Automating ns-3 Experimentation in Multi-Host Scenarios

Alina Quereilhac, Damien Saucez, Thierry Turletti, Walid Dabbous

Automating ns-3 Experimentation in Multi-Host Scenarios

Alina Quereilhac, Damien Saucez, Thierry Turletti and Walid Dabbous

Inria Sophia Antipolis, France

{alina.quereilhac,damien.saucez,thierry.turletti,walid.dabbous}@inria.fr

Abstract

ns-3 is a flexible simulator whose capabilities go beyond running purely synthetic simulations in a local desktop. Due to its ability to run unmodified Linux applications, to execute in real time mode, and to exchange traffic with live networks, ns-3 can be combined with live hosts to run distributed simulations or to transparently integrate live and simulated networks. Nevertheless, setting up ns-3 multi-host experiment scenarios might require considerable manual work and advanced system administration skills.

The NEPI experiment management framework is capable of automating deployment, execution, and result collection of experiment scenarios that combine ns-3 with multiple hosts in various ways, reducing the burden of manual scenario setup. In this paper we describe the internals of the NEPI framework that provides support for automation of ns-3 experiments, and demonstrate its usage for ns-3 multi-host scenarios with three example cases: *a*) running parallel simulations on a cluster of hosts, *b*) running distributed simulations spanning multiple hosts, and *c*) integrating live and simulated networks.

1 Introduction

Simulation is a powerful mean of studying networking problems because it gives flexibility to model a wide range of scenarios, it has a low usage and deployment cost, and provides reproducible experimentation. Among existing simulators, ns-3 is a popular network simulator that provides realistic models to mimic the behavior of Internet systems [14]. ns-3 simulations are highly customizable thanks to the modular way of representing networks by interconnecting fine grained application, protocol, and physical layer components. Users can extend ns-3 by adding new modular models, making it possible to study technologies even before they are deployed in live testbed environments or production networks.

The most common way of using ns-3 is by executing self-contained simulation programs that run in a local host, without interacting with the external world. However, ns-3 is capable of supporting other extended uses. In particular, ns-3 provides emulation capabilities to introduce realism in otherwise purely synthetic experiments. Emulation in ns-3 is supported by two main mechanisms: application and protocol emulation using Direct Code Execution (DCE) [16], which allows to run unmodified Linux applications and protocols inside simulated nodes, and hybrid experimentation, using special ns-3 network devices [3], which allows to transparently interconnect simulated and live networks.

Additionally, ns-3 simulations can be parallelized and distributed across several hosts, to speed up the study of ranges of parameters in a networking scenario and to scale the size of the network under evaluation. However, despite the potential of ns-3 to be used for multi-host experimentation, setting up such scenarios entails added costs and difficulties, such as the installation and configuration of ns-3 on multiple hosts, the synchronization of ns-3 instances across hosts, and the collection of results from the distributed host locations. The additional work and the need for advanced system administration skills, e.g., to configure tunnels to communicate distributed simulations, might discourage users from exploiting all the possibilities offered by ns-3.

In this paper we describe the usage of the NEPI framework [9, 13] to automate ns-3 experimentation using multiple hosts. NEPI provides a domain specific language to describe in a uniform way network experiments that may include simulated, emulated, and live components. It automates configuration, deployment, execution, and result collection of experiments across live hosts using a well defined Python API. NEPI also takes care of installing and configuring ns-3 across hosts, of synchronizing simulated and live applications, and of collecting and archiving experimental data in a local repository.

To show how NEPI can be used to automate multi-host experimentation with ns-3, this paper presents three use cases. In the first use case, we utilize NEPI to parallelize ns-3 simulations in a cluster of hosts, varying the number of simulated nodes in each parallel simulation. In the second case, we use NEPI to distribute a simulation across multiple hosts, with each host running a part of the simulated scenario, and to automate interconnection and synchronization of remote simulation instances. In the third and final case, we use NEPI to automate deployment of hybrid experiments, taking advantage of ns-3 special devices to exchange traffic between simulated and live networks.

This paper is organized as follows: Section 2 presents the related work, Section 3 describes the ns-3 simulator and its support for realistic and scalable experimentation, Section 4 introduces the NEPI automation framework, and Section 5 details the usage of NEPI to automate multi-host experimentation with ns-3 for the three described use cases. Finally, Section 6 concludes by providing perspectives on the automation of ns-3 experiments using NEPI.

2 Related Work

Several projects developed around the ns-3 simulator provide enhancements for basic simulator features. Some of them focus on automating operational aspects of ns-3 usage, such as installation and data collection. An example of this is the Bake tool [1] that was created to complement the native build system of ns-3 based on waf [6]. Bake is an integration tool that simplifies building ns-3 projects with complex dependencies in a reproducible way, allowing users to configure which modules to include in the ns-3 build, and automatically resolving installation dependencies. Another example is the Flow Monitor project [8] that provides automated collection and storage of network performance data. Flow Monitor is integrated into ns-3 as a module and provides a simple syntax to define flow probes, to automatically classify simulated traffic into flows, and to collect flow statistics.

While addressing basic usage difficulties is essential to support the user community around ns-3, one of the aspects that distinguishes ns-3 from other simulators is its flexibility to evolve and support beyond-basic simulation scenarios. Developments that aim at improving realism and scalability of ns-3 have been of interest of the ns-3 community for a long time. Examples of this are efforts to take advantage of live hosts to support hybrid experimentation, mixing simulated and live networks, or to parallelize and distribute simulations across multiple hosts. The frameworks for ns-3 founded by the NSF [4] attest for the existing interest to extend ns-3 to automate parallel, distributed, and hybrid experimentation.

