CORE and ns-3 Integration

Craig Dowell and Tom Henderson
University of Washington
June 2010

Abstract

This report describes the results of an effort to prototype the integration of the ns-3 discrete-event network simulator with the Common Open Research Emulator (CORE). The goal of this integration is to produce a combined tool that allows Linux network namespaces (with a high degree of implementation realism and potential for application and kernel software reuse) to be supported by high-fidelity ns-3 wireless models. The report first describes a few example use cases illustrating different levels of integration of the two tools. One such use case involves integrating CORE with ns-3 at the CORE graphical user interface (GUI) level, and a proof-of-concept prototype of this use case is next described, based on connecting an ns-3 ad-hoc WiFi real-time simulation with the Linux namespace containers available in CORE. The report next describes caveats and limitations of the integration that would need to be addressed in a final version of the integrated tool, and performs some basic performance benchmarking of the approach. The report concludes by outlining suggestions for future work.

1. Introduction

This report documents the results of an effort to improve the usability and technical soundness of a hybrid simulation and emulation tool based on the ns-3 discrete-event network simulator and the CORE network emulator. ns-3 is a discrete-event network simulator with a few capabilities that make it amenable to integration with a real-time network emulator. First, the simulator can operate in a real-time scheduling mode in which the ns-3 simulation clock is bound to the host machine clock. Second, ns-3 features a TapBridge network device model that allows it to read from character-based Tap devices on the host operating system. Third, ns-3 packets are represented as byte buffers in network-byte order, making serialization to and deserialization from real network devices straightforward.

The Common Open Research Emulator (CORE) has been developed by Boeing Research & Technology as an evolution of the IMUNES network emulator developed by Marko Zec for FreeBSD. CORE reuses and extends the same GUI as IMUNES but has replaced much of the underlying system plumbing by migrating the tool to Linux network namespaces and the use of Linux bridging, netem, and (more recently) the EMANE mobile network emulator to provide link effects.

The strength of CORE is in its high fidelity modeling of Linux computers at the kernel (TCP/IP) and above layers, including the ability to run real applications such as quagga routing daemons. However, the underlying wireless effects are not emulated with high fidelity such as modeling interference effects. Meanwhile, ns-3 has a sophisticated WiFi model and also models for other wireless technologies such as WiMAX and LTE (under development).

This report documents an effort to prototype a hybrid CORE/ns-3 tool, and is organized as follows. Section 2 describes the goals of this work and outlines a few possible use cases for integration. Section 3 describes the software architecture of the use case that was prototyped under this effort. Section 4 walks through an example use of this prototype, from a user's perspective. Section 5 outlines some limitations and caveats discovered in creating the prototype, and Section 6 provides some basic performance benchmarking of the prototype. Finally, Section 7 recommends some directions for future work.

1.1 Related work

This work is based primarily on the integration of ns-3 and CORE. ns-3 is an open-source, discrete-event network simulator targeted primarily for research and educational use [ns3]. The Common Open Research Emulator is an open source network emulator based on lightweight virtual machines, a networking subsystem such as Linux bridging or FreeBSD netgraph, and a graphical user interface [CORE]. A related project is netns3, which is a set of Python objects integrated with a RPC framework to wrap instances of Linux network namespaces with ns3; netns3 and CORE share some similar software internals [netns3].

The Extendable Mobile Ad-Hoc Network Emulator (EMANE) is an open source network emulator focused on the emulation of link and physical layer connectivity [EMANE]. Previous work has extended

CORE to allow it to use EMANE as a networking subsystem, and the work described herein for ns-3 integration with CORE parallels the approach performed for EMANE.

The Network Experiment Programming Interface (NEPI) is a related framework project that ties together simulators, emulators, and testbeds [NEPI]. NEPI is a python library that acts as a controller for a number of backends including ns-3 and, in the future, testbeds such as PlanetLab, Emulab, and ORBIT. NEPI resembles the portions of CORE that act as a controller for backends such as Linux namespaces, FreeBSD jails, and networking subsystems such as EMANE or ns-3, and some design choices appear to be similar to those made for netns3, although NEPI creates more general abstractions representing multiple disparate backends than do the other projects mentioned herein (CORE, netns3, or EMANE).

2. Integration Goals

Broadly, the goal is to develop a hybrid environment that supports the following general workflow:

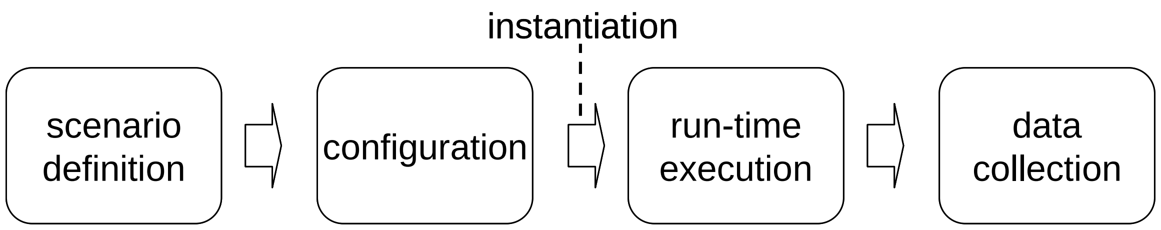


Figure 2-1. General workflow.

At each stage depicted in Figure 2-1, there are a few options. Below we describe the workflows that we intend to prioritize.

2.1 Use case 1: CORE without ns-3

As a first step, it is helpful to review how CORE works in the absence of ns-3. For this description, we refer to the latest development version of ns-3 (ns-3.8) and a development snapshot of CORE (scheduled to be CORE-4.0), with Linux namespaces.

