescription* class.

```
exp_desc = ExperimentDescription()
```

And next, instantiate a *FactoriesProvider* object, indicating the experimentation platform type for which we want it to provide the *Box* types.

```
testbed_id = "planetlab"
provider = FactoriesProvider(testbed_id)
```

For each experimentation platform instance participating in our experiment, we need to instantiate and configure a *TestbedDescription* object. For example, if we want to use a PlanetLab slice in our experiment, we will need to instantiate one *TestbedDescription* object, and then set the configuration by specifying attribute values.

```
slice_desc = exp_desc.add_testbed_description(provider)
slice_desc.set_attribute_value("homeDirectory", "/tmp/experiment_home")
slice_desc.set_attribute_value("slice", slicename)
slice_desc.set_attribute_value("sliceSSHKey", pl_ssh_key)
slice_desc.set_attribute_value("authUser", pl_user)
slice_desc.set_attribute_value("authPass", pl_pwd)
slice_desc.set_attribute_value("plcHost", "www.planet-lab.eu")
slice_desc.set_attribute_value("cleanProc", True)
slice_desc.set_attribute_value("cleanHome", True)
slice_desc.set_attribute_value("plLogLevel", "DEBUG")
```

Each one of these attributes configures a specific environment parameter.

- **homeDirectory**. The local directory where experiment configuration files will be stored.
- **slice**. The slicename.
- **sliceSSHKey**. The path to the private SSH key associated to the PlanetLab Europe account.
- **authUser**. The email address registred as the PlanetLab Europe user account.
- **authPass**. The PlanetLab Europe account password.
- **plcHost**. Should be set to `www.planet-lab.eur` for PlanetLab Europe.
- **cleanProc**. If set to True, NEPI will kill all preexistent processes in the slivers.
- **cleanHome**. If set to True, NEPI will remove all preexisting directories in the slivers. BE WARE, this will destroy result files from previous experiments.
- **plLogLevel**. Console output log level.

Once the *TestbedDescription* instance is created, it can be used to instantiate *Boxes*. A PlanetLab *Node* box would be created as follows:

```
pl_node = slice_desc.create("Node")
```

Nodes in PlanetLab have at least one network interface connected to the Internet. To indicate this, we must create a *NodeInterface* and *Internet* boxes and interconnect them on the appropriate connectors. The same *Internet* box can be used to connect all the interfaces belonging to the same *TestbedDescription* instance.

```
pl_inet = slice_desc.create("Internet")
pl_iface = slice_desc.create("NodeInterface")
# Setting the attribute 'label'
pl_iface.set_attribute_value("label", "myiface")
# Interconnecting boxes though connectors
pl_iface.connector("inet").connect(pl_inet.connector("devs"))
```

All boxes have the attribute *label*, which can be used to assign a user defined identifier to a particular box. This label can be later used to retrieve the box. The *ExperimentDescription* class exposes methods to retrieve boxes by GUID or label.

```
box = exp_desc.get_element(guid)
box = exp_desc.get_element_by_label(label)
```

PlanetLab *Application* boxes have a *command* attribute to input the string commands. It is possible to use *wildcard* expressions to reference box attributes or parameters that are not known during design time, like the IP of a network interface. In these expressions boxes are referenced by the assigned label value.

```
pl_app = slice_desc.create("Application")
pl_app.set_attribute_value("command", "ping -qc3 {#[myiface].addr[0].[
    Address]#}")
pl_app.connector("node").connect(pl_node1.connector("apps"))
```

To activate a *Trace* on a box, we invoke the *enable_trace* method on the box passing the trace name.

```
pl_app.enable_trace("stdout")
```

To create tunnels between PlanetLab nodes we need to set the *tapPortBase* attribute in the PlanetLab *TestbedDescription* object.

```
slice_desc.set_attribute_value("tapPortBase", port_base)
```

Then we need to add a pair of virtual interfaces by creating two *TunInterface* or *TapInterface* boxes. Each of them will need to be connected to a PlanetLab *Node* box.

The *pcap* and *packets* traces can be activated on the virtual interface boxes. The first one will generate a pcap file with a dump of the traffic that traversed the virtual interface. The second one will also generate a file with traffic information.

