NEPI v3.0 User Manual

Contents

1 FAQ

1.1 What is NEPI?

NEPI is not a network simulator, nor an emulator or a testbed. NEPI is a Python library that provides classes to describe and run network experiments on different experimentation platforms (e.g. Planetlab, OMF wireless testbeds, network simulators, etc).

Imagine that you want to run an experiment to test on the Internet a distributed application you just implemented. You can use NEPI to automate deployment of your application on the PlanetLab testbed, run the experiment, and then collect result files to your local machine. NEPI aims at providing a re-usable code base to run network experiments on target experimentation platforms, and in this way decrease the time you spend developing platform-specific scripts and debugging them.

NEPI is a network experiment management framework that provides a simple way of describing network experiments, and the logic to automatically deploy those experiments on the target experimentation environments. It also provides the means to control the resources used in the experiment (e.g. Nodes, applications, switches, virtual machines, routing table entries, etc) during experiment execution, and to collect results generated by the experiment to a local repository.

The experiment deployment and control are done by the ExperimentController (EC), which is the entity responsible for the global orchestration of the experiment. The EC knows nothing about how to manage specific resources (e.g. how to configure a network interface in a PlanetLab node), instead it delegates resource management to platform-specific entities called Resource Manager (RM). The RMs are responsible for controlling resources (e.g. Linux hosts, Open vSwitches on PlanetLab nodes, etc). Different types of resources are be controlled by specific RMs. All RMs implement a same API that the EC uses to communicate with them in a uniform way.

NEPI can not control all existing resources on all existing experimentation platforms by default. However, arbitrary resources can be controlled in NEPI if the

corresponding Resource Manager is implemented for it. Fortunately, NEPI already provides several ResourceManagers to manage different resources on a variety of testbeds, and new Resource Manager classes can be extended from existing ones by the experimenters. The idea behind NEPI is to support running network experiments on potentially any experimentation platform, using a single software tool, instead of using a dedicated software for each platform.

1.2 What does a NEPI script look like?

Here is a very simple experiment example, which runs a PING to "nepi.inria.fr" from a given host. Note that you will need to replace the hostname, username, and ssh_key variables va to run the example.

```
from nepi.execution.ec import ExperimentController

ec = ExperimentController(exp_id="myexperiment")

hostname = # Host that can be accessed with an SSH account
username = # SSH user account on host
ssh_key = # Path to SSH public key file to access host

node = ec.register_resource("linux::Node")
ec.set(node, "hostname", hostname)
ec.set(node, "username", username)
ec.set(node, "identity", ssh_key)

app = ec.register_resource("linux::Application")
ec.set(app, "command", "ping -c3 nepi.inria.fr")
ec.register_connection(app, node)

ec.deploy()

ec.wait_finished(app)

print ec.trace(app, "stdout")

ec.shutdown()
```

1.3 What does NEPI stands for?

It stands for Network Experiment Programming Interface.

1.4 Who developed NEPI?

NEPI was developed at INRIA, Sophia Antipolis France. A first prototype was implemented in 2010. Versions 1.0 and 2.0 were released in 2011 and 2012, respectively. The current NEPI version is 3.0.

The following people have contributed to the project:

- NEPI version 3.0: Alina Quereilhac, Julien Tribino, Lucia Guevgeozian Odizzio, Alexandros Kouvakas
- NEPI versions 1.0 and 2.0: Alina Quereilhac, Claudio Freire, Martin Ferrari, Mathieu Lacage
- NEPI prototype: Martin Ferrari, Mathieu Lacage
- Other contributors: Dirk Hasselbalch

1.5 Is it free?

Yes, NEPI is free software. It is free to use, free to modify, free to share. NEPI v2.0 is licensed under GPL v2, so you can do whatever you want with it, as long as you keep the same license.

1.6 How can I contribute?

There are many ways you can contribute to the project. The first one is using it and reporting bugs. You can report bugs on the NEPI bugzilla page at:

`http://nepi.inria.fr/bugzilla`

You can also become a part of the NEPI community and join our mailing lists:

- To subscribe to the users mailing list at *nepi-users@inria.fr* you can send an email to *sympa@inria.fr* with subject *Subscribe nepi-users <put-your-user-name-here>*
- To subscribe to the developers mailing list at *nepi-developers@inria.fr* you can send an email to *sympa@inria.fr* with subject *Subscribe nepi-developers <put-your-user-name-here>*

To contribute with bug fixes and new features, please send your code patch to the *nepi-developers* list.

1.7 How can I report a bug ?

To report a bug take a look at the NEPI bugzilla page at :

```
http://nepi.inria.fr/bugzilla
```

1.8 Where can I get more information?

For more information visit NEPI web site at:

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

2 Getting started

NEPI is written in Python, so you will need to install Python before being able to run experiments with NEPI. NEPI is known to work on Linux (Fedore, Debian, Ubuntu) and Mac (OS X).

2.1 Dependencies

Dependencies for NEPI vary according to the features you want to enable. Make sure the following dependencies are correctly installed in your system before using NEPI.