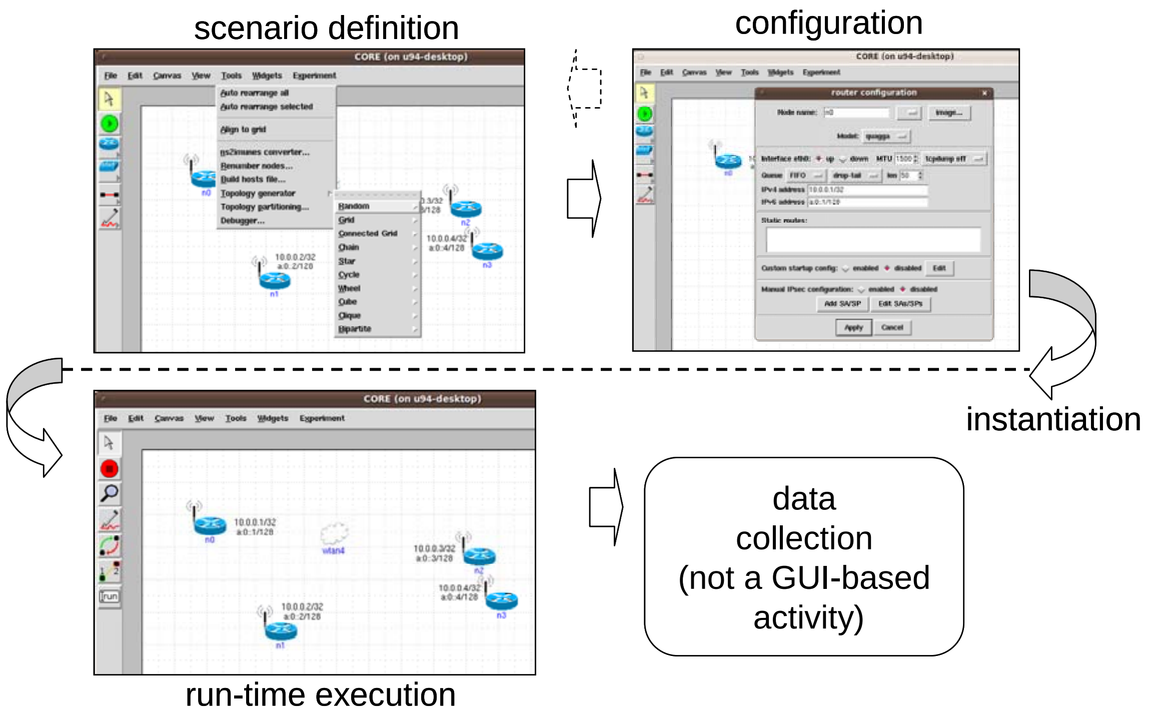


Figure 2-2. Typical CORE workflow.

Figure 2-2 illustrates a typical CORE workflow. The first stage is typically GUI driven, in which users lay out a network topology via a drag-and-drop interface, or using stock topology generators available from a pull-down menu. In this stage, IP address tools automatically assign addresses to links when links are connected to nodes. Scenario descriptions may also be saved and loaded from a CORE-specific "imn" file format. In this stage, all state is held within the Tk-based GUI. Following this topology generation stage, users may do a limited amount of fine-grained configuration on the node or link objects. Users can move back and forth between these stages as needed, as depicted by the dashed backwards arrow in Figure 2-1. Once the scenario is defined and configured, the user presses the green button, and the GUI issues a number of messages to the CORE services daemon, which acts on the messages to instantiate the required virtual machines and network emulation. Visual clues are provided to the user to alert the user to the time in which the emulation is fully instantiated (which may take several seconds). At this point, the user may interact with the GUI to use various widgets that display real-time state of the containers or processes within, may run GUI-driven utility tools such as ping or traceroute, and may change the node placement on the canvas. Finally, the user stops the emulation, and is reponsible for harvesting, archiving, and post-processing any data generated (this is not a GUI-based stage).

There exist some near-term plans to run CORE without a GUI, or allow the detachment or creation of execution GUIs on a running CORE instance, but we do not cover those future enhancements here.

2.2 Use case 2: Full CORE and ns-3 integration

Referring to Figure 2-2 above, a full integration of ns-3 and CORE from a user's perspective might work as follows. In the scenario definition stage, users are able to add (either as new palette objects or as a special "plug in" on an existing Wifi cloud object) ns-3 representations of these network elements, and to mix and match Linux netem and ns-3 channel models. In the configuration stage, right clicking on a node, link, or cloud object that has ns-3 backing would open up access to the ns-3 object attributes. The ns-3 ConfigStore could be used to manage global default variables, and the saving of configuration and scenario information could include both the CORE and the ns-3 attribute information. There probably would need to be some additional GUI (menu or object) to configure ns-3 attributes that aren't associated with a particular object displayed on the canvas.

Next we discuss the transition from configuration to execution (the instantiation phase). An open issue is whether there needs to be an additional stage in the CORE GUI to allow users to hand-edit and fine tune the configuration generated from CORE GUI. This has implications in the software design because it moves the interaction between the GUI and the CORE daemon from more of a message-driven API (the CORE API) to more of a script-driven (since the user will want to be presented with some kind of script and then the CORE services layer will need to consume and parse the script).

In the execution phase, a desirable goal is to allow ns-3-based mobility (mobility objects aggregated to the ghost nodes) to be reflected to the GUI layer. One issue with this is that it would need to be defined whether the GUI or the ns-3 mobility model took precedence if the node's position is being driven both by an ns-3 mobility model and by user interactions on the canvas. Other open issues are how much run-time state of the ns-3 real-time simulation would be visible in the GUI.

Finally, we do not propose as part of this current work any changes to the data collection (non-GUI) aspects, but we note that data collection and run-time execution management are within the scope of the new ns-3 Frameworks project and could be addressed there.

2.3 Use case 3: ns-3 and namespace integration with a CORE execution GUI

Many power users will be constrained by only having a subset of configuration options available with a CORE configuration and scenario building GUI (since the full exposure of the ns-3 configuration API to the CORE GUI will likely be impractical in the near term). For such users, it would be nice to be able to write an ns-3-like Python program and hand-configure it (or import it via ns-3 scenario generation tools), and then launch the GUI to control the instantiation and execution phases.

The ns-3 program could be made to look like a CORE GUI, from the perspective of the CORE service daemon. This type of interaction would be strongly message-based. The message passing could be abstracted from the ns-3 program by some intermediate Python objects that converted ns-3 program statements into CORE API calls. How best to architect this option needs some further discussion.

Similar to CORE without ns-3, it would be strongly preferred to allow the CORE execution GUI to be optional in this workflow, and to allow a GUI to be attached and detached.

2.4 Use case 4: ns-3 and namespace integration with no GUI

Some users may not care about even attaching the execution GUI at all. In this case, the resulting code and workflow will resemble Tom Goff's netns3 framework, and there will be little need for a CORE daemon. It is an open issue how much this use case can be harmonized with the previous use case, in which the CORE daemon is running and manages state such that a user may connect the execution GUI.

2.5 Discussion

Another big-picture consideration is whether and how to adopt a NEPI-like abstraction that would allow the ns-3-like program to configure testbed machines, and not Linux containers, as an additional option.

We have decided in this project to prototype an example of use case 2 (Full CORE and ns-3 integration) to identify issues with the integration, and to identify development priorities. In so doing, we discovered that the support of all of the above use cases will be a lot of work and will have implications on the overall software design, including on CORE. In the remaining sections, we describe our prototype, how well it performs, and directions for future work.

3. Software Architecture

The CORE ns-3 architecture closely parallels the CORE EMANE architecture and is based on the same underlying Python implementation of the CORE daemon. The ns-3 implementation uses the same CORE configuration and position messages as the EMANE implementation, and also uses the same plugin model architecture. Both systems provide built-in support for their respective plugins through managers added to the Python CORE daemon.

