Common Open Research Emulator

Network emulation tool version 4.3

Table of Contents

Introduction

The Common Open Research Emulator (CORE) is a tool for building virtual networks. As an emulator, CORE builds a representation of a real computer network that runs in real time, as opposed to simulation, where abstract models are used. The live-running emulation can be connected to physical networks and routers. It provides an environment for running real applications and protocols, taking advantage of virtualization provided by the Linux or FreeBSD operating systems.

Some of its key features are:

- efficient and scalable
- runs applications and protocols without modification
- easy-to-use GUI
- highly customizable

CORE is typically used for network and protocol research, demonstrations, application and platform testing, evaluating networking scenarios, security studies, and increasing the size of physical test networks.

Architecture

The main components of CORE are shown in Figure 1. A *CORE Services* layer manages emulation sessions. It builds emulated networks using kernel virtualization for virtual nodes and some form of bridging and packet manipulation for virtual networks. The nodes and networks come together via interfaces installed on nodes. The services layer is controlled via the graphical user interface, the *CORE GUI*, or directly using Python scripts. One component of the services is a CORE Python module that can be imported by these scripts. The GUI and the services layer communicate using a custom, asynchronous, sockets-based API, known as the *CORE API*.

The system is modular to allow mixing different components. The virtual networks component, for example, can be realized with other network simulators and emulators, such as ns-3 and EMANE. Another example is how a session can be designed and started using the GUI, and continue to run in "headless" operation with the GUI closed. The CORE API is sockets based, to allow the possibility of running different components on different physical machines.

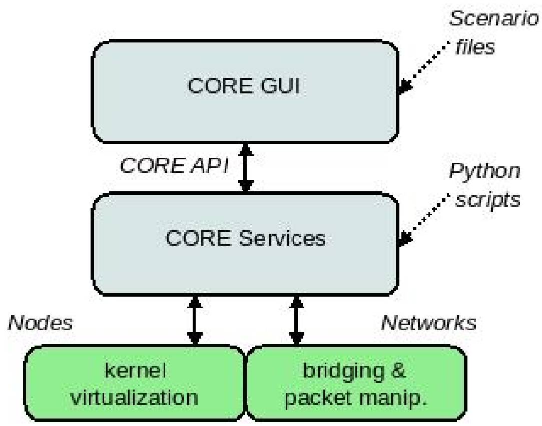

Figure 1: CORE Architecture

The CORE GUI is a Tcl/Tk program; it is started using the command `core`. The CORE Services, provided by the `cored.py` Python daemon, are controlled via the `/etc/init.d/core` init script. The CORE Services manage the virtual nodes and networks, of which other scripts and utilities may be available for further control.

How Does it Work?

A CORE node is a lightweight virtual machine. The CORE framework runs on Linux and FreeBSD systems. The primary platform used for development is Linux.

- [Linux], page 2 CORE uses Linux network namespace virtualization to build virtual nodes, and ties them together with virtual networks using Linux Ethernet bridging.
- [FreeBSD], page 3 CORE uses jails with a network stack virtualization kernel option to build virtual nodes, and ties them together with virtual networks using BSD's Netgraph system.

Linux

Linux network namespaces (also known as netns, LXC, or Linux containers[1]) is the primary virtualization technique used by CORE. LXC has been part of the mainline Linux kernel since 2.6.24. Recent Linux distributions such as Fedora and Ubuntu have namespaces-enabled kernels out of the box, so the kernel does not need to be patched or recompiled. A namespace is created using the `clone()` system call. Similar to the BSD jails, each namespace has its own process environment and private network stack. Network namespaces share the same filesystem in CORE.

CORE combines these namespaces with Linux Ethernet bridging to form networks. Link characteristics are applied using Linux Netem queuing disciplines. Ebtables is Ethernet frame filtering on Linux bridges. Wireless networks are emulated by controlling which interfaces can send and receive with ebtables rules.

[1] http://lxc.sourceforge.net/

FreeBSD

FreeBSD jails provide an isolated process space, a virtual environment for running programs. Starting with FreeBSD 8.0, a new `vimage` kernel option extends BSD jails so that each jail can have its own virtual network stack – its own networking variables such as addresses, interfaces, routes, counters, protocol state, socket information, etc. The existing networking algorithms and code paths are intact but operate on this virtualized state.