Mandatory dependencies:

- Python 2.6+
- Mercurial
- python-ipaddr
- python-networkx
- python-pygraphviz
- python-matplotlib

Optional dependencies:

- SleekXMPP - Required to run experiments on OMF testbeds

Install dependencies on Debian/Ubuntu

```
$ sudo apt-get -y install python mercurial python-ipaddr python-networkx python-pygraphviz pyt
```

Install dependencies on Fedora

```
$ sudo yum -y install python mercurial python-ipaddr python-networkx graphviz-python python-ma
```

Install dependencies on Mac

First install homebrew (`http://mxcl.github.io/homebrew/`), then you can install Python and the rest of the dependencies as follows:

```
$ brew install python
$ sudo port install mercurial
$ sudo easy_install pip
$ sudo pip install ipaddr
$ sudo pip install networkx
$ sudo  pip install pygraphviz
$ sudo  pip install matplotlib
```

To use Python you will need to set the PATH environmen variable as:

```
$ export PATH=$PATH:/usr/local/share/python
```

Install SleekXMPP

You will need *git* to get the SleekXMPP sources.

```
$ git clone -b develop git://github.com/fritzy/SleekXMPP.git
$ cd SleekXMPP
$ sudo python setup.py install
```

2.2 The source code

To get NEPI's source code you will need Mercurial version control system. The Mercurial NEPI repo can also be browsed online at:

`http://nepi.inria.fr/code/nepi/`

Clone the repo

```
$ hg clone http://nepi.inria.fr/code/nepi -r nepi-3.2.0
```

2.3 Install NEPI in your system

You don't need to install NEPI in your system to be able to run experiments. However this might be convenient if you don't plan to modify or extend the sources.

To install NEPI, just run *make install* in the NEPI source folder.

```
$ cd nepi
$ make install
```

If you are developing your own NEPI extensions, the installed NEPI version might interfere with your work. In this case it is probably more convenient to tell Python where to find the NEPI sources, using the PYTHONPATH environmental variable, when you run a NEPI script.

```
$ export PYTHONPATH=$PYTHONPATH:<path-to-nepi>/src
```

2.4 Run experiments

There are two ways you can use NEPI to run your experiments. The first one is writing a Python script, which will import NEPI libraries, and run it. The second one is in interactive mode by using Python console.

Run from script

Writing a simple NEPI expeiment script is easy. Take a look at the example in the FAQ section 1.2. Once you have written down the script, you can run it using Python. If NEPI is not installed in your system, you will need to export the path to NEPI's source code to the PYTHONPATH environment variable, so that Python can find NEPI's libraries.

```
$ export PYTHONPATH=<path-to-nepi>/src:$PYTHONPATH
$ cd <path-to-nepi>
$ python examples/linux/ping.py -a localhost
```

Run NEPI interactively

The IPython console can be used as an interactive interpreter to execute Python instructions. We can take advantage of this feature, to interactively run NEPI experiments. We will use the IPython console for the example below.

You can easily install IPython on Debian, Ubuntu, Fedora or Mac as follows:

Debian/Ubuntu

```
$ sudo apt-get install ipython
```

Fedora

```
$ sudo yum install ipython
```

Mac

```
$ pip install ipython
```

Before starting, make sure to add Python and IPython source directory path to the PYTHONPATH environment variable

```
$ export PYTHONPATH=$PYTHONPATH:/usr/local/lib/python:/usr/local/share/python/ipython
```

Then you can start IPython as follows:

```
$ export PYTHONPATH=<path-to-nepi>/src:$PYTHONPATH
$ ipython
Python 2.7.3 (default, Jan  2 2013, 13:56:14)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
```

If you want to paste many lines at once in IPython, you will need to type *%cpaste* and finish the paste block with .

The first thing we need to do to describe an experiment with NEPI is to import the NEPI Python modules. In particular we need to import the ExperimentController class. To do this type the following in the Python console:

```
from nepi.execution.ec import ExperimentController
```

After importing the ExperimentController class, it is possible to create a new instance of an the ExperimentController (EC) for your experiment. The <exp-id> argument is the name you want to give the experiment to identify it and distinguish it from other experiments.

```
ec = ExperimentController(exp_id = "<your-exp-id>")
```

Next we will define two Python functions: *add_node* and *add_app*. The first one to register linux::Node resources and the second one to register linux::Application resources.

```
%cpaste
def add_node(ec, hostname, username, ssh_key):
```

```
    node = ec.register_resource("linux::Node")
    ec.set(node, "hostname", hostname)
    ec.set(node, "username", username)
    ec.set(node, "identity", ssh_key)
    ec.set(node, "cleanExperiment", True)
    ec.set(node, "cleanProcesses", True)
    return node

def add_app(ec, command, node):
    app = ec.register_resource("linux::Application")
    ec.set(app, "command", command)
    ec.register_connection(app, node)
    return app
--
```

The method *register_resource* registers a resource instance with the Experiment-Controller. The method *register_connection* indicates that two resources will interact during the experiment. Note that invoking *add_node* or *add_app* has no effect other than informing the EC about the resources that will be used during the experiment. The actual deployment of the experiment requires the method *deploy* to be invoked.

The linux::Node resource exposes the hostname, username and identity attributes. This attributes provide information about the SSH credentials needed to log in to the Linux host. The *hostname* is the one that identifies the physical host you want to access during the experiment. The *username* must correspond to a valid account on that host, and the *identity* attribute is the 'absolute' path to the SSH private key in your local computer that allows you to log in to the host.

The *command* attribute of the linux::Application resource expects a BASH command line string to be executed in the remote host. Apart from the *command* attribute, the linux::Application resource exposes several other attributes that allow to upload, compile and install arbitrary sources. The add_app function registers a connection between a linux::Node and a linux::Application.