The SAFE [12] project makes several advances in this direction. It provides a framework and a dedicated infrastructure to automate parallel execution of ns-3 simulations using parametrizable models and experiment termination detectors for steady-state simulations. The parametrizable models simplify the description of simulations, making ns-3 more accessible to non-experienced users. SAFE also provides automated data processing, data storage, and data visualization functionalities that are complemented by the ns-3 data collection framework (DCF) [11]. The DCF framework extends ns-3 with primitives to instrument simulations, and to collect, process, and aggregate simulated data across hosts.

Other projects take advantage of special features of ns-3, such as the DCE emulation extension [16] and the special ns-3 devices for simulated and live network integration [3], to support parallel, distributed, and hybrid simulations. These solutions go from ad-hoc scripts to interconnect ns-3 instances with virtual machines [2], to full fledged frameworks to distribute simulations over multiple CPUs and hosts. Examples of these solutions are the Distributed Client Extension architecture for hybrid experimentation [7], the MPI ns-3 mod-

ule [10] to distribute ns-3 simulations over multiple cores and hosts using the Message Passing Interface (MPI), and the DNEmu [15] project that builds on top of MPI and DCE to create distributed simulations with live network traffic integration.

Nevertheless, these tools and frameworks to support extended multi-host simulation scenarios in ns-3 target specific cases, e.g., either distributed simulation or parallel simulation, and often require manual installation and configuration of the hosts prior to experimentation.

3 ns-3 Background

ns-3 is a discrete-event network simulator that models Internet systems using a modular approach. Simulated networks are described at the application, protocol, and physical layers by interconnecting modules that represent network components. The central component in a simulation scenario is the node. Nodes are attached to other components, such as applications, protocols, devices, and links, to describe the network and customize simulation behavior. Figure 1 depicts the modular abstraction used to describe simulations in ns-3.

The components used to model a simulation can be independently replaced by others, allowing to easily modify scenarios. For instance, the physical network model used in a scenario can be replaced by substituting the modules that represent network devices and links, without affecting the application or protocol layer components.

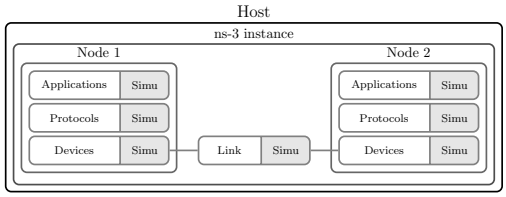

Fig. 1: ns-3 simulations are constructed by interconnecting modular components that model nodes, applications, protocols, devices, and links.

The basic usage of ns-3 consists on running a simulation instance as a program in a host, usually the laptop or desktop of the experimenter. ns-3 simulation programs are written in C++, using modules provided by the ns-3 libraries to describe network scenarios. Simulation programs usually run until termination, without interaction with the experimenter or with external programs, and results are stored in local files. Additionally, ns-3 provides extended usage possibilities involving the use of emulation and multiple simulation instances.

3.1 Emulation Support in ns-3

ns-3 is more than a simulator. It can be transformed into an emulator by adding modules that support emulation at different layers of the network. Support for emulation in ns-3 is provided by two main mechanisms, software-based emulation for application and protocol level emulation, and hybrid emulation by interconnecting an ns-3 simulated network to live networks at the device layer, for device layer emulation.

3.1.1 Application and Protocol Layer Emulation

Application and protocol layer emulation can be added to ns-3 using the Direct Code Execution (DCE) [16] extension. DCE allows to execute unmodified Linux applications and protocols inside ns-3 nodes, making it possible to test the same software used for real networks inside a controlled simulation environment. DCE also supports executing the network stack of the Linux kernel on ns-3 nodes, instead of a simulated stack. Emulation at the application and protocol layers can be added independently from one another. Figure 2 shows the use of DCE at different layers of the network to provide software emulation in ns-3.

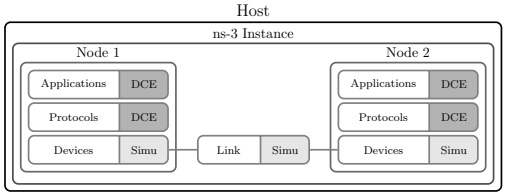

Fig. 2: Application and Protocol Layer Emulation: DCE is used to execute unmodified Linux applications and the Linux protocol stack inside simulated nodes.

3.1.2 Hybrid Emulation

To support interconnection between simulated and live networks, ns-3 provides special devices [3], such as the TapBridge and the FdNetDevice. These devices are capable of sending simulated traffic into a real network device and receiving real traffic from it. By connecting these special device components to ns-3 nodes and attaching them to real devices in the local host, using a local connection, ns-3 is able to exchange traffic between a simulation and the real world. What makes it possible for ns-3 to exchange traffic with live networks is its ability to process events in real time, using the host system's clock, and to generate and consume real Ethernet traffic. Figure 3 depicts the integration between ns-3 and a live host at the ns-3 device layer.

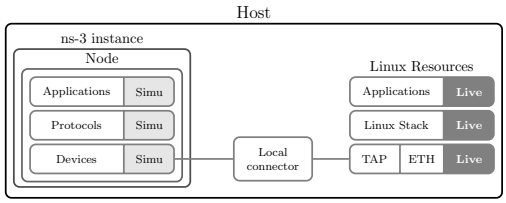

Fig. 3: Hybrid Simulation: ns-3 nodes are interconnected at the device level with live devices in the local host.