Two primary Python classes implement the built-in, device-independent functions of the CORE ns-3 plugin support. These are the Ns3Manager and Ns3Node classes. Conceptually, the Ns3Manager provides the overall control and orchestration of the parts of the simulation that require ns-3 implementations. It is the Ns3Manager, for example, that configures the real-time ns-3 simulator and spins up a thread for the ns-3 simulator event loop.

The Ns3Manager holds references to Ns3Node objects that represent communication channels with their associated ghost nodes and devices. From the perspective of the CORE GUI, the Ns3Node object represents a wireless LAN (the wireless cloud) that has been configured to use one of the ns-3 plugins via the GUI.

In addition to the two device independent classes mentioned above, there must be a class that implements a specific kind of ns-3 topology. These classes are derived from the Ns3Model Python class. There would be a new Ns3Model-derived class implemented to, for example, implement an ad-hoc mobile network simulation. Another separate Ns3Model-derived class might be implemented to provide a WiMax network simulation.

The current implementation provides a single Ns3WifiAdhocModel class that provides a (CORE GUI) wireless cloud using the ns-3 WifiChannel and WifiNetdevice classes along with a 3-dimensional constant position mobility model. The fact that the mobility model is static does not mean that node positions cannot be changed, it only means that the ns-3 simulation will not move nodes automatically. The position of ns-3 nodes may be changed by moving their associated bitmaps on the GUI.

Figure 3-1 shows the relationship of the different CORE ns-3 and ns-3 objects. CORE objects to support ns-3 functions are shown as boxes and ns-3 objects are shown as ovals.

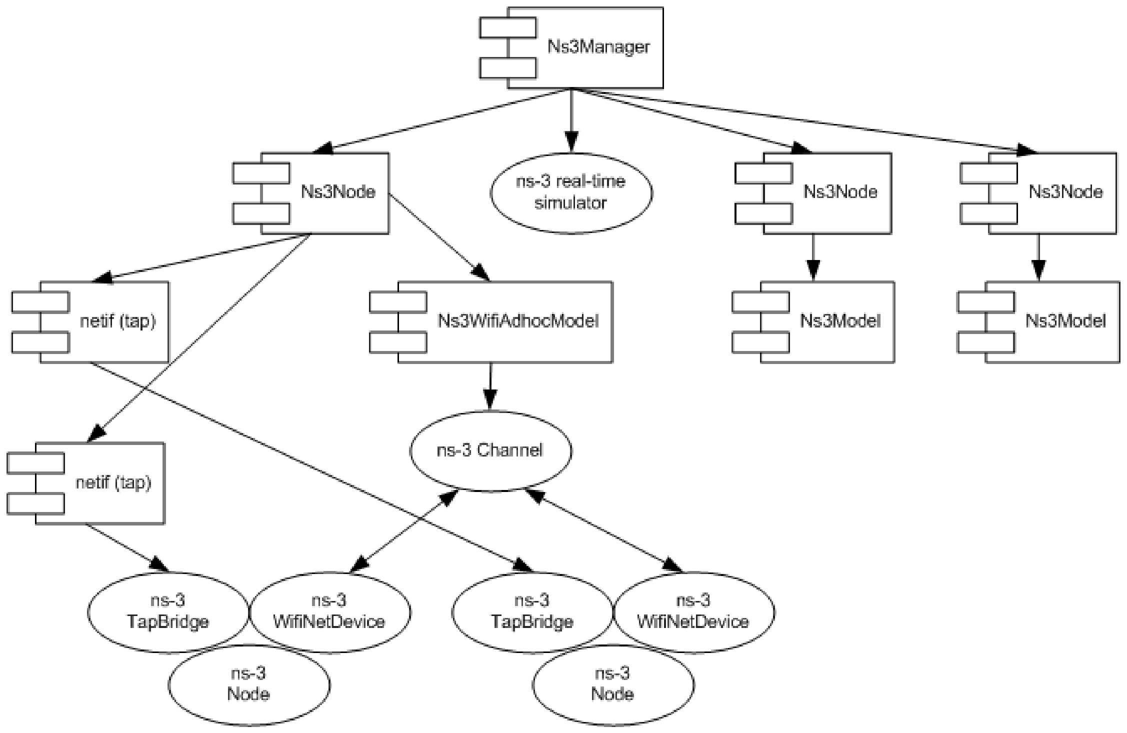


Figure 3-1. CORE and ns-3 object relationships

There is a single Ns3Manager class instantiated in the CORE daemon (cored.py). Every time a CORE wireless cloud is created on the GUI and configured as a ns-3 plugin, a new Ns3Node object is caused to be created in the CORE daemon to represent the implied wireless network. By associating, for example, the ns3_wifiadhoc model with the created Ns3Node (using the CORE GUI configure dialog), a new Ns3WifiAdhocModel object is created and associated with the Ns3Node representing the network. This model object knows how to make the ns-3 objects implied by the configuration. As the Ns3Node on the CORE GUI is associated with other nodes on the canvas (router pillboxes, for example), an association is made between a CORE netif object (which represents a tun/tap device on a linux container) and an implied ghost node on in the ns-3 simulation.

When the CORE simulation is started, the Ns3Manager loops through all of its associated Ns3Nodes asking that the underlying network infrastructure be instantiated. Each Ns3Node instantiates its respective ns-3 channel and then loops through all of its netif associations. For each of these netifs (tun/tap devices) , a call is made into the Ns3Model to create an underlying ns-3 ghost node, mobility model, net device and tap bridge. The tap bridge is associated with the tun/tap device provided by the netif, and the net device is associated with the ns-3 channel managed by the Ns3Node.

Once the entire ns-3 topology is instantiated, the Ns3Manager runs the ns-3 realtime simulator providing a stop time of one year in the future and pushes the netifs into their respective containers. The ns-3 simulation runs in the context of the cored.py CORE daemon.

The ns-3-dependent models are communicated to the system by way of the core.conf file (usually installed in /usr/local/etc/core/core.conf) in the [cored.py] section. A new item, ns3_models, has been defined that indicates which model plugins are available to the Ns3Manager. The name of the item is mapped to a Python file which is expected to provide the actual implementation. For example, adding the following lines to core.conf,

```
[cored.py]
ns3_models = WifiAdhoc
```

will enable ns-3-based wireless ad-hoc simulations. In order to form the plugin name, the model name is downshifted and prepended with “ns3_”. In the case of the WifiAdhoc model above, the GUI will provide the opportunity to load a plugin named “ns3_wifiadhoc”. The CORE daemon separately takes this string and appends “.py” for form a Python module name. In this case it would be “ns3_wifiadhoc.py”. This file is loaded and a class name is constructed by prepending “Ns3” and appending “Model” to the model name. The class Ns3WifiAdhocModel must then be defined in the Python module and must inherit from class Ns3Model. It is this loaded Python module that provides the network implementation-dependent code to the Ns3Manager and Ns3Node.