Two PlanetLab virtual interface boxes can be connected though the *udp*, *tcp* or *gre* connectors. The connector chosen will define the way the tunnel is constructed during deployment.

```
pl_tun1 = pl_desc.create("TunInterface")
pl_tun1.enable_trace("pcap")
pl_tun1.enable_trace("packets")
pl_node1.connector("devs").connect(pl_tun1.connector("node"))
```

```
pl_tun2 = pl_desc.create("TunInterface")
pl_tun2.enable_trace("pcap")
pl_tun2.enable_trace("packets")
pl_node2.connector("devs").connect(pl_tun2.connector("node"))

pl_tun1.connector("udp").connect(pl_tun2.connector("udp"))
```

When working with virtual interfaces in PlanetLab, we will need to set the IP addresses. These addresses must belong to the network segment assigned to our slice by the administrator on the *vsys_vnet* tag.

```
ip = pl_tun.add_address()
ip.set_attribute_value("Address", "192.168.3.1")
ip.set_attribute_value("NetPrefix", 30)
```

We might also need to add routes to reflect the tunnels configuration to the PlanetLab node boxes.

```
route = node.add_route()
route.set_attribute_value("Destination", "192.168.3.0")
route.set_attribute_value("NetPrefix", 30)
route.set_attribute_value("NextHop", "192.168.1.1")
```

Finally, to get the XML **ED** we use the *ExperimentDescription* object.

```
xml = exp_desc.to_xml()
```

6.2 Execution

6.2.1 Object model

There are two NEPI classes that are used for executing and controling an experiment.

- **ExperimentController**. Global experiment orchestrator. The user will interact directly with this entity to manage the experiment.
- **TestbedController**. Experimentation environments instance orchestrator. It takes care of performing testbed specific tasks, such as component creation and interconnection. The *ExperimentController* will automatically instantiated one *TestbedController* entity during the experiment deployment, for each *TestbedDescription* object added during design.

6.2.2 API

This section describes NEPI execution API.

The NEPI classes responsible for executing an experiment can be found in the *nepi.core.execute* module. We need to import the *ExperimentController* class.

```
from nepi.core.execute import ExperimentController
```

Then, we need to instantiate an *ExperimentController* entity, using the XML **ED** obtained from the *ExperimentDescription* object.

```
xml = exp_desc.to_xml()
controller = ExperimentController(xml)
```

To start experiment execution, we need to invoke the method *start* on the *ExperimentController* entity.

```
controller.start()
```

All *Box* objects are associated to a Globally Universal Identifier (GUID). We cane use this GUID to query or modify the runtime state of any experiment component.

```
status = controller.status(pl_app.guid)
```

It is important to take into account that not all parameters on a component are modifiable during runtime.

```
value = controller.get(pl_tun.guid, "up")
controller.set(pl_tun.guid, "up", False)
```

We can use the ability to query the status of the experimental components to define when the experiment ends.

```
while not controller.is_finished(pl_app.guid):
    time.sleep(0.5)

controller.stop()
```

To get results generated from a Trace, we can use the *trace* method on the *ExperimentController* entity.

```
result = controller.trace(pl_app.guid, "stdout")
```

To shutdown the experiment, which will clean up all processes and release all resources in use, we need to invoke the *shutdown* method on the *ExperimentController* entity. After shutdown, we will no longer be able to automatically retrieve remote result files from the experiment though the controller. The generated results will however remain in the remote locations where they were generated, and we can always retrieve them manually.

```
controller.shutdown()
```

CHAPTER

7 GUI

NEPI provides a Graphical User Interface called NEF. It provides graphical tools to design and execute experiments with NEPI. Although it presents some limitations compared to using NEPI from a Python script, it still covers the complete experiment life-cycle, presenting different graphical views to manage experiments.

7.1 Design view

The first active view in NEF is the *Design view*. This view consists of a *Canvas* to Drag & Drop elements, and a *Tool Box* on the right, listing all available box components grouped by the experimentation environment.