3.2 Using Multiple Instances of ns-3

Experimental studies might require running multiple variations of a same scenario or running simulations composed of thousands of hosts. Time constraints and resource limitations might impact the possibility of conducting such studies. Solutions to reduce the duration and increase the scale of ns-3 experiments consist on parallelizing and distributing ns-3 simulations using multiple ns-3 instances.

3.2.1 Parallel Simulation

ns-3 simulations run as independent system processes without requiring interaction with the experimenter or other processes. This allows multiple instances of a same simulation scenario to be executed simultaneously in batch and without interference in a same or different hosts. Executing simulations in parallel is useful to quickly scan ranges of network parameters in a study by assigning a value in the range to each instance. Parallelization can be an efficient approach if the simulation is not greedy on system resources. For greedy simulations, parallelization can be more efficient if multiple hosts are used. Figure 4 shows the execution of multiple parallel instances of ns-3 in a same host.

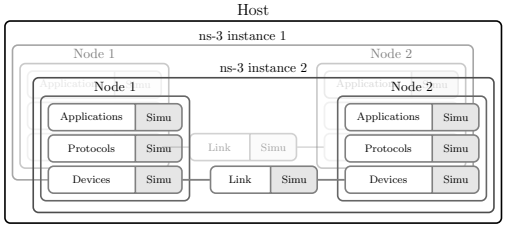

Fig. 4: Parallel Simulation: independent ns-3 instances executed in parallel in the same host to evaluate different parameter values for a same scenario.

3.2.2 Distributed Simulation

Using a single host limits the scale of simulations to the resources available on that host. Distributing a simulation over several CPUs or hosts provides a work-around for scalability limitations. Multiple instances of ns-3 can be interconnected, using a distributed connection mechanism, and synchronized thanks to ns-3 support for distributed simulations. Two alternatives for distributing ns-3 simulations are available: using the MPI module [10] or interconnecting ns-3 instances at the device layer through tunnels or other distributed connectors. Figure 5 shows a simulation divided into two ns-3 instances.

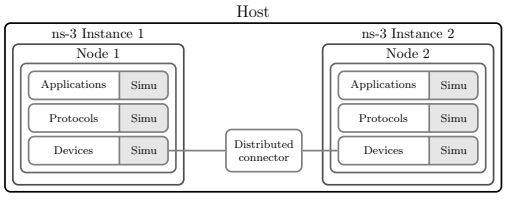

Fig. 5: Distributed Simulation: ns-3 instances interconnected to scale the simulation size.

3.3 Difficulties of Multi-Hosts Scenarios

Using multiple hosts allows to take better advantage of parallelization and distribution of ns-3 instances. Nevertheless, running multiple ns-3 instances across many hosts requires considerable effort since ns-3 by itself does not provide a way to automate configuration and deployment of ns-3 on remote hosts. This means that in order to use multiple hosts, users must manually configure, deploy, and synchronize ns-3 simulations.

In order to simplify the exploitation of ns-3 capabilities for multi-host scenarios, in this paper we propose the use of the NEPI framework to describe experiment scenarios that mix simulated, emulated, and live components and to automate their deployment and execution across multiple hosts.

4 NEPI Automation Framework

NEPI, the Network Experiment Programming Interface, is a framework that generalizes experiment description and experiment life cycle automation to arbitrary network experimentation platforms. Platforms can implement diverse evaluation approaches including simulation, emulation, and live experimentation. NEPI models all elements of an experiment in a same way regardless of the evaluation approach provided by the platforms.

In NEPI, experiments are modeled as a graph of interconnected resources, where a resource can be anything: a live host, a Linux application, a simulated link, etc. Resources have a type that identifies their platform and their function, a state, a list of attributes, and a list of traces.

All resources in a NEPI experiment follow the same life cycle that includes the *deploy*, *start*, *stop*, and *release* operations. During experiment run, these operations are automatically invoked on all resources by the NEPI scheduler. The scheduler orchestrates the experiment and ensures the correct transitions between the different states of a resource, i.e., deploy, start, stopped, respecting the logical dependencies between resources, e.g., an application starts only after the host it runs on is up and running. Users can describe workflows between resources, by adding extra dependencies between resource states and operations. For instance, a user can define a workflow to force a client application to start after a server application started.

Users describe and run experiments in NEPI by writing scripts that use the NEPI Python API. NEPI scripts can be executed from any host, from where local or remote resources, such as hosts, links, applications, simulations, etc, can be accessed and configured. The host running the NEPI script performs the role of *controller host*, but can also be used as another resource in the experimentation. The NEPI API provides design, deployment, monitoring, and data collection primitives that can be used to run experiments interactively or in batch.

The central element in the NEPI architecture is the ExperimentController (EC). An EC object represents an experiment and implements the NEPI API, exposing all the methods users need to design and run experiments. The EC is in charge of automatically orchestrating and monitoring experiments, and of collecting and archiving results.

```
1  from nepi.execution.ec import ExperimentController
2
3  ec = ExperimentController(exp_id="my-experiment")
4
5  node = ec.register_resource("linux::Node")
6  ec.set(node, "hostname", "my-hostname")
7  ec.set(node, "username", "my-user")
8  ec.set(node, "identity", "ssh-key")
9
10 app = ec.register_resource("linux::Application")
11 ec.set(app, "command", "ping -c3 192.168.0.1")
12 ec.register_connection(node, app)
13
14 ec.deploy()
15
16 ec.wait_finished(app)
17
18 print ec.trace(app, "stdout")
19
20 ec.shutdown()
```

Listing 1: NEPI experiment script

Listing 1 shows a basic NEPI experiment that consists on executing the ping command on a Linux host. Line 3 instantiates the ExperimentalController that manages the experiment. Lines 5 to 12 define the experiment scenario, declaring resources with their types, interconnecting them, and specifying their configuration. Line 14 triggers experiment orchestration and line 16 sets a barrier that blocks the user script until the execution of the ping command is finalized. Line 18 retrieves the output of the ping command and line 20 terminates the experiment.