The loaded model provides and handles configuration exchanges with the CORE GUI and also provides methods such as add_channel, add_node, add_device and setposition that are intended to capture all of the model-specific ns-3 script code.

4. Prototype Walk-Through

The following steps demonstrate the current capabilities of this prototype, as tested on an Ubuntu-9.10 Linux machine (2.6.31-17 kernel, 32 bit system). We assume that core-ns3 and ns-3-allinone from ns-3.8 or later are each unpacked to a home directory.

First, build CORE

```
cd core-ns3
./bootstrap.sh
./configure
make
sudo make install
```

Next, configure and build ns-3.

```
cd ~/ns-allinone-3.8/ns-3.8/
./waf configure --enable-sudo
```

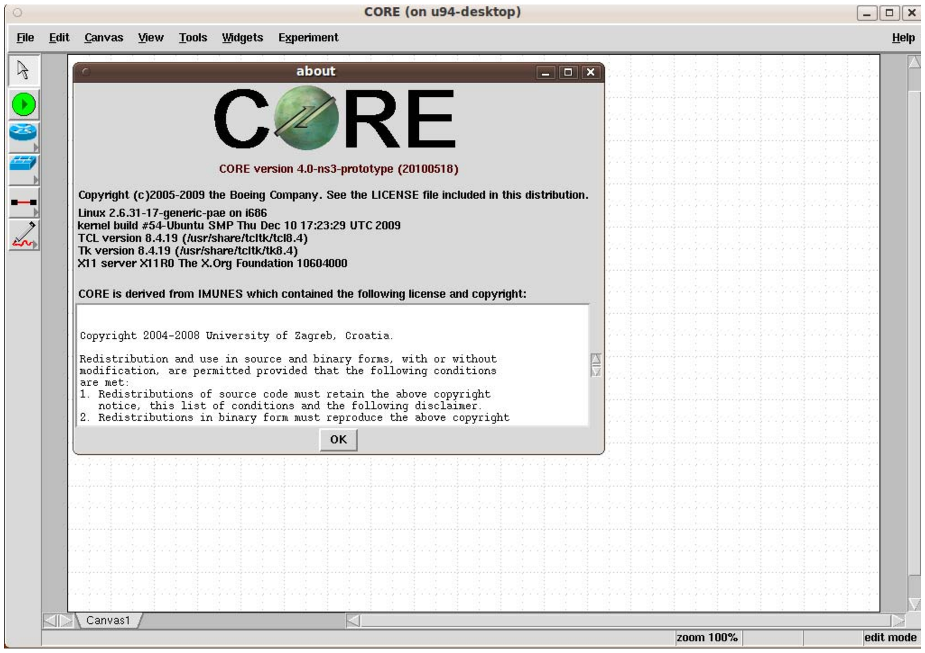
Make sure that Python Bindings and Real-Time Scheduler are enabled, and that “Use sudo to set suid bit” is enabled. Next, build ns-3.

```
./waf
sudo ./waf shell
```

Now from within the waf shell (which knows where the ns-3 libraries are), start CORE:

```
cd ~/core-ns-3/netns/pycore
./cored.py
```

Move to another shell, and type “core” . The first time after installing CORE that you run the “core” GUI executable, it is strongly recommended to run it as root, since some additional configuration files are installed by the GUI. However, subsequent executions of the core GUI do not need to be run as root.. Under Help/About, the version should clearly say “ns3-prototype” as follows:



CORE version 4.0-ns3-prototype (20100518)

Copyright (c)2005-2009 the Boeing Company. See the LICENSE file included in this distribution.
Linux 2.6.31-17-generic-pae on i686
kernel build #54-Ubuntu SMP Thu Dec 10 17:23:29 UTC 2009
TCL version 8.4.19 (/usr/share/tcltk/tcl8.4)
Tk version 8.4.19 (/usr/share/tcltk/tk8.4)
X11 server X11R0 The X.Org Foundation 10604000

CORE is derived from IMUNES which contained the following license and copyright:

```
Copyright 2004-2008 University of Zagreb, Croatia.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
```

Figure 4-1. Display CORE version information.

Next, configure a few nodes on the canvas, and one wireless cloud. Right click on the cloud and select "Link to all routers." Figure 4-2. shows a typical result.