Lets now use these functions to describe a simple experiment. Choose a host where you have an account, and can access using SSH key authentication.

```
hostname = <the-hostname>
username = <my-username>
identity = </home/myuser/.ssh/id_rsa>

node = add_node(ec, hostname, username, ssh_key)
app = add_app(ec, "ping -c3 nepi.inria.fr",  node)
```

The values returned by the functions add_node and add_app are global unique identifiers (guid) of the resources that were registered with the EC. The guid is used

to reference the ResourceManager associated to a registered resource (for instance to retrieve results or change attribute values).

Now that we have registered some resources, we can ask the ExperimentController (EC) to deploy them. Invoking the *deploy* command will not only configure the resource but also automatically launch the applications.

```
ec.deploy()
```

After some seconds, we should see some output messages informing us about the progress in the host deployment. If you now open another terminal and you connect to the host using SSH (as indicated below), you should see that a directory for your experiment has been created in the host. In the remote host you will see that two NEPI directories were created in the $HOME directory: *nepi-src* and *nepi-exp*. The first one is where NEPI will store files that might be re used by many experiments (e.g. source code, input files) . The second directory *nepi-exp*, is where experiment specific files (e.g. results, deployment scripts) will be stored.

```
$ ssh -i identity username@hostname
```

Inside the *nepi-exp* directory, you will find another directory with the <exp-id> assigned to your EC, and inside that directory you should find one directory named node-1 which will contain the files (e.g. result traces) associated to the LinuxNode reosurce you just deployed. In fact for every resource deployed associated to that host (e.g. each linux::Application), NEPI will create a directory to place files related to it. The name of the directory identifies the type of resources (e.g. 'node', 'app', etc) and it is followed by the global unique identifier (guid).

We can see if a resource finished deploying by querying its state through the EC

```
ec.state(app, hr=True)
```

Once a linux::Application has reached the state 'STARTED', we can retrieve the 'stdout' trace, which should contain the output of the PING command.

```
ec.trace(app, "stdout")
```

That is it. We can terminate the experiment by invoking the method *shutdown*.

```
ec.shutdown()
```

Define a workflow

Now that we have introduced to the basics of NEPI, we will register two more applications and define a workflow where one application will start after the other one has finished executing. For this we will use the EC *register_condition* method described below:

```
register_condition(self, guids1, action, guids2, state, time=None):
    Registers an action START, STOP or DEPLOY for all RM on list
    guids1 to occur at time 'time' after all elements in list guids2
    have reached state 'state'.

        :param guids1: List of guids of RMs subjected to action
        :type guids1: list

        :param action: Action to perform (either START, STOP or DEPLOY)
        :type action: ResourceAction

        :param guids2: List of guids of RMs to we waited for
        :type guids2: list

        :param state: State to wait for on RMs of list guids2 (STARTED,
            STOPPED, etc)
        :type state: ResourceState

        :param time: Time to wait after guids2 has reached status
        :type time: string
```

To use the *register_condition* method we will need to import the ResourceState and the ResourceAction classes

```
from nepi.execution.resource import ResourceState, ResourceAction
```

Then, we register the two applications. The first application will wait for 5 seconds and the create a file in the host called "greetings" with the content "HELLO WORLD". The second application will read the content of the file and output it to standard output. If the file doesn't exist il will instead output the string "FAILED".

```
app1 = add_app(ec, "sleep 5; echo 'HELLO WORLD!' > ~/greetings", node)
app2 = add_app(ec, "cat ~/greetings || echo 'FAILED'", node)
```

In order to guarantee that the second application is successful, we need to make sure that the first application is executed first. For this we register a condition:

```
ec.register_condition (app2, ResourceAction.START, app1, ResourceState.STOPPED)
```

We then deploy the two application:

```
ec.deploy(guids=[app1,app2])
```

Finally, we retrieve the standard output of the second application, which should return the string "HELLO WORLD!".

```
ec.trace(app2, "stdout")
```

3 Introduction to NEPI

During the past decades a wide variety of platforms to conduct network experiments, including simulators, emulators and live testbeds, have been made available to the research community. Some of these platforms are tailored for very specific use cases (e.g. PlanetLab for very realistic Internet application level scenarios), while others support more generic ones (e.g. ns-3 for controllable and repeatable experimentation). Nevertheless, no single platform is able to satisfy all possible scenarios, and so researchers often rely on different platforms to evaluate their ideas.

Given the huge diversity of available platforms, it is to be expected a big disparity in the way to carry out an experiment between one platform and another. Indeed, different platforms provide their own mechanisms to access resources and different tools to conduct experiments. These tools vary widely, for instance, to run a ns-3 simulation it is necessary to write a C++ program, while to conduct an experiment using PlanetLab nodes, one must first provision resources through a special web service, and then connect to the nodes using SSH to launch any applications involved in the experiment.

Mastering such diversity of tools can be a daunting task, but the complexity of conducting network experiments is not only limited to having to master different tools and services. Designing and implementing the programs and scripts to run an experiment can be a time consuming and difficult task, specially if distributed resources need to be synchronised to perform the right action at the right time. Detecting and handling possible errors during experiment execution also posses a challenge, even more when dealing with large size experiments. Additionally, difficulties related to instrumenting the experiment and gathering the results must also be considered.

In this context, the challenges that NEPI addresses are manifold. Firstly, to simplify the complexity of running network experiments. Secondly, to simplify the use of different experimentation platforms, allowing to easily switch from one to another. Thirdly, to simplify the use of resources from different platforms at the same time in a single experiment.

The approach proposed by NEPI consists on exposing a generic API that re-

searchers can use to *program* experiments, and providing the libraries that can execute those experiments on target network experimentation platforms. The API abstracts the researchers from the details required to actually run an experiment on a given platform, while the libraries provide the code to automatically perform the steps necessary to deploy the experiment and manage resources.