Internally, in order to automate experimentation, the EC instantiates a ResourceManager (RM) object for each resource registered by the user. RM objects are instances of ResourceManager classes

and implement the actions needed to execute life cycle operations for resources on different platforms. A RM class is like a driver that provides a specific implementation of the generic life cycle operations for a particular resource. For example, a RM class that represents a Linux application will implement the deploy operation so that it takes care of installing the application binaries in the host where the application runs, and the start operation so that it executes those binaries in that host. The ability of NEPI to support arbitrary platform resources is based on the extensibility of the ResourceManager class hierarchy and the possibility of providing specific implementations of life cycle operations for different resources.

4.1 ns-3 Support in NEPI

In order to provide support for experimentation with ns-3, NEPI implements ResourceManagers to manage ns-3 components. An experiment that includes ns-3 resources is described by registering, configuring, and connecting ns-3 RMs as shown in Listing 2. In this example an ns-3 simulation resource is registered in line 4 and connected to the local Linux host, in order to execute a simulation in the local host. Then a simulated node and a simulated ping are registered and interconnected in lines 7 to 16. The attribute *enableStack* configures a full simulated stack in the ns-3 node. Line 19 defines a *Collector* RM and connects it to the simulation resource associated to the *stdout* trace. Upon experiment termination, the *Collector* takes care of automatically retrieving experiment results and archiving them in a local directory.

```
1  node = ec.register_resource("linux::Node")
2  ec.set(node, "hostname", "localhost")
3
4  simu = ec.register_resource("linux::ns3::Simulation")
5  ec.register_connection(simu, node)
6
7  nsnode = ec.register_resource("ns3::Node")
8  ec.set(nsnode, "enableStack", True)
9  ec.register_connection(nsnode, simu)
10
11 ping = ec.register_resource("ns3::V4Ping")
12 ec.set (ping, "Remote", "10.0.0.2")
13 ec.set (ping, "Interval", "1s")
14 ec.set (ping, "StartTime", "0s")
15 ec.set (ping, "StopTime", "20s")
16 ec.register_connection(ping, nsnode)
17 ...
18
19 collector = ec.register_resource("Collector")
20 ec.set(collector, "traceName", "stdout")
21 ec.register_connection(collector, simu)
22
23 ec.deploy()
```

Listing 2: Experiment script with ns-3 ResourceManagers

The previous example can be easily extended to describe a simulated network with more nodes, applications, devices, and links by registering and connecting the corresponding ns-3 RMs. DCE applications can be added by registering resources of type *linux::ns3::dce::Application* and connecting them to *ns3::Node* resources. All *ns3::Node* RMs must be connected to exactly one *linux::ns3::Simulation* RM. During the deployment of an ns-3 simulation instance, NEPI takes care of compiling and installing all ns-3 dependencies in the host where the simulation runs, and of synchronizing the start of the simulation with the other resources in the experiment.

4.2 Parallel, Distributed, and Hybrid ns-3 Experiments

Each *linux::ns3::Simulation* RM added to the ExperimentController manages an independent ns-3 instance. By adding many RMs of this type, and connecting them to a same *linux::Node* RM, it is possible to automatically deploy multiple ns-3 instances on a same Linux host. Multiple hosts, with multiple ns-3 instances, can be managed from a single NEPI script running on the controller host.

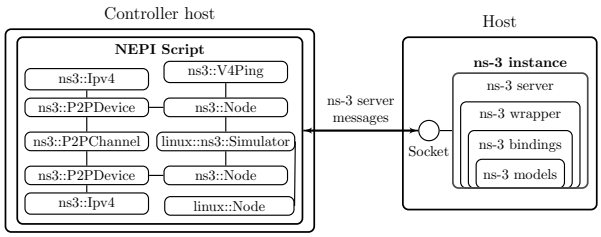

Fig. 6: Communication between NEPI and ns-3 instances.

In order to control independent, and possibly remote, ns-3 instances from a NEPI script, we implemented a *ns-3 server* program. The ns-3 server runs as a standalone process in a local or remote host, and receives messages from NEPI. These messages are part of an ad-hoc protocol and permit to send instructions to create, configure, and connect ns-3 components in a ns-3 instance, as well as to start and stop the simulation.

The ns-3 server is implemented in Python, using an architecture divided in four layers. The innermost layer contains the ns-3 models, i.e., C++ objects provided by ns-3 libraries, the layer on top contains the Python bindings provided by ns-3 to interact with ns-3 C++ objects using Python. The wrapper layer is the core layer of the architecture, it takes care of translating the instructions received from NEPI into actions that modify the simulation. Finally, the outermost layer, the ns-3 server, enables communication by listening for messages in a local Unix socket, and passing those messages to the wrapper layer. The NEPI script connects to the socket to send messages, locally if the simulation runs in the controller host, or using SSH if the simulation runs in a remote host.

Figure 6 shows the interaction between the NEPI script and the ns-3 server instance.

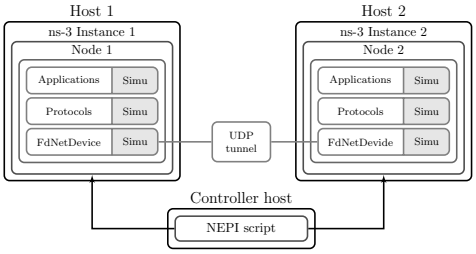

Fig. 7: Distributed ns-3 instances connected through UDP tunnels and FdNetDevices.