Each jail plus network stack forms a lightweight virtual machine. These are named jails or *virtual images* (or *vimages*) and are created using a the `jail` command or `vimage` command. Unlike traditional virtual machines, vimages do not feature entire operating systems running on emulated hardware. All of the vimages will share the same processor, memory, clock, and other system resources. Because the actual hardware is not emulated and network packets can be passed by reference through the in-kernel Netgraph system, vimages are quite lightweight and a single system can accommodate numerous instances.

Virtual network stacks in FreeBSD were historically available as a patch to the FreeBSD 4.11 and 7.0 kernels, and the VirtNet project[2][3] added this functionality to the mainline 8.0-RELEASE and newer kernels.

The FreeBSD Operating System kernel features a graph-based networking subsystem named Netgraph. The netgraph(4) manual page quoted below best defines this system:

> The netgraph system provides a uniform and modular system for the implementation of kernel objects which perform various networking functions. The objects, known as nodes, can be arranged into arbitrarily complicated graphs. Nodes have hooks which are used to connect two nodes together, forming the edges in the graph. Nodes communicate along the edges to process data, implement protocols, etc.
>
> The aim of netgraph is to supplement rather than replace the existing kernel networking infrastructure.

Prior Work

The Tcl/Tk CORE GUI was originally derived from the open source IMUNES project from the University of Zagreb[4], as a custom project within Boeing Research and Technology's Network Technology research group in 2004. Since then they have developed the CORE framework to use not only FreeBSD but Linux virtualization, have developed a Python framework, and made numerous user- and kernel-space developments, such as support for wireless networks, IPsec, the ability to distribute emulations, simulation integration, and more. The IMUNES project also consists of userspace and kernel components. Originally, one had to download and apply a patch for the FreeBSD 4.11 kernel, but the more recent VirtNet effort[5] has brought network stack virtualization to the more modern FreeBSD 8.x kernel.

---

[2] http://www.nlnet.nl/project/virtnet/

[3] http://www.imunes.net/virtnet/

[4] http://www.tel.fer.hr/imunes/

[5] http://www.nlnet.nl/project/virtnet/

Open Source Project and Resources

CORE has been released by Boeing to the open source community under the BSD license. If you find CORE useful for your work, please contribute back to the project. Contributions can be as simple as reporting a bug, dropping a line of encouragement or technical suggestions to the mailing lists, or can also include submitting patches or maintaining aspects of the tool. For details on contributing to CORE, please visit the wiki.

Besides this manual, there are other additional resources available online:

- CORE website - main project page containing demos, downloads, and mailing list information.
- CORE supplemental website - supplemental Google Code page with a quickstart guide, wiki, bug tracker, and screenshots.

The CORE wiki is a good place to check for the latest documentation and tips.

Goals

These are the Goals of the CORE project; they are similar to what we consider to be the [Introduction], page 1.

1. Ease of use - In a few clicks the user should have a running network.
2. Efficiency and scalability - A node is more lightweight than a full virtual machine. Tens of nodes should be possible on a standard laptop computer.
3. Real software - Run real implementation code, protocols, networking stacks.
4. Networking - CORE is focused on emulating networks and offers various ways to connect the running emulation with real or simulated networks.
5. Hackable - The source code is available and easy to understand and modify.

Non-Goals

This is a list of Non-Goals, specific things that people may be interested in but are not areas that we will pursue.

1. Reinventing the wheel - Where possible, CORE reuses existing open source components such as virtualization, Netgraph, netem, bridging, Quagga, etc.
2. 1,000,000 nodes - While the goal of CORE is to provide efficient, scalable network emulation, there is no set goal of N number of nodes. There are realistic limits on what a machine can handle as its resources are divided amongst virtual nodes. We will continue to make things more efficient and let the user determine the right number of nodes based on available hardware and the activities each node is performing.
3. Solves every problem - CORE is about emulating networking layers 3-7 using virtual network stacks in the Linux or FreeBSD operating systems.
4. Hardware-specific - CORE itself is not an instantiation of hardware, a testbed, or a specific laboratory setup; it should run on commodity laptop and desktop PCs, in addition to high-end server hardware.