The API is generic enough to allow describing potentially any type of experiment, while the architecture of the libraries was designed to be extensible to support arbitrary platforms. A consequence of this is that any new platform can be supported in NEPI without changing the API, in a way that is transparent to the users.

3.1 Experiment Description

NEPI represents experiments as graphs of interconnected resources. A resource is an abstraction of any component that takes part of an experiment and that can be controlled by NEPI. It can be a software or hardware component, it could be a virtual machine, a switch, a remote application process, a sensor node, etc.

Resources in NEPI are described by a set of attributes, traces and connections. The attributes define the configuration of the resource, the traces represent the results that can be collected for that resource during the experiment and the connections represent how a resource relates to other resources in the experiment.

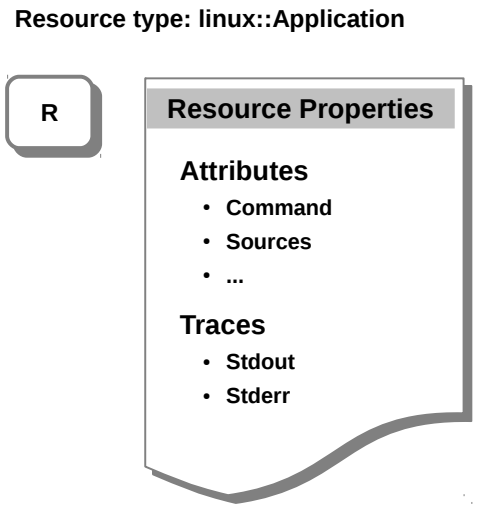

Figure 3.1: Properties of a resource of type LinuxApplication

Examples of attributes are a linux hostname, an IP address to be assigned to a network interface, a command to run as a remote application. Examples of traces

are the standard output or standard error of a running application, a tcpdump on a network interface, etc.

Resources are also associated to a type (e.g. a Linux host, a Tap device on PlanetLab, an application running on a Linux host, etc). Different types of resources expose different attributes and traces and can be connected to other specific types (e.g. A resource representing a wireless channel can have an attribute SSID and be connected to a Linux interface but not directly to a Linux host resource) Figure 3.1 exemplifies this concept.

There are two different types of connections between resources, the first one is used to define the *topology graph* of the experiment. This graph provides information about which resources will interact with which other resources during the experiment (e.g. application A should run in host B, and host B will be connected to wireless channel D through a network interface C). Figure 3.2 shows a representation of the concept of topology graph to describe the an experiment.

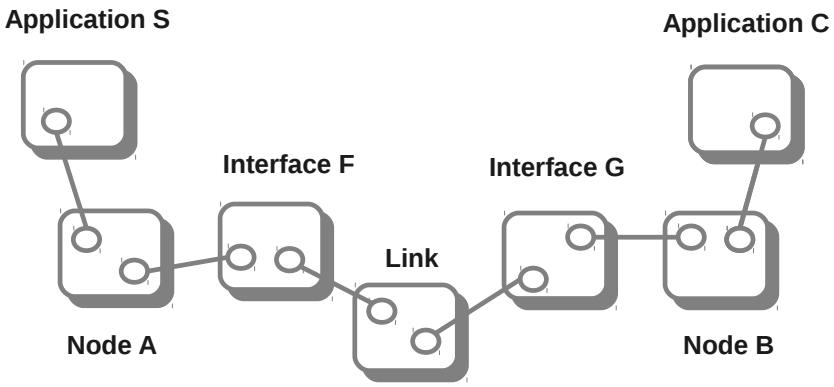

Figure 3.2: A topology graph representation of an abstract experiment

The second type of connections (called conditions to differentiate them from the first type) specifies the *dependencies graph*. This graph is optional and imposes constraints on the experiment workflow, that is the order in which different events occur during the experiment. For instance, as depicted in Figure 3.3 a condition on the experiment could specify that a server application has to start before a client application does, or that an network interface needs to be stopped (go down) at a certain time after the beginning of the experiment.

It is important to note, that the *topology graph* also defines implicit and compulsory workflow constraints (e.g. if an application is *topologically* connected to a host, the host will always need to be up and running before an application can run

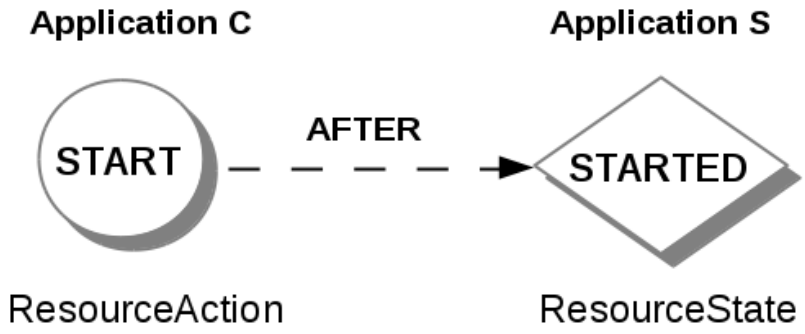

Figure 3.3: A dependencies graph representation involving two applications resources in an experiment

on it). The difference is that the *dependency graph* adds complementary constraints specified by the user, related to the behavior of the experiment.

This technique for modeling experiments is generic enough that can be used to describe experiments involving resources from any experimentation environment (i.e. testbed, simulator, emulator, etc). However, it does not provide by itself any information about how to actually deploy and run an experiment using concrete resources.