To run distributed ns-3 simulations, NEPI provides the possibility of interconnecting ns-3 instances using UDP tunnels. Connections are done at the ns-3 device layer, using the *FdNetDevice* special device of ns-3. A FdNetDevice is associated to a file descriptor, and by reading and writing Ethernet packets from/to the file descriptor it can exchange traffic generated inside an ns-3 simulation with external processes. NEPI provides ResourceManagers to manage FdNetDevice objects as well as UDP tunnels between ns-3 FdNetDevices. Figure 7 shows the interconnection of two distributed ns-3 instances using a UDP tunnel. Other types of tunnels or other techniques to run distributed ns-3 experiments can be used in NEPI, if the corresponding ResourceManagers are implemented to manage each component.

Hybrid experiments can also be described in NEPI by connecting a FdNetDevice inside a simulation with a TAP device in the live host running the simulation. TAP devices are associated to a file descriptor that can be used by a FdNetDevice to write simulated network traffic into the host and to read live traffic from the host. NEPI provides RMs to model TAP devices as well as RMs to model the connection between TAP devices and FdNetDevices. The connector RMs work as virtual local links and take care of passing the file descriptor from the TAP device to the FdNetDevice during experiment deployment. Figure 8 shows an hybrid experiment setup.

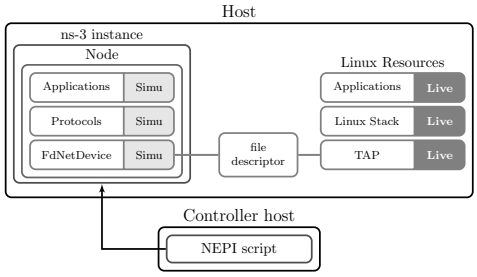

Fig. 8: Hybrid experiments using ns-3 FdNetDevices and Linux TAP devices.

5 Anatomy of a multi-host ns-3 simulation with NEPI

Section 4 explains how NEPI helps in complex ns-3 experiments. In this section, we use a common example to show how to transparently perform parallel and distributed ns-3 simulations over multiple hosts as well as hybrid emulation [1].

To illustrate the three cases, we take as example a wireless sensor network scenario composed of a fixed access point (AP) to which several sensor mobile nodes are connected. Each mobile sensor node periodically transmits a report to the AP, which runs an *agent* to analyze the messages received from the mobile nodes. The AP also plays the role of default gateway for the network. The example is summarized in Figure 9.

[1] The source code of the NEPI scripts used in this section are available in the NEPI repository and can be viewed online at: `http://nepi.inria.fr/code/nepi/file/tip/examples/ns3/multi_host`.

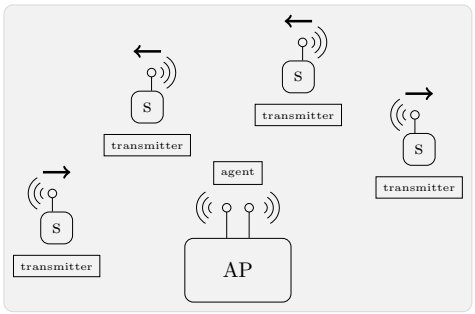

Fig. 9: Example of wireless network scenario. The access point (AP) runs the *agent*, and the mobile sensor nodes (s) run the *transmitter* to send reports to the *agent*.

For the three cases, the anatomy of the NEPI script follows the same structure. First, the script defines the hosts involved in the experiment and specifies that each of them must run an ns-3 simulator instance.[2] In this example, we use hosts from the PlanetLab [5] testbed to run the simulations. PlanetLab is a public research network, with hosts that experimenters can access using a SSH credentials. As we can see in Listing 3, the only configuration information that is needed to manage the host is its name and the SSH credentials of the user, i.e., username and SSH key. Using this information, NEPI can automate installation of ns-3 on the host during deployment and start the simulations without any manual intervention.

```
def add_host_simu(ec, hostname, username, ssh_key):
  host = ec.register_resource("planetlab::Node")
  ec.set(host, "hostname", hostname)
  ec.set(host, "username", username)
  ec.set(host, "identity", ssh_key)

  simu = ec.register_resource("linux::ns3::Simulation")
  ec.register_connection(simu, host)

  return host, simu
```

Listing 3: NEPI function to describe a PlanetLab host with an ns-3 simulator instance running on it.

Second, the script describes the network that will be simulated in the ns-3 instance. As detailed in Listing 4, a wireless channel is defined as transmission medium for the wireless network. An ns-3 node is then created to simulate the AP and is connected to the wireless channel as illustrated in Listing 5. An emulated application to run the agent is then attached to the AP. As Listing 6 shows, it is possible to specify the source code and compilation instructions for the DCE application. NEPI ensures that the application binaries are built and installed on the right path in the host. The script then defines as many mobile nodes as desired, all attached to the wireless channel and running the transmitter application defined in Listing 7. Mobile nodes use the *ns3::RandomDirection2dMobilityModel* to model the mobility pattern. In addition, a default route towards the AP is added to each mobile node (see details in Listing 8).

```
def build_ns3_topology(ec, simu, node_count, network,
    prefixlen, agent_ip):
  channel = ec.register_resource("ns3::YansWifiChannel
    ")

  net = ipaddr.IPv4Network("%s/%s" % (network,
    prefixlen))
  itr = net.iterhosts()

  ap_ip = itr.next().exploded
  ap = add_ns3_node(ec, simu, ap_ip, prefixlen,
    channel, ap_mode=True)

  if ap_ip == agent_ip:
    add_dce_agent(ec, ap)

  for i in range(0, node_count):
    ip = itr.next().exploded
    sensor = add_ns3_node(ec, simu, ip, prefixlen,
     channel, ap_mode=False)
    transmitter = add_dce_transmitter(ec, sensor,
     ap_ip)
    add_ns3_route(ec, sensor, network="0.0.0.0",
     prefixlen="0", nexthop=ap_ip)

  return ap
```