1 Installation

This chapter describes how to set up a CORE machine. Note that the easiest way to install CORE is using a binary package on Ubuntu or Fedora (deb or rpm) using the distribution's package manager to automatically install dependencies, see Section 1.2 [Installing from Packages], page 6.

Ubuntu and Fedora Linux are the recommended distributions for running CORE. Ubuntu 10.04, 10.10, 11.04, or 11.10 and Fedora 14, 15, or 16 ship with kernels with support for namespaces built-in. They support the latest hardware. However, these distributions are not strictly required. CORE will likely work on other flavors of Linux, see Section 1.3 [Installing from Source], page 8.

The primary dependencies are Tcl/Tk (8.5 or newer) for the GUI, and Python 2.6 or 2.7 for the CORE services.

1.1 Prerequisites

The Linux or FreeBSD operating system is required. The GUI uses the Tcl/Tk scripting toolkit, and the CORE services require Python. Details of the individual software packages required can be found in the installation steps.

1.1.1 Required Hardware

Any computer capable of running Linux or FreeBSD should be able to run CORE. Since the physical machine will be hosting numerous virtual machines, as a general rule you should select a machine having as much RAM and CPU resources as possible.

A *general recommendation* would be:

- 2.0GHz or better x86 processor, the more processor cores the better
- 2 GB or more of RAM
- about 3 MB of free disk space (plus more for dependency packages such as Tcl/Tk)
- X11 for the GUI, or remote X11 over SSH

The computer can be a laptop, desktop, or rack-mount server. A keyboard, mouse, and monitor are not required if a network connection is available for remotely accessing the machine. A 3D accelerated graphics card is not required.

1.1.2 Required Software

CORE requires the Linux or FreeBSD operating systems because it uses virtualization provided by the kernel. It does not run on the Windows or Mac OS X operating systems (unless it is running within a virtual machine guest.) There are two different virtualization technologies that CORE can currently use: Linux network namespaces and FreeBSD jails, see see [How Does it Work?], page 2 for virtualization details.

Linux network namespaces is the recommended platform. Development is focused here and it supports the latest features. It is the easiest to install because there is no need to patch, install, and run a special Linux kernel.

FreeBSD 9.0-RELEASE may offer the best scalability. If your applications run under FreeBSD and you are comfortable with that platform, this may be a good choice. Device and application support by BSD may not be as extensive as Linux.

The CORE GUI requires the X.Org X Window system (X11), or can run over a remote X11 session. For specific Tcl/Tk, Python, and other libraries required to run CORE, refer to the Chapter 1 [Installation], page 5 section.

1.2 Installing from Packages

The easiest way to install CORE is using the pre-built packages. The package managers on Ubuntu or Fedora (`gdebi-gtk` or `yum`) will automatically install dependencies for you. You can obtain the CORE packages from the CORE downloads page.

1.2.1 Installing from Packages on Ubuntu

First install the Ubuntu 10.04, 10.10, 11.04, or 11.10 operating system.

1. **Optional:** install the prerequisite packages (otherwise skip this step and have the package manager install them for you.)
```
# make sure the system is up to date; you can also use synaptic or
#  update-manager instead of apt-get update/dist-upgrade
sudo apt-get update
sudo apt-get dist-upgrade
sudo apt-get install bash bridge-utils ebtables iproute \
     libev3 libtk-img python tcl8.5 tk8.5 xterm
```
2. Install Quagga for routing. If you plan on working with wireless networks, we recommend installing OSPF MDR (replace `amd64` below with `i386` if needed to match your architecture):
```
wget http://downloads.pf.itd.nrl.navy.mil/ospf-manet/\
    quagga-0.99.20mr2.1/quagga-mr_0.99.20mr2.1_amd64.deb
sudo dpkg -i quagga-mr_0.99.20mr2.1_amd64.deb
```
   or, for the regular Ubuntu version of Quagga:
```
sudo apt-get install quagga
```
3. Install the CORE deb package for Ubuntu, using a GUI that automatically resolves dependencies (note that the absolute path to the deb file must be used with `software-center`):
```
software-center /home/user/Downloads/core_4.3-0ubuntu1_amd64.deb
```
   or install from command-line:
```
sudo dpkg -i core_4.3-0ubuntu1_amd64.deb
```
4. Start the CORE services as root.
```
sudo /etc/init.d/core start
```
5. Run the CORE GUI as a normal user:
```
core
```

After running the `core` command, a GUI should appear with a canvas for drawing topologies. Messages will print out on the console about connecting to the CORE daemons.