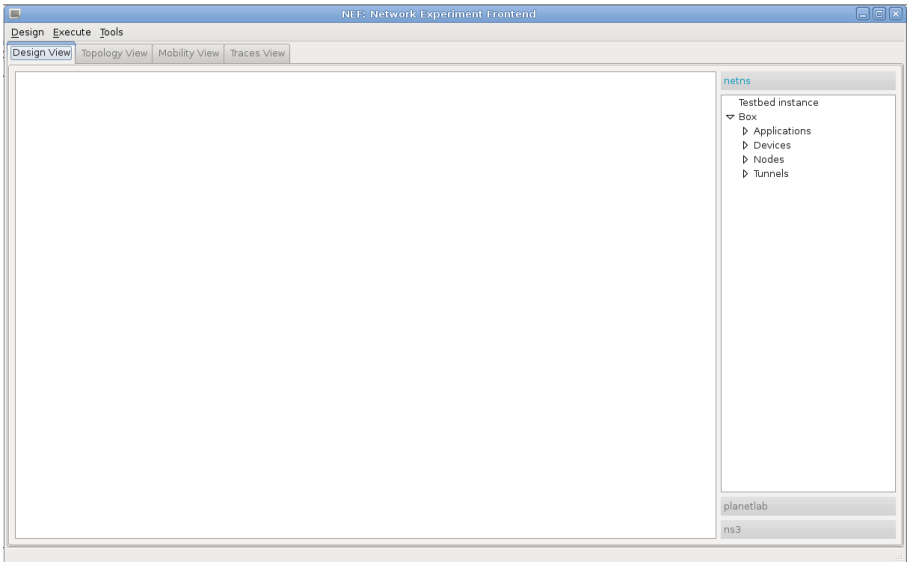

Figure 7.1: NEF Design view

Before adding *Box* components, we need to Drag & Drop a *Testbed Instance* from the desired experimentation platform section of the Tool Box. A *Testbed Instance* element represents an *TestbedDescription* component. Then we will be able to drop Boxes belonging to that experimentation platform, on top of the *Testbed Instance* component.

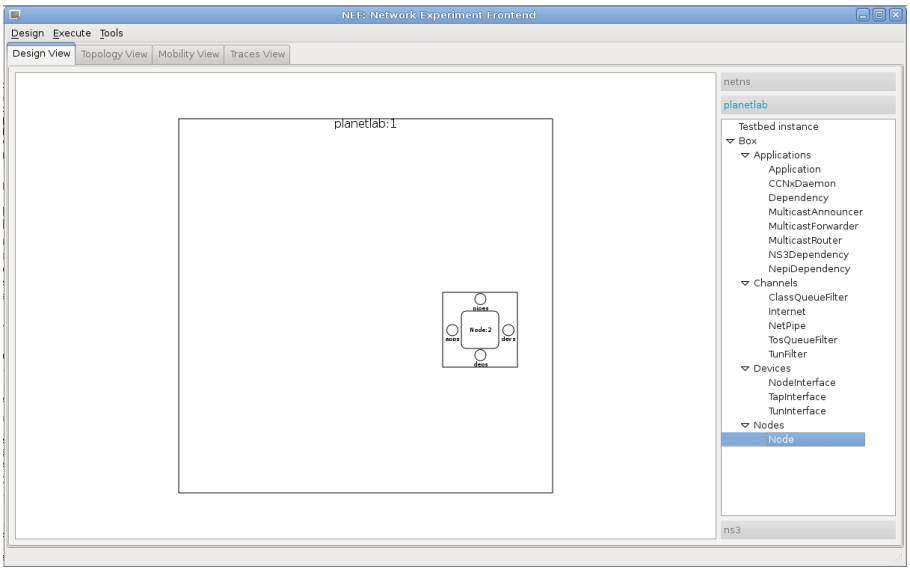

Figure 7.2: Dropping Boxes

Once boxes are added to the canvas, they can be connected by clicking on the connector circles inside the boxes. When selecting a connector, all other compatible connectors will be marked in red. A connection is made by holding the click on the first connector and dragging the mouse until the other connector is reached.