Listing 4: NEPI function to describe an ns-3 simulated network.

```
def add_ns3_node(ec, simu, ip, prefixlen, channel,
    ap_mode=False):
  ns3_node = ec.register_resource("ns3::Node")
  ec.set(ns3_node, "enableStack", True)
  ec.register_connection(ns3_node, simu)

  dev, phy = add_ns3_wifi_device(ec, ns3_node, ip,
    prefixlen, ap_mode)
  ec.register_connection(channel, phy)

  if not ap_mode:
    add_ns3_random_mobility(ec, ns3_node)

  return ns3_node
```

Listing 5: NEPI function to add an ns-3 node in the simulation instance.

```
def add_dce_agent(ec, ns3_node):
  agent = ec.register_resource("linux::ns3::dce::
    Application")
  ec.set(agent, "sources", "code/agent.c")
  ec.set(agent, "build", "gcc -fPIC -pie -rdynamic ${
    SRC}/agent.c -o ${BIN_DCE}/agent")
  ec.set(agent, "binary", "agent")
  ec.register_connection(agent, ns3_node)

  return agent
```

Listing 6: NEPI function to add the DCE agent to an ns-3 node.

[2] For the sake of readability, we omit some details in the code excerpts given in this section, the complete code is available online at the NEPI repository.

```
def add_dce_transmitter(ec, ns3_node, target):
  transmitter = ec.register_resource("linux::ns3::dce
     ::Application")
  ec.set(transmitter, "sources", "code/transmitter.c")
  ec.set(transmitter, "build", "gcc -fPIC -pie -
     rdynamic ${SRC}/transmitter.c -o ${BIN_DCE}/
     transmitter")
  ec.set(transmitter, "binary", "transmitter")
  ec.set(transmitter, "arguments", target)
  ec.register_connection(transmitter, ns3_node)

  return transmitter
```

Listing 7: NEPI function to add the DCE transmitter to an ns-3 node.

```
def add_ns3_route(ec, ns3_node, network, prefixlen,
     nexthop):
  route = ec.register_resource("ns3::Route")
  ec.set(route, "network", network)
  ec.set(route, "prefix", prefixlen)
  ec.set(route, "nexthop", nexthop)
  ec.register_connection(route, ns3_node)

  return route
```

Listing 8: NEPI function to add a route to an ns-3 node.

At this stage, the simulated network is entirely defined. In order to customize a ns-3 instance to allow traffic exchange with another ns-3 instance or with a live network, the FdNetDevice component is used. As shown in Listing 9, NEPI provides a ResourceManager to manage FdNetDevices. In the example, the FdNetDevice is connected to the ns-3 AP node.

```
def add_fdnet_device(ec, ap, ip, prefixlen):
  fddev = ec.register_resource("ns3::FdNetDevice")
  ec.set(fddev, "ip", ip)
  ec.set(fddev, "prefix", prefixlen)
  ec.register_connection(ap, fddev)

  return fddev
```

Listing 9: NEPI function to define an ns-3 FdNetDevice RM and connect it to the AP ns-3 node.

Two remote ns-3 instances can be connected using a tunnel attached to FdNetDevices in each simulated network. To interconnect two FdNetDevice components in different ns-3 instances running on remote PlanetLab hosts, NEPI provides the *planetlab::ns3::FdUdpTunnel* RM shown in Listing 10. This RM creates a UDP tunnel between the PlanetLab hosts and takes care of establishing the interconnection between the FdNetDevice endpoints. As a result, every packet sent on one FdNetDevice is transparently transmitted to the FdNetDevice at the other end of the tunnel, which injects it into its simulation instance. Without NEPI, the configuration and synchronization of the tunnels and the ns-3 instances would have to be done manually.

```
def connect_with_udp_tunnel(ec, fddev1, fddev2):
  tunnel = ec.register_resource("planetlab::ns3::
     FdUdpTunnel")
  ec.register_connection(tunnel, fddev1)
  ec.register_connection(tunnel, fddev2)

  return tunnel
```

Listing 10: NEPI function to define a tunnel between ns-3 instances.

The FdNetDevice can also be connected directly to a TAP device on the local host with a virtual link as shown in Listing 11.

```
def connect_with_virtual_link(ec, tap, fddev):
  link = ec.register_resource("planetlab::ns3::
     TunTapFdLink")
  ec.register_connection(link, tap)
  ec.register_connection(link, fddev)

  return link
```

Listing 11: NEPI function to create a link between an ns-3 simulation and a PlanetLab host.

In the rest of this section, we use the primitives defined above to illustrate the case of parallel simulation in Sec. 5.1, the case of distributed simulation in Sec. 5.2, and the case of hybrid emulation in Sec. 5.3.

5.1 Parallel Simulations on a Cluster of Hosts

It is common to evaluate a same simulated scenario multiple times with small variations, either using different seeds or varying system parameter values. Running the scenarios in parallel minimizes the total time required for the study. When several hosts are available, it is desirable to take advantage of their computational power and parallelize the execution of the simulations on all hosts.