1.2.2 Installing from Packages on Fedora

The commands shown here should be run as root. First Install the Fedora 14, 15, or 16 operating system. The `x86_64` architecture is shown in the examples below, replace with `i386` is using a 32-bit architecture. Also, `fc15` is shown below for Fedora 15 packages, replace with the appropriate Fedora release number.

1. **Optional:** install the prerequisite packages (otherwise skip this step and have the package manager install them for you.)

   ```
   # make sure the system is up to date; you can also use the
   #  update applet instead of yum update
   yum update
   yum install bash bridge-utils ebtables libev python \
       tcl tk tkimg urw-fonts xauth xorg-x11-server-utils xterm
   ```

2. Install Quagga for routing. If you plan on working with wireless networks, we recommend installing OSPF MDR:

   ```
   wget http://downloads.pf.itd.nrl.navy.mil/ospf-manet/quagga-0.99.20mr2.1\
       quagga-0.99.20mr2.1.fc15.x86_64.rpm
   rpm -ivh quagga-0.99.20mr2.1.fc15.x86_64.rpm
   ```

   or, for the regular Fedora version of Quagga:

   ```
   yum install quagga
   ```

3. Install the CORE RPM package for Fedora and automatically resolve dependencies:

   ```
   yum localinstall core-4.3-1.fc15.x86_64.rpm --nogpgcheck
   ```

   or install from the command-line:

   ```
   rpm -ivh core-4.3-1.fc15.x86_64.rpm
   ```

4. Turn off SELINUX by setting `SELINUX=disabled` in the '`/etc/sysconfig/selinux`' file, and adding `setlinux=0` to the kernel line in your '`/etc/grub.conf`' file; on Fedora 15 and newer, disable sandboxd using `chkconfig sandbox off`; you need to reboot in order for this change to take effect
5. Turn off firewalls with `chkconfig iptables off`, `chkconfig ip6tables off` or configure them with permissive rules for CORE virtual networks; you need to reboot after making this change, or flush the firewall using `iptables -F`, `ip6tables -F`.
6. Start the CORE services as root.

   ```
   /etc/init.d/core start
   ```

7. Run the CORE GUI as a normal user:

   ```
   core
   ```

After running the `core` command, a GUI should appear with a canvas for drawing topologies. Messages will print out on the console about connecting to the CORE daemons.

1.2.3 Installing from Packages on FreeBSD

First install the FreeBSD 9.0 operating system.

1. Install the prerequisite packages:

   ```
   pkg_add -r tk85
   pkg_add -r libimg
   pkg_add -r bash
   pkg_add -r libev
   pkg_add -r sudo
   ```

```
pkg_add -r xterm
```

The `sudo` package needs to be configured so a normal user can run the CORE GUI using the command `core` (opening a shell window on a node uses a command such as 'sudo vimage n1'.)

2. Install Quagga for routing. If you plan on working with wireless networks, we recommend installing OSPF MDR. On FreeBSD this must be built from source. Another option is to use the regular FreeBSD port of Quagga:

   ```
   pkg_add -r quagga
   ```

3. Download the CORE packages on the FreeBSD machine and install them using `pkg_add`:

   ```
   pkg_add core-9.x-4.3.tbz
   pkg_add core-freebsd-kernel-9.x-amd64.tbz
   ```

4. Reboot following the install to enable the new kernel.
5. Start the CORE services daemons as root. You can add `core_enable="YES"` to your '`/etc/rc.conf`' file for automatic startup.