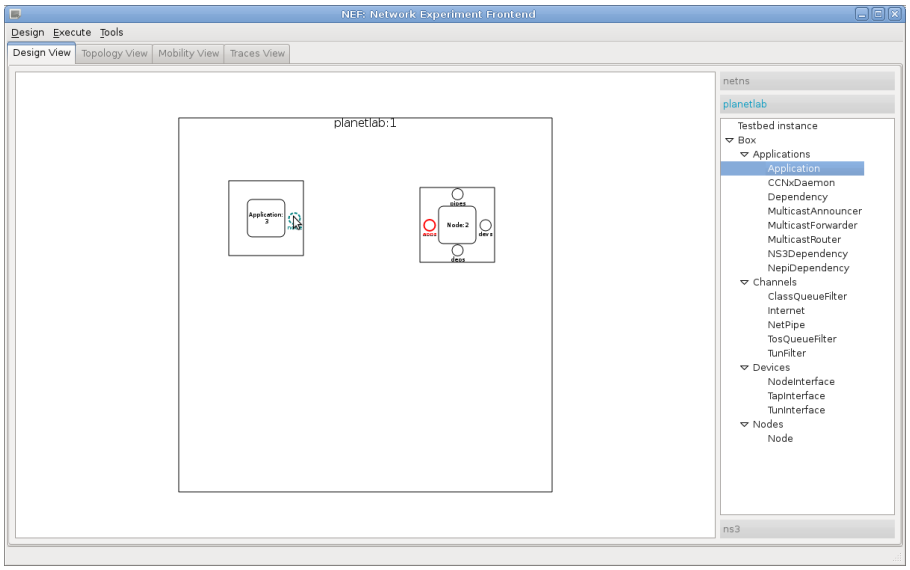

Figure 7.3: Selecting connectors

When a box is selected, we can click on it with the right button of the mouse to display a menu. The menu can include different options depending on the box type.

- **Configure** will display a window to edit the box attributes.
- **Delete** will remove the box.
- **Add traces** allows to enable traces on the box.
- **Edit routes** allows to add/delete routing entries to node boxes.
- **Edit IP addresses** allows to assign/remove addresses from network interface boxes.

Attributes can be edited by modifying the *value* column in the attributes edit box. When overing the mouse over an item in the *name* column, a help tooltip will appear describing the use of the attribute.

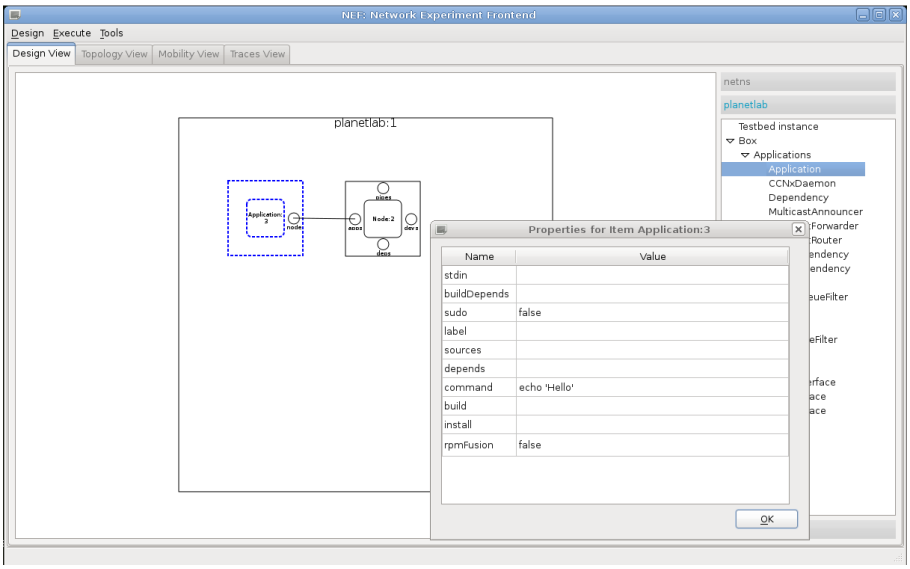

Figure 7.4: Editing Box Attributes

7.2 The main menu

The main menu provides two principal sub menus, *Design* and *Execute*. The *Design* sub menu provides options to *Store* the current design to a XML file, o to *Load* a previously stored file into the canvas.