In our example, we wish to evaluate how varying the number of mobile nodes that send messages affects the performance of the system. For this, we execute the same scenario using 10, 50, 100, and 1000 mobile nodes using a pool of available hosts from the PlanetLab testbed. Listing 12, shows the main lines of the NEPI script to run parallel instances of ns-3 using 4 hosts. For the sake of clarity, only one simulation is executed per host, but it is possible to run multiple simulations on each host.

```
ec = ExperimentController(exp_id="parallel")
counts = [10, 50, 100, 1000]
hosts = ["host1", "host2", "host3", "host4"]

for hostname in hosts:
```

```
6   host, simu = add_host_simu(ec, hostname, username,
        ssh_key)
7
8   node_count = counts.pop()
9   build_ns3_topology(ec, simu, node_count, network="
        10.1.0.0", prefixlen="24", agent_ip="10.1.0.1")
10
11 ec.deploy()
```

Listing 12: NEPI script for parallel simulations on a cluster of hosts.

5.2 Distributed Simulation on Multiple Hosts

In the event that the number of mobile nodes in the scenario is too large, the simulation can be scaled up by distributing it over several hosts. NEPI can do this as explained in Section 4, by interconnecting ns-3 instances through UDP tunnels. To illustrate this case, we split the network of our example into two wireless networks, each running as a separate ns-3 instance on a different PlanetLab host. For simplicity of the example, and to show how a same topology defined in NEPI can be re-used multiple times, we use the same *build_ns3_topology* function to define the two networks. Each wireless network is modeled as in the previous example, with an AP that functions as the default IP gateway for the network, with its own IP prefix. In this scenario, the two AP nodes are interconnected through FdNetDevices using a UDP tunnel as shown in Listing 13. Only the AP with IP address *10.1.0.1* runs the *agent* application. An additional route is added to each AP to enable packet forwarding between the simulated networks via the UDP tunnel. As mentioned before, NEPI deploys this setup automatically, without the need of manual intervention.

```
1  ec = ExperimentController(exp_id="distributed")
2
3  host1, simu1 = add_host_simu(ec, hostname1, username,
       ssh_key)
4  ap1 = build_ns3_topology(ec, simu1, node_count,
       network="10.1.0.0", prefixlen="24", agent_ip="
       10.1.0.1")
5
6  host2, simu2 = add_host_simu(ec, hostname2, username,
       ssh_key)
7  ap2 = build_ns3_topology(ec, simu2, node_count,
       network="10.2.0.0", prefixlen="24", agent_ip="
       10.1.0.1")
8
9  fddev1 = add_fdnet_device(ec, ap1, "10.0.0.1", "30")
10 fddev2 = add_fdnet_device(ec, ap2, "10.0.0.2", "30")
11
12 connect_with_udp_tunnel(ec, fddev1, fddev2)
13
14 add_ns3_route(ec, ap1, network="10.2.0.0", prefixlen="
       24", nexthop="10.0.0.2")
15 add_ns3_route(ec, ap2, network="10.1.0.0", prefixlen="
       24", nexthop="10.0.0.1")
16
17 ec.deploy()
```

Listing 13: NEPI script for distributed simulation using two hosts.

5.3 Integration of Live and Simulated Networks

In some situations, it is not possible or desirable to rely only on simulation, e.g., when sensors take part in the experiment. Section 4 explains how NEPI can be used to describe experiments that combine ns-3 simulated networks with live networks. To illustrate how to implement such scenario with NEPI, we use the wireless sensor network scenario from the previous examples, but this time running one of the sensor transmitter applications directly on the live host instead of inside the simulation. In order to exchange traffic between the application on the live host and the simulation, we use the FdNetDevice, connecting it to another remote FdNetDevice instead of to a local TAP device.

Listing 14, shows the main lines of the NEPI script for this scenario. The simulated network is described as in the previous examples, but now the ns-3 AP is connected to a FdNetDevice RM and the host is connected to a TAP device RM, and both devices are connected to a local virtual link RM. The connection with the local virtual link RM of type *planetlab::ns3::TunTapFdLink* is shown in Listing 11.

Finally, routing entries are added to the AP and the host to permit packet routing between the simulated and the live networks. A transmitter application RM is also defined and connected to the live host, to send traffic to the ns-3 AP from outside the simulation. It is worth noticing that as DCE is used in ns-3, the same application code is executed both in the live and simulated parts of the network.

When the *deploy* method in line 25 is executed, NEPI installs and launches the ns-3 instance in the host, it creates the TAP device and connects it to the simulation, and only when all components are ready, it starts the simulation and the transmitter application in the host.

```
1  ec = ExperimentController(exp_id="hybrid")
2
3  host, simu = add_host_simu(ec, hostname, username,
       ssh_key)
4  ap = build_ns3_topology(ec, simu, node_count, network=
       "192.168.3.0", prefixlen="25", agent_ip="
       192.168.3.1")
5
6  fddev = add_fdnet_device(ec, ap, "192.168.3.129", "25"
       )
7
8  tap = ec.register_resource("planetlab::Tap")
9  ec.set(tap, "ip", "192.168.3.130")
10 ec.set(tap, "prefix", "25")
11 ec.set(dev, "pointopoint", "192.168.3.129")
12 ec.register_connection(host, tap)
13
14 connect_with_virtual_link(ec, tap, fddev)
```

```
15
16 add_ns3_route(ec, ap, network="192.168.3.128",
       prefixlen="25", nexthop="192.168.3.1")
17 add_planetlab_route(ec, tap, network="192.168.3.0",
       prefixlen="25", nexthop="192.168.3.129")
18
19 transmitter = ec.register_resource("linux::Application
       ")
20 ec.set(transmitter, "sources", "code/transmitter.c")
21 ec.set(transmitter, "build", "gcc ${SRC}/transmitter.c
       -o ${BIN}/transmitter")
22 ec.set(transmitter, "command", "${BIN}/transmitter
       192.168.3.1")
23 ec.register_connection(transmitter, host)
24
25 ec.deploy()
```