3.2 Experiment Life Cycle

The Experiment Description by itself is not enough to conduct an experiment. In order to run an experiment it is necessary to translate the description into concrete actions and to perform these actions on the specific resources taking part of the experiment. NEPI does this for the user in an automated manner.

Given that different resources will require performing actions in different ways (e.g. deploying an application on a Linux machine is different than deploying a mobile wireless robot), NEPI abstracts the life cycle of resources into common stages associated to generic actions, and allows to plug-in different implementation of these actions for different types of resources. Figure 3.4 shows the three main stages of the network experiment life cycle, *Deployment*, *Control* and *Result (collection)*, and the actions that are involved in each of them.

In order to be able to control different types of resources in a uniform way, NEPI assigns a generic state to each of these actions and expects all resources to follow

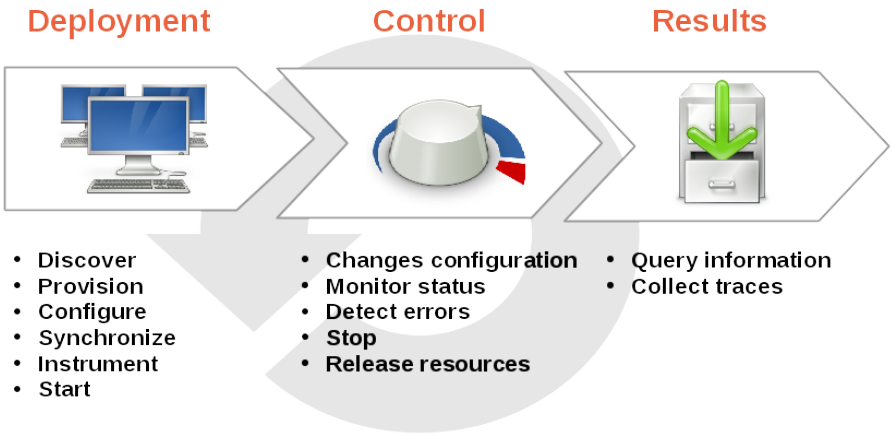

Figure 3.4: Common stages of a network experiment life cycle

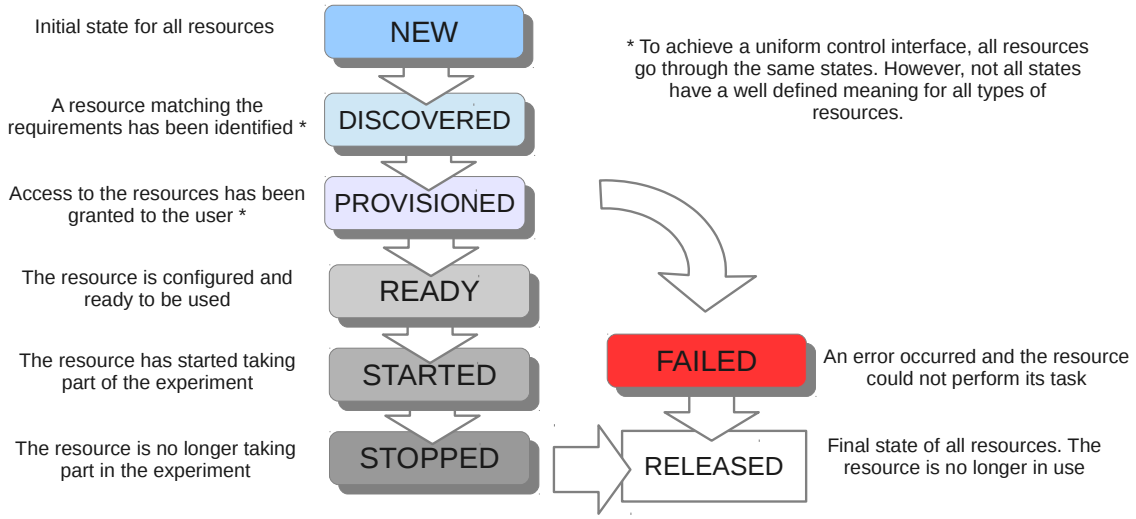

Figure 3.5: Resources state transitions

the same set of state transitions during the experiment life. The states and state transitions are depicted in Figure 3.5.

It is important to note that NEPI does not require these states to be globally synchronized for all resources (e.g. resources are not required to be all ready or started at the same time). NEPI does not even require all resources to be declared and known at the beginning of the experiment, making it possible to use an *interactive deployment* mode, where new resources can de declared and deployed on the fly, according to the experiment needs. This interactive mode can be useful to run

experiments with the purpose of exploring a new technology, or to use NEPI as an adaptive experimentation tool, that could change an experiment according to external conditions or measurements.

3.3 Resource Management: The EC & The RMs

The Experiment Controller (EC) is the entity that is responsible for translating the Experiment Description into a running experiment. It holds the *topology* and *dependencies* graphs, and it exposes a generic experiment control API that the user can invoke to deploy experiments, control resources and collect results.