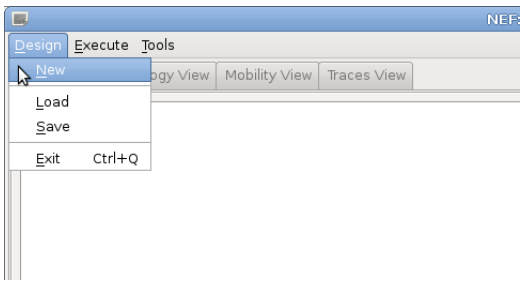

Figure 7.5: Design menu

The *Execute* sub menu provides options to *Start*, *Stop*, and *Shutdown* and experiment.

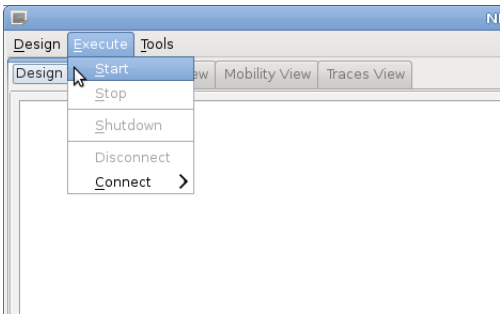

Figure 7.6: Execute menu

When an experiment is launched by clicking the *Start* button in the *Execute* sub menu, NEF will invoke NEPI to deploy the designed experiment. The experiment topology will no longer be modifiable during the experiment execution, however it will still be possible to edit certain box attributes using the attributes edit box, to modify the experiment configuration during runtime. When experiment deployment is finished, two more views will became active. The *Topology* and *Traces* view.

NEF provides experiment examples that can be loaded into the *Design* view using the design sub menu. You will find these examples in the "examples" folder on NEF source directory.

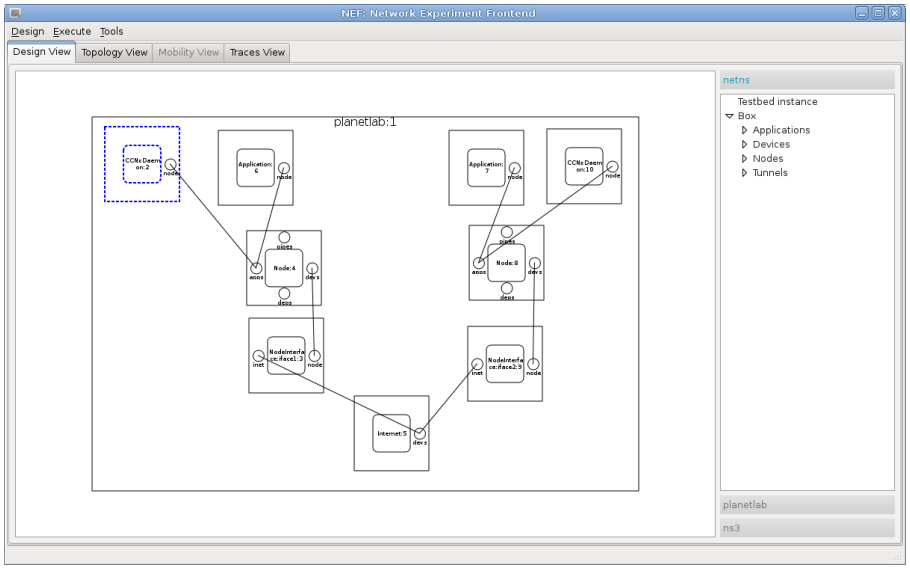

Figure 7.7: CCNx on PlanetLab Europe experiment

The "examples/pl_ccnx_unicast.xml" file provides a basic example to conduct

an experiment using CCNX and PlanetLab Europe. Before trying to run this example, remember to edit the attributes in the PlanetLab *TestbedInstance* to add you PlanetLab Europe slice information.

7.3 Topology view

The *Topology view* shows the runtime topology of the experiment. The tool box on the right allows to visualize and modify attributes for the components that appear in the view.