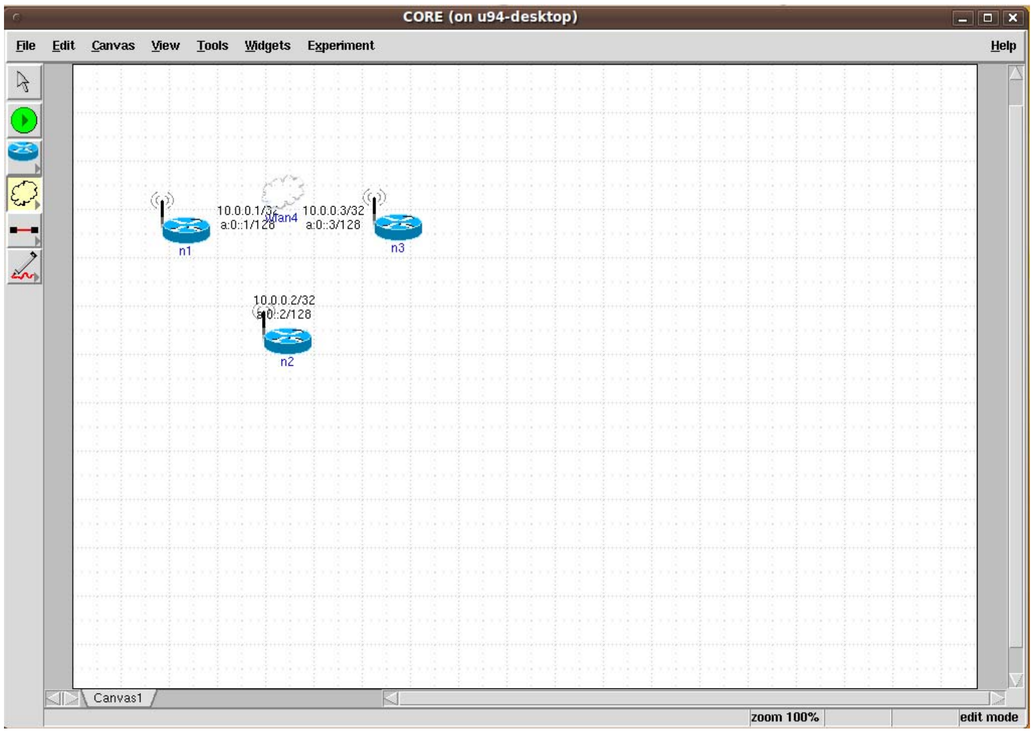
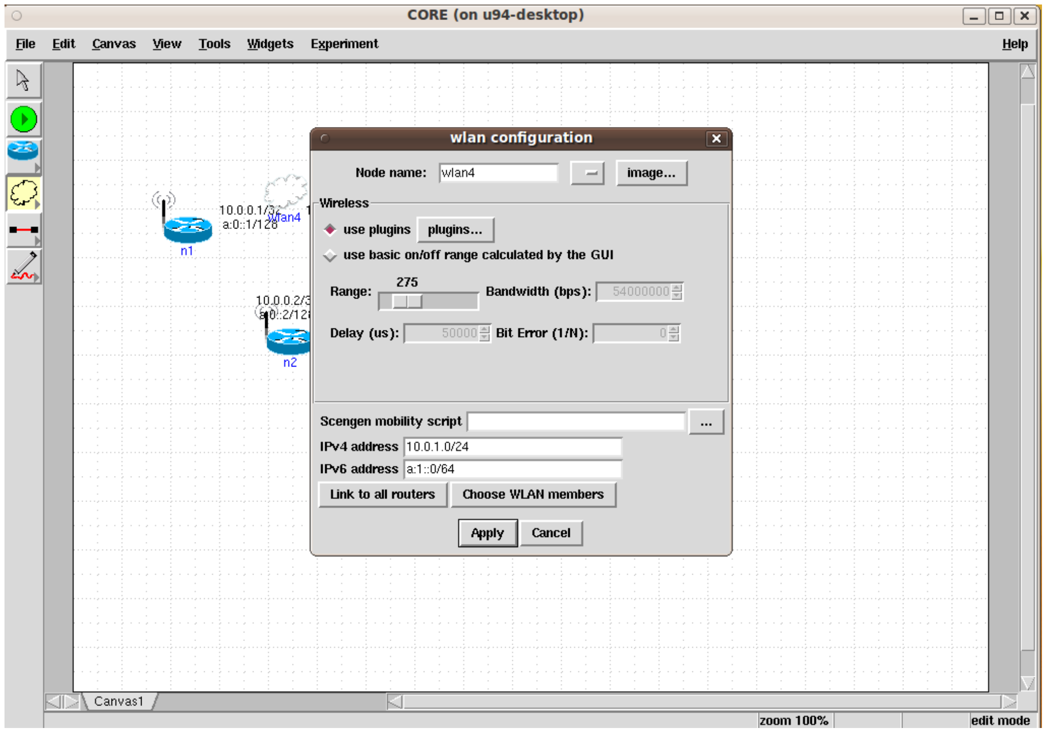


Figure 4-2. Three nodes configured with addresses.

Next, select "Configure" on the wireless cloud, which should show an option to select Plugins, as shown in Figure 4-3.



Node name: wlan4

Wireless

[✓ use plugins]

use basic on/off range calculated by the GUI

Range: 275

Bandwidth (bps): 54000000

Delay (us): 50000

Bit Error (1/N): 0

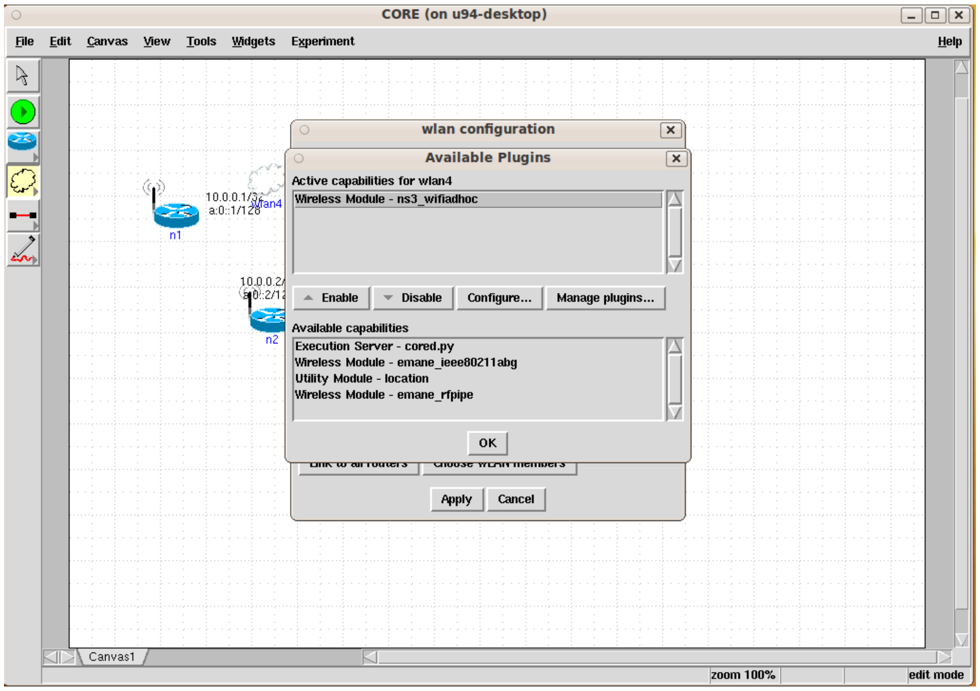
Scengen mobility script

IPv4 address 10.0.1.0/24

IPv6 address a:1::0/64

Figure 4-3. WLAN plugins dialog.

Select plugins, and move to the next menu. One should see an ns3_wifiadhoc plugin among the available ones. Select "Enable" to enable this, as shown in Figure 4-4.



Active capabilities for wlan4
Wireless Module - ns3_wifiadhoc

Available capabilities
Execution Server - cored.py
Wireless Module - emane_ieee80211abg
Utility Module - location
Wireless Module - emane_rfpipe

Figure 4-4. ns3_wifiadhoc plugin selected.

In Figure 4-4, you must next select the "Configure" button after enabling the capability. Failing to do so will result in a later run-time error ("AttributeError: 'NoneType' object has no attribute 'add_channel'). You will see a dialog with a number of ns-3 attribute values. These can be ignored as they have no effect on the underlying simulation. Exit from the three levels of configuration menus by selecting "Apply", "OK", and "Apply".

Now, CORE is ready to run. Select the green execution button. To see whether OSPF adjacencies are formed over the ns-3-enabled topology, select the Observer->Adjacency widget as shown in Figure 4-5. This highlights one of the main benefits of having the CORE execution GUI displaying the emulation (the built-in support for execution widgets that operate on the containers).