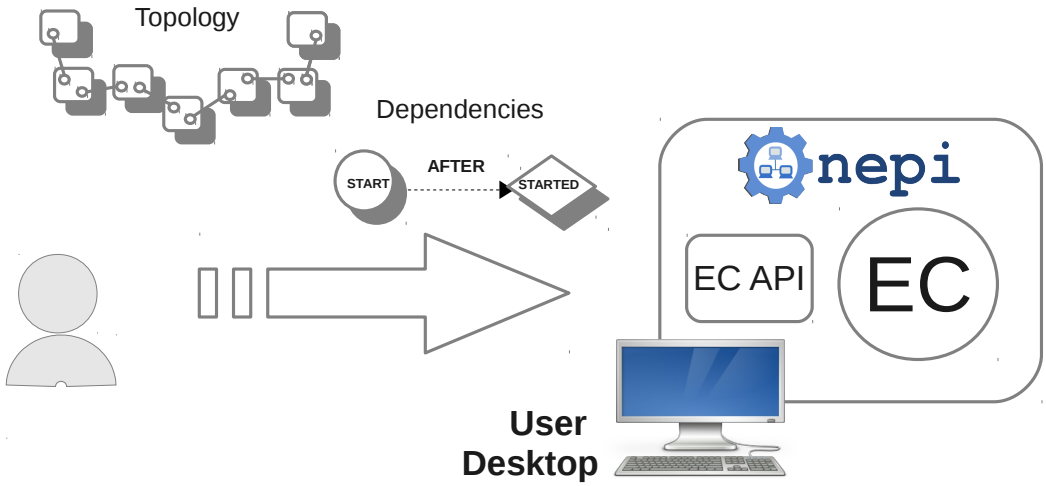

Figure 3.6: User interacting with the Experiment Controller

As shown in Figure 3.6, the user declares the resources and their dependencies directly with the EC. When the user requests the EC to deploy a certain resource or a group of resources, the EC will take care of performing all the necessary actions without further user intervention, including the sequencing of actions to respect user defined and topology specific dependencies, through internal scheduling mechanisms.

The EC is a generic entity responsible for the global orchestration of the experiment. As such, it abstracts itself from the details of how to control concrete resources and relies on other entities called Resource Managers (RM)s to perform resource specific actions.

For each resource that the user registers in the *topology graph*, the EC will instantiate a RM of a corresponding type. A RM is a resource specific controller and

different types of resources require different type of RMs, specifically adapted to manage them.

The EC communicates with the RMs through a well defined API that exposes the necessary methods (actions) to achieve all the state transitions defined by the common resource life-cycle. Each type of RM must provide a specific implementation for each action and ensure that the correct state transition has been achieved for the resource (e.g. upon invocation of the START action, the RM must take the necessary steps to start the resource and set itself to state STARTED). This decoupling between the EC and the RMs makes it possible to extend the control capabilities of NEPI to arbitrary resources, as long as a RM can be implemented to support it.

As an example, a testbed *X* could allow to control host resources using a certain API X, which could be accessed via HTTP, XMLRPC, or via any other protocol. In order to allow NEPI to run experiments using this type of resource, it would suffice to create a new RM of type host X, which extends the common RM API, and implements the API X to manage the resources.

Figure 3.7 illustrates how the user, the EC, the RMs and the resources collaborate together to run an experiment.

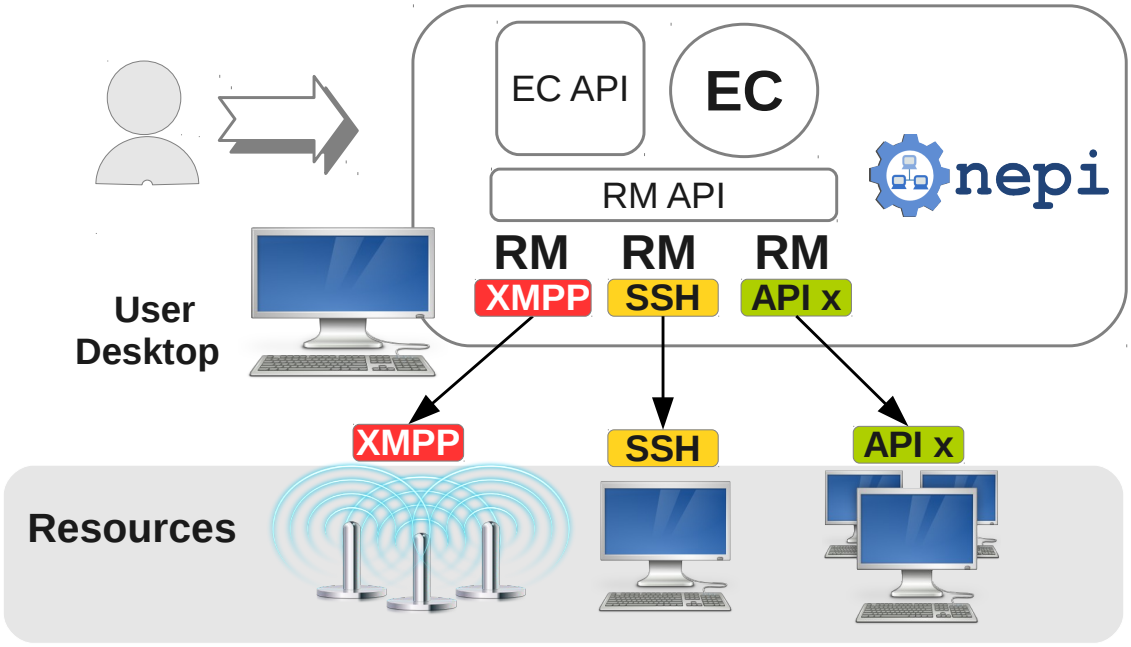

Figure 3.7: Resource management in NEPI

4 The ExperimentController API

The ExperimentController (EC) is the entity in charge of turning the experiment description into a running experiment. In order to do this the EC needs to know which resources are to be used, how they should be configured and how resources relate to one another. To this purpose the EC exposes methods to register resources, specify their configuration, and register dependencies between. These methods are part of the EC design API. Likewise, in order to deploy and control resources, and collect data, the EC exposes another set of methods, which form the execution API. These two APIs are described in detail in the rest of this chapter.