Listing 14: NEPI script for connect an ns-3 simulation to a live host.

```
1 def add_planetlab_route(ec, dev, network, prefixlen,
      nexthop):
2   route = ec.register_resource("planetlab::Vroute")
3   ec.set(route, "network", network)
4   ec.set(route, "prefix", prefixlen)
5   ec.set(route, "nexthop", nexthop)
6   ec.register_connection(route, dev)
```

Listing 15: NEPI function to add a route to PlanetLab node.

6 Conclusion

ns-3 is a flexible simulator that can be used both for pure simulation and for emulation. Thanks to its advanced features, like real time execution and real Ethernet traffic generation, and its special network devices, it is possible to execute ns-3 simulations in parallel, distributed, or interconnected with live networks. Using ns-3 in combination with multiple hosts permits to take the most advantage of these possibilities. Nevertheless, ns-3 does not provide a way of automating experimentation with multiple hosts. Deploying ns-3 in multiple hosts and synchronizing their execution requires considerable manual work and advanced administration skills.

In this paper we show how to use NEPI to simplify the design and automate the deployment of ns-3 simulations in multi-host environments. NEPI permits to simplify the many complex technical aspects that are necessary to interconnect multiple ns-3 instances together. However, it does not hide the technical details of the ns-3 simulator architecture. This means that users must understand how to model ns-3 scenarios in order to use NEPI to automate ns-3 experimentation. What NEPI provides is the means to focus on the scenario rather than on how to run it. Moreover, because NEPI is a generic framework to automate experimentation, it relies on extensive hosts and simulation state verifications, so running experiments with NEPI might take longer than running a local simulation in the traditional way. This factor has to be compared with the fact that using NEPI makes the experimentation process more rigorous and less prone to bugs since it replaces ad-hoc manual steps by automated and reproducible mechanisms.

Simulators and emulators change with time to fit the needs of research communities. Therefore, the ns-3 models we have presented in this paper might be outdated in the coming years. Nevertheless while these models can be outdated, the fundamental concepts in NEPI, i.e., the generic experiment programming interface and comprehensive experiment life-cycle, are designed to be adaptable to support arbitrary ns-3 models and even other experimentation platforms, including other simulators, emulators, and live testbeds.

NEPI scripts can be shared with other researchers to reproduce and verify experiments. The NEPI scripts to reproduce the examples presented in this paper are available online at `http://nepi.inria.fr/code/nepi/file/tip/examples/ns3/multi_host`, for anyone to use and modify.

References

[1] Bake. `http://www.nsnam.org/docs/bake/tutorial/html/bake-over.html`.

[2] ns-3 linux containers setup. `https://www.nsnam.org/wiki/HOWTO_Use_Linux_Containers_to_set_up_virtual_networks`.

[3] ns-3 special devices. `http://www.nsnam.org/docs/models/html/emulation-overview.html`.

[4] NSF Frameworks for ns-3. `http://www.eg.bucknell.edu/~perrone/research-docs/NSFProjectDescription.pdf`.

[5] PlanetLab testbed. `http://planet-lab.eu`.

[6] Waf. `http://code.google.com/p/waf/`.

[7] A. Alvarez, R. Orea, S. Cabrero, X. G. Pañeda, R. García, and D. Melendi. Limitations of Network Emulation with Single-machine and Distributed Ns-3. In *SIMUTools '10*, pages 67:1–67:9, 2010.

[8] G. Carneiro, P. Fortuna, and M. Ricardo. FlowMonitor: A Network Monitoring Framework for the Network Simulator 3 (ns-3). In *VALUETOOLS '09*, pages 1:1–1:10, 2009.

[9] M. Lacage, M. Ferrari, M. Hansen, T. Turletti, and W. Dabbous. NEPI: using independent simulators, emulators, and testbeds for easy experimentation. *ACM SIGOPS Operating Systems Review*, 43(4):60–65, 2010.

[10] J. Pelkey and G. Riley. Distributed Simulation with MPI in ns-3. In *SIMUTools '11*, pages 410–414, 2011.

[11] L. F. Perrone, T. R. Henderson, Felizardo, V. D., and M. J. Watrous. The Design of an Output Data Collection Framework for ns-3. In *WSC '13*, 2013.

[12] L. F. Perrone, C. S. Main, and B. C. Ward. SAFE: Simulation Automation Framework for Experiments. In *WSC '12*, 2012.

[13] A. Quereilhac, M. Lacage, C. Freire, T. Turletti, and W. Dabbous. NEPI: An integration framework for network experimentation. In *SoftCOM '11*, pages 1–5, 2011.

[14] G. F. Riley and T. R. Henderson. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*, pages 15–34. 2010.

[15] H. Tazaki and H. Asaeda. Dnemu: Design and implementation of distributed network emulation for smooth experimentation control. In *Testbeds and Research Infrastructure. Development of Networks and Communities*, pages 162–177. Springer, 2012.

[16] H. Tazaki, F. Urbani, and T. Turletti. DCE cradle: simulate network protocols with real stacks for better realism. In *SIMUTools '13*, pages 153–158, 2013.