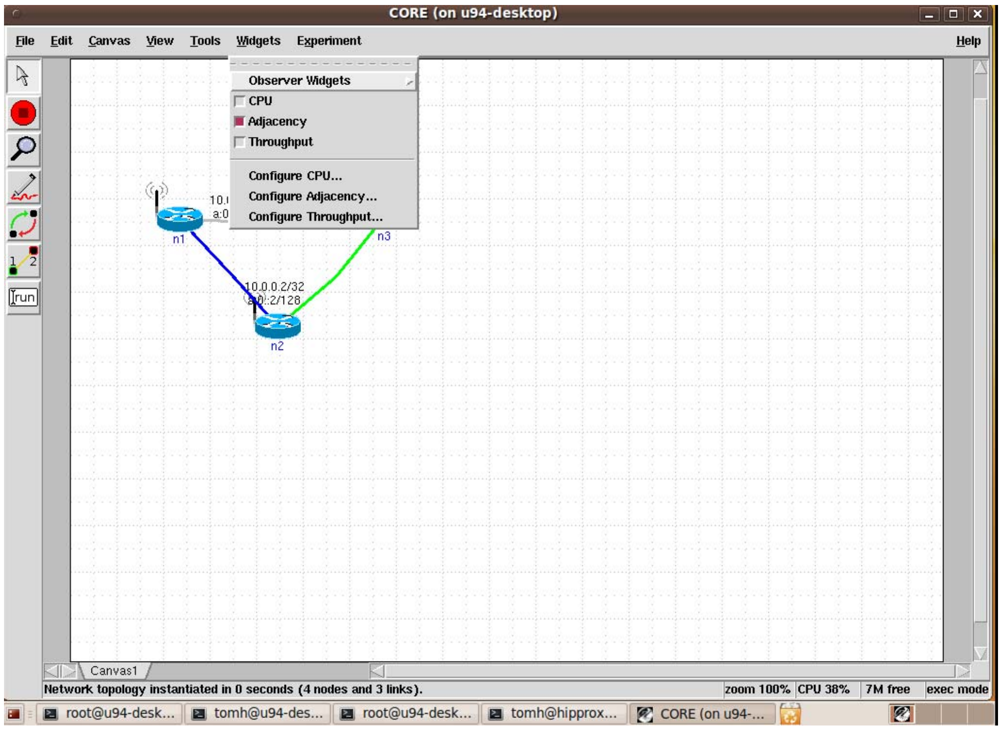
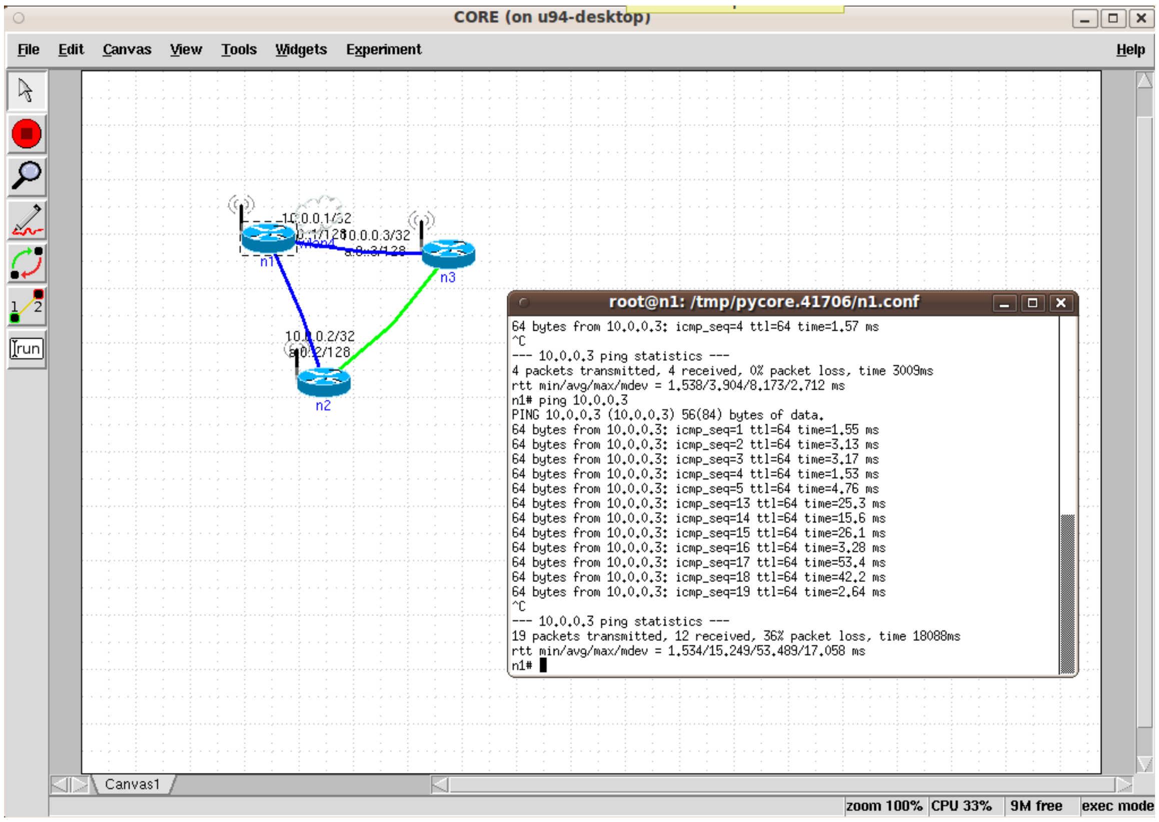


Figure 4-5. CORE adjacency widget.

The next screenshot shows the result of double clicking on one of the nodes to get a shell, starting a ping to another node, and dragging the node out of and back into radio range. There is a gap in the ICMP sequence numbers shown in Figure 4-6.



```
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=1.57 ms
^C
--- 10.0.0.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3009ms
rtt min/avg/max/mdev = 1.538/3.904/8.173/2.712 ms
n1# ping 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=1.55 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=3.13 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=3.17 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=1.53 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=4.76 ms
64 bytes from 10.0.0.3: icmp_seq=13 ttl=64 time=25.3 ms
64 bytes from 10.0.0.3: icmp_seq=14 ttl=64 time=15.6 ms
64 bytes from 10.0.0.3: icmp_seq=15 ttl=64 time=26.1 ms
64 bytes from 10.0.0.3: icmp_seq=16 ttl=64 time=3.28 ms
64 bytes from 10.0.0.3: icmp_seq=17 ttl=64 time=53.4 ms
64 bytes from 10.0.0.3: icmp_seq=18 ttl=64 time=42.2 ms
64 bytes from 10.0.0.3: icmp_seq=19 ttl=64 time=2.64 ms
^C
--- 10.0.0.3 ping statistics ---
19 packets transmitted, 12 received, 36% packet loss, time 18088ms
rtt min/avg/max/mdev = 1.534/15.249/53.489/17.058 ms
n1# 
```

Figure 4-6. Radio connectivity affected by GUI movement.