4.1 The experiment script

NEPI is a Python-based language and all classes and functions can be used by importing the *nepi* module from a Python script.

In particular, the ExperimentController class can be imported as follows:

```
from nepi.execution.ec import ExperimentController
```

Once this is done, an ExperimentController must be instantiated for the experiment. The ExperimentController constructor receives the optional argument *exp_id*. This argument is important because it defines the experiment identity and allows to distinguish among different experiments. If an experiment id is not explicitly given, NEPI will automatically generate a unique id for the experiment.

```
ec = ExperimentController(exp_id="my-exp-id")
```

The experiment id can always be retrieved as follows

```
exp_id = ec.exp_id
```

Since a same experiment can be ran more than one time, and this is often desirable to obtain statistical data, the EC identifies different runs of an experiment

with a same *exp_id* with another attribute, the *run_id*. The *run_id* is a timestamp string value, and in combination with the *exp_id*, it allows to uniquely identify an experiment instance.

```
run_id = ec.run_id
```

4.2 The design API

Once an ExperimentController has been instantiated, it is possible to start describing the experiment. The design API is the set of methods which allow to do so.

Registering resources

Every resource supported by NEPI is controlled by a specific ResourceManager (RM). The RM instances are automatically created by the EC, and the user does not need to interact with them directly.

Each type of RM is associated with a *type_id* which uniquely identifies a concrete kind of resource (e.g PlanetLab node, application that runs in a Linux machine, etc). The *type_ids* are string identifiers, and they are required to register a resource with the EC.

To discover all the available RMs and their *type_ids* we can make use of the ResourceFactory class. This class is a *Singleton* that holds the templates and information of all the RMs supported by NEPI. We can retrieve this information as follows:

```
from nepi.execution.resource import ResourceFactory

for type_id in ResourceFactory.resource_types():
    rm_type = ResourceFactory.get_resource_type(type_id)
    print type_id, ":", rm_type.get_help()
```

Once the *type_id* of the resource is known, the registration of a new resource with the EC is simple:

```
type_id = "SomeRMType"
guid = ec.register_resources(type_id)
```

When a resource is registered, the EC instantiates a RM of the requested *type_id* and assigns a global unique identifier (guid) to it. The guid is an incremental integer number and it is the value returned by the *register_resource* method. The EC keeps internal references to all RMs, which the user can reference using the corresponding guid value.

Attributes

ResourceManagers expose the configurable parameters of resources through a list of attributes. An attribute can be seen as a *name:value* pair, that represents a certain aspect of the resource (whether information or configuration information).

It is possible to discover the list of attributes exposed by an RM type as follows:

```
from nepi.execution.resource import ResourceFactory

type_id = "SomeRMType"
rm_type = ResourceFactory.get_resource_type(type_id)

for attr in rm_type.get_attributes():
    print "       ",  attr.name, ":", attr.help
```

To configure or retrieve the value of a certain attribute of an registered resource we can use the *get* and *set* methods of the EC.

```
old_value = ec.get(guid, "attr_name")
ec.set(guid, "attr_name", new_value)
new_value = ec.get(guid, "attr_name")
```

Since each RM type exposes the characteristics of a particular type of resource, it is to be expected that different RMs will have different attributes. However, there a particular attribute that is common to all RMs. This is the *critical* attribute, and it is meant to indicate to the EC how it should behave when a failure occurs during the experiment. The *critical* attribute has a default value of *True*, since all resources are considered critical by default. When this attribute is set to *False* the EC will ignore failures on that resource and carry on with the experiment. Otherwise, the EC will immediately interrupt the experiment.

Traces

A Trace represent a stream of data collected during the experiment and associated to a single resource. ResourceManagers expose a list of traces, which are identified by a name. Particular traces might or might not need activation, since some traces are enabled by default.

It is possible to discover the list of traces exposed by an RM type as follows:

```
from nepi.execution.resource import ResourceFactory

type_id = "SomeRMType"
rm_type = ResourceFactory.get_resource_type(type_id)

for trace in rm_type.get_traces():
```

```
    print "       ",  trace.name, ":", trace.enabled
```

The *enable_trace* method allows to enable a specific trace for a RM instance

```
ec.enable_trace(guid, "trace-name")

print ec.trace_enabled(guid, "trace-name")
```

Registering connections

In order to describe the experiment set-up, a resources need to be associated at least to one another. Through the process of connecting resources the *topology graph* is constructed. A certain application might need to be configured and executed on a certain node, and this must be indicated to the EC by connecting the application RM to the node RM.

Connections are registered using the *register_connection* method, which receives the guids of the two RM.

```
ec.register_connection(node_guid, app_guid)
```

The order in which the guids are given is not important, since the *topology_graph* is not directed, and the corresponding RMs *'know'* internally how to interpret the connection relationship.

Registering conditions

All ResourceMangers must go through the same sequence of state transitions. Associated to those states are the actions that trigger the transitions. As an example, a RM will initially be in the state NEW. When the DEPLOY action is invoked, it will transition to the DISCOVERED, then PROVISIONED, then READY states. Likewise, the action START will make a RM pass from state READY to STARTED, and the action STOP will change a RM from state STARTED to STOPPED.