   ```
   sudo /usr/local/etc/rc.d/core onestart
   ```

6. Run the CORE GUI as a normal user:

   ```
   core
   ```

After running the `core` command, a GUI should appear with a canvas for drawing topologies. Messages will print out on the console about connecting to the CORE daemons.

On FreeBSD, the `sudo` command should be configured such that the GUI, running as a normal user, can execute the `vimage` command as root. This allows opening a shell by double-clicking on a node.

1.3 Installing from Source

This option is listed here for developers and advanced users who are comfortable patching and building source code. Please consider using the binary packages instead for a simplified install experience.

1.3.1 Installing from Source on Ubuntu

To build CORE from source on Ubuntu, first install these development packages. These packages are not required for normal binary package installs.

```
sudo apt-get install bash bridge-utils ebtables iproute \
     libev3 libtk-img python tcl8.5 tk8.5 xterm \
     autoconf automake gcc libev-dev make \
     pkg-config python-dev libreadline-dev \
     imagemagick texinfo help2man
```

You can obtain the CORE source from the CORE source page. Choose either a stable release version or the development snapshot available in the `nightly_snapshots` directory.

```
tar xzf core-4.3.tar.gz
cd core-4.3
./bootstrap.sh
./configure
```

```
make
sudo make install
```

1.3.2 Installing from Source on Fedora

To build CORE from source on Fedora, install these development packages. These packages are not required for normal binary package installs.

```
yum install bash bridge-utils ebtables libev python \
    tcl tk tkimg urw-fonts xauth xorg-x11-server-utils xterm \
    autoconf automake gcc libev-devel make \
    pkgconfig python-devel readline-devel \
    texinfo ImageMagick help2man
```

You can obtain the CORE source from the CORE source page. Choose either a stable release version or the development snapshot available in the `nightly_snapshots` directory.

```
tar xzf core-4.3.tar.gz
cd core-4.3
./bootstrap.sh
./configure
make
sudo make install
```

Note that the Linux RPM and Debian packages do not use the `/usr/local` prefix, and files are instead installed to `/usr/sbin`, and `/usr/lib`. This difference is a result of aligning with the directory structure of Linux packaging systems and FreeBSD ports packaging.

Another note is that the Python distutils in Fedora Linux will install the CORE Python modules to '`/usr/lib/python2.7/site-packages/core`', instead of using the '`dist-packages`' directory.

1.3.3 Installing from Source on FreeBSD

Rebuilding the Kernel

The source code for the FreeBSD kernel is located in '`/usr/src/sys`'. This kernel has support for network stack virtualization built-in, but the option needs to be turned on and the kernel recompiled. Additionally, a small patch is required to allow per-node directories in the filesystem.

Instructions below will use the '`/usr/src/sys/amd64`' architecture directory, but you should substitute the directory '`/usr/src/sys/i386`' if you are using a 32-bit architecture.

The kernel patch is available from the CORE source tarball under '`core-4.3/kernel/symlinks-8.1-RELEASE.diff`'. This patch applies to the FreeBSD 8.x or 9.x kernels.

```
cd /usr/src/sys
# first you can check if the patch applies cleanly using the '-C' option
patch -p1 -C < ~/core-4.3/kernel/symlinks-8.1-RELEASE.diff
# without '-C' applies the patch
patch -p1 < ~/core-4.3/kernel/symlinks-8.1-RELEASE.diff
```

A kernel configuration file named '`CORE`' can be found within the source tarball:

'`core-4.3/kernel/freebsd8-config-CORE`'

The contents of this configuration file are shown below; you can edit it to suit your needs.

```
# this is the FreeBSD 8.x kernel configuration file for CORE
include  GENERIC
ident CORE

options  VIMAGE
nooptions SCTP
options IPSEC
device crypto

options IPFIREWALL
options  IPFIREWALL_DEFAULT_TO_ACCEPT
```

The kernel configuration file can be linked or copied to the kernel source directory. Use it to configure and build the kernel:

```
cd /usr/src/sys/amd64/conf
cp ~/core-4.3/kernel/freebsd8-config-CORE CORE
config CORE
cd ../compile/CORE
make cleandepend && make depend
make -j8 && make install
```

Change the number 8 above to match the number of CPU cores you have times two. Note that the `make install` step will move your existing kernel to `/boot/kernel.old` and removes that directory if it already exists. Reboot to enable this new patched kernel.