If one were to look at the output of cored.py during this demo, something like the following would be shown.

```
ns3/netns/pycore# ./cored.py
CORE Python daemon v.1.6 started Tue May 18 14:21:37 2010
starting server: localhost:4038
new connection: 127.0.0.1:41706
ns3_mgr __init__ <ns3_mgr.Ns3Manager object at 0x9fefb8c> <pycore.PyCore object at
0x9fefb4c>
modelfile = ns3_wifiadhoc
clsname = Ns3WifiAdhocModel
importcmd = from ns3_wifiadhoc import Ns3WifiAdhocModel
registering 'cored.py' as emulation and execution server
ns3_mgr configure_ns3 <ns3_mgr.Ns3Manager object at 0x9fefb8c> <pycore.PyCore object
at 0x9fefb4c> CoreConfMessage <msgtype = CORE_API_CONF_MSG, flags = 0x0 <>>
  CORE_TLV_CONF_NODE: 4
  CORE_TLV_CONF_OBJ: all
  CORE_TLV_CONF_TYPE: 3
EVENT 2: CORE_EVENT_CONFIGURATION_STATE at Tue May 18 14:25:23 2010
location configured: (0.00,0.00,0.00) = (47.57917,-122.13232,2.00000) scale=150.00
location configured: UTM(565249.28792,5269892.80917,2.00000)
Ns3Node __init__ <ns3_nodes.Ns3Node object at 0x9ff254c> <pycore.PyCore object at
0x9fefb4c> 4 wlan4 False
Ns3Node attach <ns3_nodes.Ns3Node object at 0x9ff254c> <vnode.TunTap object at
0x9ff2aac>
append ifname n1.0.72 node <ns3.Node object at 0x9ff2bcc>
Ns3Node linkconfig <ns3_nodes.Ns3Node object at 0x9ff254c> <vnode.TunTap object at
0x9ff2aac> 54000000 None None None None
Ns3Node attach <ns3_nodes.Ns3Node object at 0x9ff254c> <vnode.TunTap object at
0x9ff2acc>
append ifname n2.0.72 node <ns3.Node object at 0x9ff2c0c>
Ns3Node linkconfig <ns3_nodes.Ns3Node object at 0x9ff254c> <vnode.TunTap object at
0x9ff2acc> 54000000 None None None None
Ns3Node attach <ns3_nodes.Ns3Node object at 0x9ff254c> <vnode.TunTap object at
0x9ff2bec>
append ifname n3.0.72 node <ns3.Node object at 0x9ff2c4c>
Ns3Node linkconfig <ns3_nodes.Ns3Node object at 0x9ff254c> <vnode.TunTap object at
0x9ff2bec> 54000000 None None None None
EVENT 3: CORE_EVENT_INSTANTIATION_STATE at Tue May 18 14:25:24 2010
ready to instantiate ns-3
ns3_mgr reset <ns3_mgr.Ns3Manager object at 0x9fefb8c>
checking obj <pycore_nodes.QuaggaNode object at 0x9ff23ec>
checking obj <pycore_nodes.QuaggaNode object at 0x9ff228c>
checking obj <pycore_nodes.QuaggaNode object at 0x9ff244c>
checking obj <ns3_nodes.Ns3Node object at 0x9ff254c>
obj is Ns3Node, addobj <ns3_nodes.Ns3Node object at 0x9ff254c>
```

and then, once positions are updated in the GUI, one can see many lines such as the following:

```
Ns3Node setposition 30.0 30.8 0.0 on ifname n1.0.72
Ns3Node setposition on node <ns3.Node object at 0x9ff2bcc>
Ns3Node setposition <ns3_nodes.Ns3Node object at 0x9ff254c> n1.0.72 149.0 153.0 None
Ns3Node setposition 29.8 30.6 0.0 on ifname n1.0.72
Ns3Node setposition on node <ns3.Node object at 0x9ff2bcc>
Ns3Node setposition <ns3_nodes.Ns3Node object at 0x9ff254c> n1.0.72 147.0 153.0 None
Ns3Node setposition 29.4 30.6 0.0 on ifname n1.0.72
Ns3Node setposition on node <ns3.Node object at 0x9ff2bcc>
Ns3Node setposition <ns3_nodes.Ns3Node object at 0x9ff254c> n1.0.72 138.0 148.0 None
```

The above lines correspond to CORE API messages as the nodes are dragged across the canvas.

5. Caveats and Known Issues

Limited scope of utility. The scenario is limited to an N-node adhoc wifi scenario (single network), from the GUI. Additional configurations require the definition of more Ns3Model subclasses and their mapping to GUI configuration elements.

No attribute configuration. The configuration values modified in the Configuration dialog are currently ignored. The only way to make changes to the ns-3 Attributes is by changing the ns-3 script segments in the Ns3Model-derived code (cf. Ns3WifiAdhocModel.py).

No reflection of ns-3 mobility to the execution GUI. Movement driven by ns-3 mobility models (if added to the scenario) is not rendered on the canvas. Only canvas-based GUI movements are reflected to the underlying ns-3 Node position.

Performance. ns-3 wireless simulations are compute intensive. Since the ns-3 simulations are run in real-time, it is unlikely that large wireless networks can provide real-time simulation of full-speed networks. In this case, thundering-herd-like effects can be observed as the simulator uses any idle time to "catch up."

Miscellaneous issues. When configuring the available plugins (see Figure 4-1) for ns3 models, the Configure button must be pressed to convey the model selection to the Ns3Node in the CORE daemon. It is not necessary to change any of the values in the configure dialog.

Some crashes have been observed when tearing down ns-3 simulations as the CORE GUI is exited or experiments are stopped.

The modified CORE (cored.py) must be run as root, from within a root waf ns-3 shell.

6. Performance Profiles

It was expected that the performance of the CORE ns-3 system would be limited by the interference calculations of the ns-3 wifi model. This is in contrast with an ns-3 CSMA-based system where the performance will be limited by the number of packets and therefore the number of scheduled ns-3 events.

To verify this hypothesis, the CORE GUI was used to assemble wireless ad-hoc networks composed of various numbers of nodes, all within reception range of one another. The well-known program "iperf" was used to calculate throughput between two nodes of the simulation using bash shells associated with the nodes' Linux containers. Figure 6-1 summarizes the results for measurements taken on a 2.8 GHz Intel core i7-processor equipped machine with 8 GB of RAM running Fedora 12.