Using these states and actions, it is possible to specify workflow dependencies between resources. For instance, it would be possible to indicate that one application should start after another application by registering a condition with the EC.

```
from nepi.execution.resource import ResourceState, ResourceActions

ec.register_condition(app1_guid, ResourceAction.START, app2_guid, ResourceState.
    STARTED)
```

The above invocation should be read "Application 1 should START after application 2 has STARTED". It is also possible to indicate a relative time from the moment a state change occurs to the moment the action should be taken as follows:

```
from nepi.execution.resource import ResourceState, ResourceActions

ec.register_condition(app1_guid, ResourceAction.START, app2_guid, ResourceState.
    STARTED, time = "5s")
```

This line should be read "Application 1 should START at least 5 seconds after application 2 has STARTED".

Allowed actions are: DEPLOY, START and STOP.

Existing states are: NEW, DISCOVERED, PROVISIONED, READY, STARTED, STOPPED, FAILED and RELEASED.

4.3 The execution API

After registering all the resources and connections and setting attributes and traces, once the experiment we want to conduct has been described, we can proceed to run it. To this purpose we make use of the *execution* methods exposed by the EC.

Deploying an experiment

Deploying an experiment is very easy, it only requires to invoke the *deploy* method of the EC.

```
ec.deploy()
```

Given the experiment description provided earlier, the EC will take care of automatically performing all necessary actions to discover, provision, configure and start all resources registered in the experiment.

Furthermore, NEPI does not restrict deployment to only one time, it allows to continue to register, connect and configure resources and deploy them at any moment. We call this feature *interactive* or *dynamic* deployment.

The *deploy* method can receive other optional arguments to customize deployment. By default, the EC will deploy all registered RMs that are in state NEW. However, it is possible to specify a subset of resources to be deployed using the *guids* argument.

```
ec.deploy(guids=[guid1, guid2, guid3])
```

Another useful argument of the *deploy* method is *wait_all_ready*. This argument has a default value of *True*, and it is used as a barrier to force the START action to be invoked on all RMs being deploy only after they have all reached the state READY.

```
ec.deploy(wait_all_ready=False)
```

Getting attributes

Attribute values can be retrieved at any moment during the experiment run, using the *get* method. However, not all attributes can be modified after a resource has been deployed. The possibility of changing the value of a certain attribute depends strongly on the RM and on the attribute itself. As an example, once a *hostname* has been specified for a certain Node RM, it might not be possible to change it after deployment.

```
attr_value = ec.get(guid, "attr-name")
```

Attributes have flags that indicate whether their values can be changed and when it is possible to change them (e.g. before or after deployment, or both). These flags are *NoFlags* (the attribute value can be modified always), *ReadOnly* (the attribute value can never be modified), *ExecReadOnly* (the attribute value can only be modified before deployment). The flags of a certain attribute can be validated as shown in the example below, and the value of the attribute can be changed using the *set* method.

```
from nepi.execution.attribute import Flags

attr = ec.get_attribute(guid, "attr-name")

if not attr.has_flag(Flags.ReadOnly):
    ec.set(guid, "attr-name", attr_value)
```

Quering the state

It is possible to query the state of any resource at any moment. The state of a resource is requested using the *state* method. This method receives the optional parameter *hr* to output the state in a *human readable* string format instead of an integer state code.

```
state_id = ec.state(guid)

# Human readable state
state = ec.state(guid, hr = True)
```

Getting traces

After a ResourceManager has been deployed it is possible to get information about the active traces and the trace streams of the generated data using the *trace* method.

Most traces are collected to a file in the host where they are generated, the total trace size and the file path in the (remote) host can be retrieved as follows.

```
from nepi.execution.trace import TraceAttr

path = ec.trace(guid, "trace-name", TraceAttr.PATH)
size = ec.trace(guid, "trace-name", TraceAttr.SIZE)
```

The trace content can be retrieved in a stream, block by block.

```
trace_block = ec.trace(guid, "trace-name", TraceAttr.STREAM, block=1, offset=0)
```

It is also possible to directly retrieve the complete trace content.

```
trace_stream = ec.trace(guid, "trace-name")
```

Using the *trace* method it is easy to collect all traces to the local user machine.

```
for trace in ec.get_traces(guid):
    trace_stream = ec.trace(guid, "trace-name")
    f = open("trace-name", "w")
    f.write(trace_stream)
    f.close()
```

API reference

Further information about classes and method signatures can be found using the Python *help* method. For this inspection work, we recommend to instantiate an ExperimentController from an IPython console. This is an interactive console that allows to dynamically send input to the python interpreter.

If NEPI is not installed in the system, you will need to add the NEPI sources path to the PYTHONPATH environmental variable before invoking *ipython*.

```
$ PYTHONPATH=$PYTHONPATH:src ipython
Python 2.7.3 (default, Jan  2 2013, 13:56:14)
Type "copyright", "credits" or "license" for more information.


IPython 0.13.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: from nepi.execution.ec import ExperimentController

In [2]: ec = ExperimentController(exp_id="test")

In [3]: help(ec.set)
```

The example above will show the following information related to the *set* method of the EC API.

```
Help on method set in module nepi.execution.ec:

set(self, guid, name, value) method of nepi.execution.ec.ExperimentController
    instance
    Modifies the value of the attribute with name 'name' on the RM with guid 'guid
    '.

    :param guid: Guid of the RM
    :type guid: int

    :param name: Name of the attribute
    :type name: str

    :param value: Value of the attribute
```

5 Debbuging

NEPI can be run in debug mode by using the environment variable NEPI_LOGLEVEL to *debug*. By default the NEPI runs in *info* mode.

```
$ export NEPI_LOGLEVEL=info #Info mode
$ export NEPI_LOGLEVEL=debug # Debug mode
```