Building CORE from source

Here are the prerequisite packages from the FreeBSD ports system:

```
pkg_add -r tk85
pkg_add -r libimg
pkg_add -r bash
pkg_add -r libev
pkg_add -r sudo
pkg_add -r xterm
pkg_add -r autotools
pkg_add -r gmake
```

The CORE source is built using autotools and gmake:

```
tar xzf core-4.3.tar.gz
cd core-4.3
./bootstrap.sh
./configure
gmake
sudo gmake install
```

Build and install the `vimage` utility for controlling virtual images. The source can be obtained from http://svn.freebsd.org/viewvc/base/head/tools/tools/vimage/, or it is included with the CORE source for convenience:

```
cd core-4.3/kernel/vimage
```

```
make
make install
```

On FreeBSD you should also install the CORE kernel modules for wireless emulation. Perform this step after you have recompiled and installed FreeBSD kernel.

```
cd core-4.3/kernel/ng_pipe
make
sudo make install
cd ../ng_wlan
make
sudo make install
```

The 'ng_wlan' kernel module allows for the creation of WLAN nodes. This is a modified 'ng_hub' Netgraph module. Instead of packets being copied to every connected node, the WLAN maintains a hash table of connected node pairs. Furthermore, link parameters can be specified for node pairs, in addition to the on/off connectivity. The parameters are tagged to each packet and sent to the connected 'ng_pipe' module. The 'ng_pipe' has been modified to read any tagged parameters and apply them instead of its default link effects.

The 'ng_wlan' also supports linking together multiple WLANs across different machines using the 'ng_ksocket' Netgraph node, for distributed emulation.

On FreeBSD, the `sudo` command should be configured such that the GUI, running as a normal user, can execute the `vimage` command as root. This allows opening a shell by double-clicking on a node.

The Quagga routing suite is recommended for routing, see Section 1.4 [Quagga Routing Software], page 11 for installation.

1.4 Quagga Routing Software

Virtual networks generally require some form of routing in order to work (e.g. to automatically populate routing tables for routing packets from one subnet to another.) CORE builds OSPF routing protocol configurations by default when the blue router node type is used. The OSPF protocol is available from the Quagga open source routing suite. Other routing protocols are available using different node services, see Section 2.6.1 [Default Services and Node Types], page 29.

Quagga is not specified as a dependency for the CORE package because there are two different Quagga packages that you may use:

- Quagga - the standard version of Quagga, suitable for static wired networks, and usually available via your distribution's package manager.
- OSPF MANET Designated Routers (MDR) - the Quagga routing suite with a modified version of OSPFv3, optimized for use with mobile wireless networks. The *mdr* node type (and the MDR service) requires this variant of Quagga.

If you plan on working with wireless networks, we recommend installing OSPF MDR; otherwise install the standard version of Quagga using your package manager or from source.

1.4.1 Installing Quagga from Packages

To install the standard version of Quagga from packages, use your package manager (Linux) or the ports system (FreeBSD).

Ubuntu users:

```
sudo apt-get install quagga
```

Fedora users:

```
yum install quagga
```

FreeBSD users:

```
pkg_add -r quagga
```

To install the Quagga variant having OSPFv3 MDR, first download the appropriate package, and install using the package manager.

Ubuntu users:

```
wget http://downloads.pf.itd.nrl.navy.mil/ospf-manet/quagga-0.99.20mr2.1\
    /quagga-mr_0.99.20mr2.1_amd64.deb
sudo dpkg -i quagga-mr_0.99.20mr2.1_amd64.deb
```

Replace `amd64` with `i386` if using a 32-bit architecture.

Fedora users:

```
wget http://downloads.pf.itd.nrl.navy.mil/ospf-manet/quagga-0.99.20mr2.1\
    /quagga-0.99.20mr2.1.fc15.x86_64.rpm
rpm -ivh quagga-0.99.20mr2.1.fc15.x86_64.rpm
```

Replace `x86_64` with `i686` if using a 32-bit architecture.