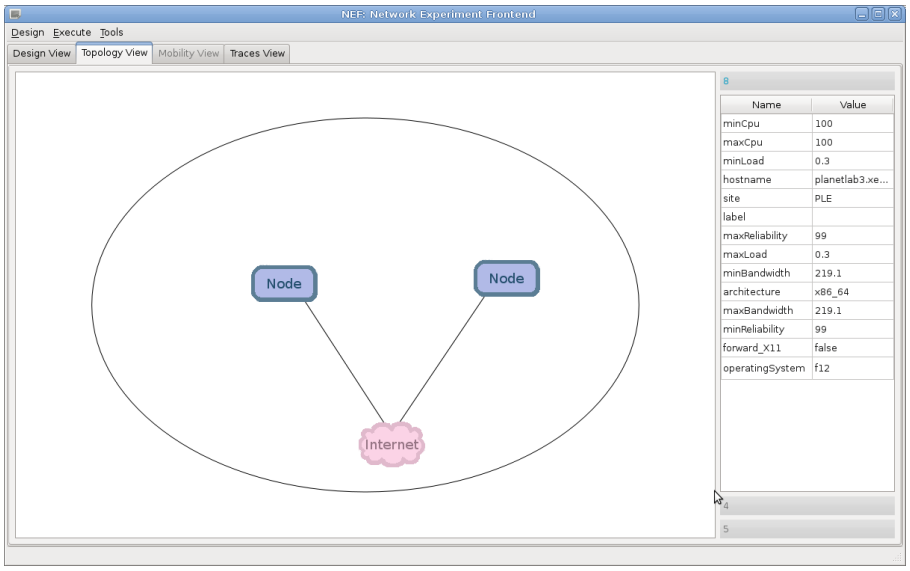

Figure 7.8: Topology view

7.4 Traces view

The *Traces view* shows the list of enabled traces per component. To retrieve traces from a remote location you can select them from the traces list and click on the “Download” button on the bottom.

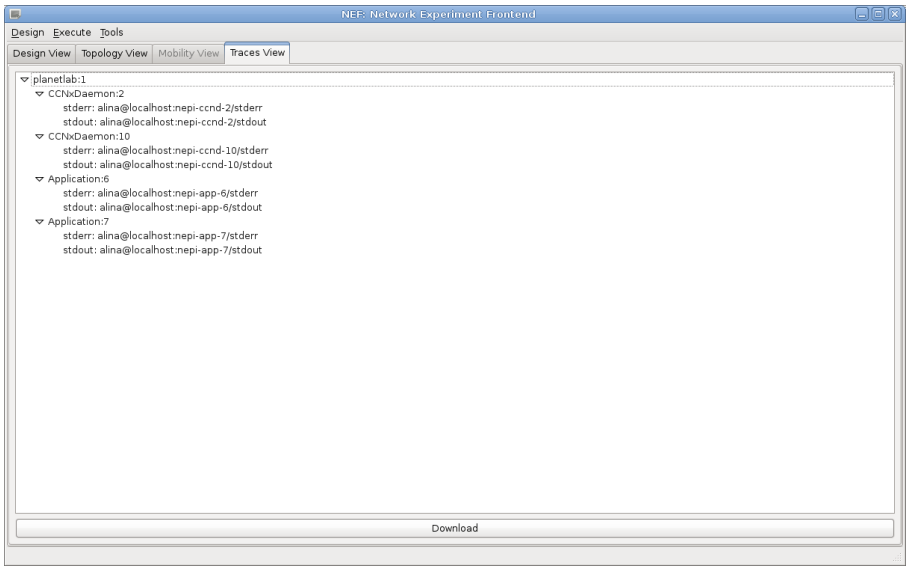

Figure 7.9: Traces view

Bibliography

[1] The netns testbed. `http://nepi.pl.sophia.inria.fr/wiki/netns`.

[2] The ns-3 network simulator. `http://www.nsnam.org/`.

[3] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *NSDI'04*, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association.

[4] Mathieu Lacage, Martin Ferrari, Mads Hansen, Thierry Turletti, and Walid Dabbous. Nepi: using independent simulators, emulators, and testbeds for easy experimentation. *SIGOPS Oper. Syst. Rev.*, 43(4):60–65, 2010.

[5] Alina Quereilhac, Mathieu Lacage, Claudio Freire, Thierry Turletti, and Walid Dabbous. Nepi: An integration framework for network experimentation. *Software, Telecommunications and Computer Networks (SoftCOM)*, 19:1–5, 2011.