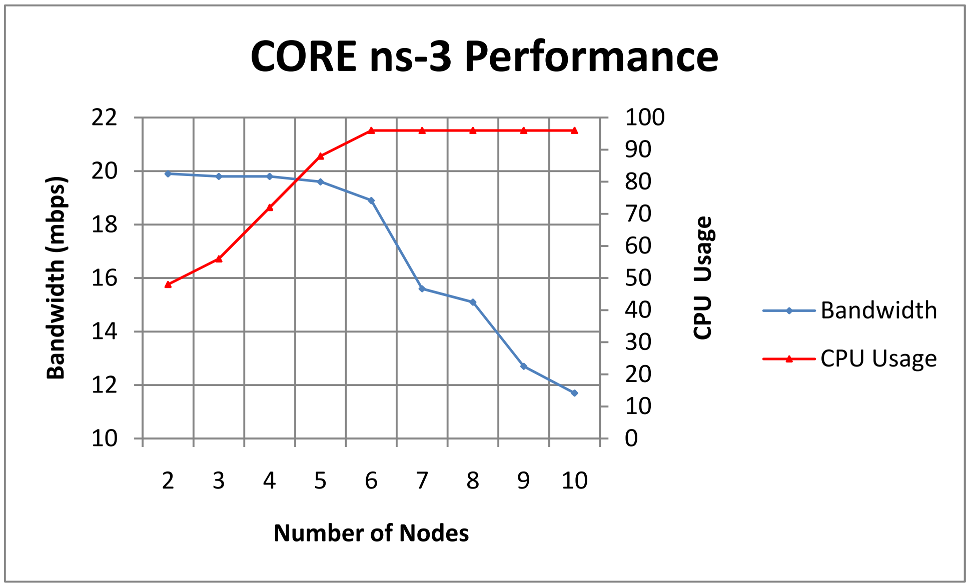


Figure 6-1. performance of various ns-3 and CORE configurations.

As a general rule of thumb, one should begin to get worried about CPU utilizations of around 70-80 percent. This rule of thumb demonstrates its applicability in the CORE ns-3 case. The red curve in Figure 6-1 is the CPU utilization of the processor running the ns-3 simulator. As the processor utilization crosses the 70 to 80 percent range, the bandwidth accomplished by the simulation begins to drop off. This suggests that the interference calculations for a five node ns-3 wifi simulation cannot be done in real time, and the situation degrades as more and more nodes are added.

As the loads increase, the ns-3 real-time simulator attempts to keep up with real time and does not fail, but the amount of data moved across the network is significantly reduced.

The support of only four wireless nodes on a capable modern Linux machine before the program becomes CPU-bound is a concern. Performance profiling reveals that Wifi DCF and interference computations are at the top:

```
CPU: Intel Architectural Perfmon, speed 2801 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of
0x00 (No unit mask) count 100000
samples  %        symbol name
41        3.5745  ns3::DcfManager::GetAccessGrantStart() const
38        3.3130  __i686.get_pc_thunk.bx
30        2.6155  ns3::DcfManager::MostRecent(ns3::TimeUnit<1>, ns3::TimeUnit<1>,
ns3::TimeUnit<1>, ns3::TimeUnit<1>, ns3::TimeUnit<1>, ns3::TimeUnit<1>,
ns3::TimeUnit<1>) const
19        1.6565  ns3::YansErrorRateModel::Binomial(unsigned int, double, unsigned
int) const
16        1.3949  __i686.get_pc_thunk.cx
14        1.2206
ns3::InterferenceHelper::CalculatePer(ns3::Ptr<ns3::InterferenceHelper::Event const>,
std::vector<ns3::InterferenceHelper::NiChange,
std::allocator<ns3::InterferenceHelper::NiChange> >*) const
12        1.0462  ns3::Simulator::Now()
12        1.0462  ns3::YansErrorRateModel::CalculatePdEven(double, unsigned int) const
```

7. Recommendations for Future Work

This prototyping effort achieved the desired goal of producing a working demonstration and also highlighting the areas that require further work or rework. As CORE shifts to a python implementation and support of Linux namespaces, the internals of CORE will change. This work will be maintained as a core-ns3 branch on the NRL subversion repository in the meantime. Below are some issues that require architectural discussion or significant implementation.

1) Align CORE with ns-3 node and channel convention. In ns-3, a channel is a container that has references to a number of NetDevices, but each NetDevice is the property of an ns-3 node. We understand that a similar abstraction holds for EMANE. However, in CORE, the wireless channel "object" or node conceptually holds the state for all of the NetDevices (while the state is represented in the GUI). This will make integration more difficult, and leads to difficulties in configuring one or more non-default configurations for the channel (e.g. configuring an access point among lots of station devices). We recommend that CORE associate NetDevices (with layer-2 configuration capabilities) with the router Nodes.

2) Agree on conventions for storing scenario definition files. CORE, EMANE, and ns-3 all have their own specific formats.

3) Focus on supporting execution mode only with the CORE GUI and building emulations out of scripts. This would make the netns3 mode of configuring experiments the preferred mode until a GUI scenario builder is produced. Plumbing of attributes and mapping CORE GUI API messages back and forth to ns-3 commands is going to be expensive to design, test, and maintain for the wide range of the ns-3 API.

4) Add Pause() and Resume() capability to ns-3 mobility models, and link to CORE GUI. When an ns-3 mobility model is in use, and the user wants to drag a node to a position on the canvas, there is no convention for which action is in charge of the subsequent mobility (whether the ns-3 mobility takes over or whether the position is static). We suggest a convention of having GUI movement override the ns-3 mobility; if a user drags a node, the ns-3 mobility is paused, and SetPosition() is called. User must somehow call "Resume()" to continue with ns-3 mobility.

5) More performance profiling. Improvements to model scalability are needed to use ns-3 wifi in this configuration. Next step is more comprehensive profiling and experimentation with other device models. Experimentation with setting different policies in the ns-3 real time scheduler, and logging/warning of events that fall outside of a tolerance window, is needed.

Acknowledgments

This work greatly benefited from software contributed by, and technical discussions with, Jeff Ahrenholz and Tom Goff of Boeing Research & Technology, and from helpful discussions with Mathieu Lacage of INRIA. This material is based upon work supported by the National Science Foundation under Grant No. CNS-0551686. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[CORE] J. Ahrenholz, C. Danilov, T. Henderson, and J.H. Kim, CORE: A real-time network emulator, *Proceedings of IEEE MILCOM Conference, 2008*.

{EMANE} Unpublished work; available online at http://cs.itd.nrl.navy.mil/work/emane/index.php.

[NEPI] M. Lacage, M. Ferrari, M. Hansen, and T. Turletti, "NEPI: Using Independent Simulators, Emulators, and Testbeds for Easy Experimentation," *Proceedings of ROADS Workshop*, October 2009.

[netns3] Unpublished work; available online at http://www.nsnam.org/wiki/index.php/HOWTO_use_Linux_namespaces_with_ns-3.

[ns3] Unpublished work; available online at http://www.nsnam.org.