1.4.2 Compiling Quagga for CORE

To compile Quagga to work with CORE on Linux:

```
tar xzf quagga-0.99.20mr2.1.tar.gz
cd quagga-0.99.20mr2.1
./configure --enable-user=root --enable-group=root --with-cflags=-ggdb \
--sysconfdir=/usr/local/etc/quagga --enable-vtysh \
--localstatedir=/var/run/quagga
make
sudo make install
```

Note that the configuration directory '`/usr/local/etc/quagga`' shown for Quagga above could be '`/etc/quagga`', if you create a symbolic link from '`/etc/quagga/Quagga.conf -> /usr/local/etc/quagga/Quagga.conf`' on the host. The '`quaggaboot.sh`' script in a Linux network namespace will try and do this for you if needed.

To compile Quagga to work with CORE on FreeBSD:

```
tar xzf quagga-0.99.20mr2.1.tar.gz
cd quagga-0.99.20mr2.1
./configure --enable-user=root --enable-group=wheel \
    --sysconfdir=/usr/local/etc/quagga --enable-vtysh \
    --localstatedir=/var/run/quagga
```

```
gmake
gmake install
```

On FreeBSD 9.0 you can use `make` or `gmake`. You probably want to compile Quagga from the ports system in `/usr/ports/net/quagga`.

1.5 VCORE

CORE is capable of running inside of a virtual machine, using software such as VirtualBox, VMware Server or QEMU. However, CORE itself is performing machine virtualization in order to realize multiple emulated nodes, and running CORE virtually adds additional contention for the physical resources. **For performance reasons, this is not recommended.** Timing inside of a VM often has problems. If you do run CORE from within a VM, it is recommended that you view the GUI with remote X11 over SSH, so the virtual machine does not need to emulate the video card with the X11 application.

A CORE virtual machine is provided for download, named VCORE. This is the perhaps the easiest way to get CORE up and running as the machine is already set up for you. This may be adequate for initially evaluating the tool but keep in mind the performance limitations of running within VirtualBox or VMware. To install the virtual machine, you first need to obtain VirtualBox from http://www.virtualbox.org, or VMware Server or Player from http://www.vmware.com (this commercial software is distributed for free.) Once virtualization software has been installed, you can import the virtual machine appliance using the `vcore.ovf` file. See the documentation that comes with VCORE for login information.

2 Using the CORE GUI

CORE can be used via the GUI or Chapter 3 [Python Scripting], page 33. A typical emulation workflow is outlined in Figure 2.1. Often the GUI is used to draw nodes and network devices on the canvas. A Python script could also be written, that imports the CORE Python module, to configure and instantiate nodes and networks. This chapter primarily covers usage of the CORE GUI.

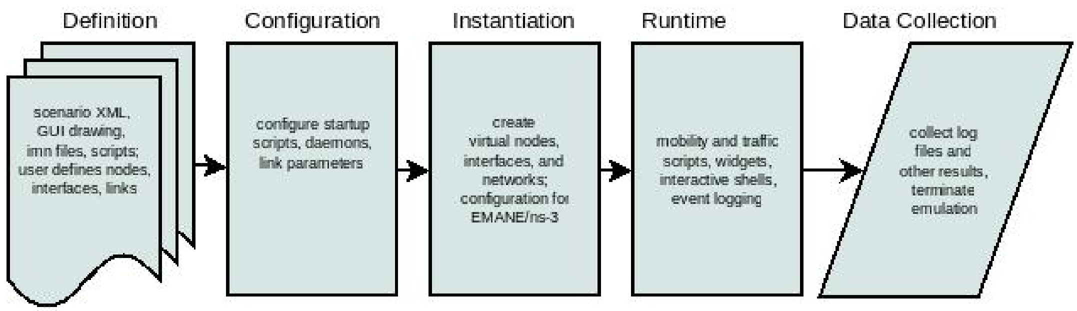

Figure 2.1: Emulation Workflow

CORE can be customized to perform any action at each phase depicted in Figure 2.1. See the *Hooks...* entry on the Section 2.3.7 [Experiment Menu], page 21 for details about when these session states are reached.

2.1 Modes of Operation

The CORE GUI has two primary modes of operation, **Edit** and **Execute** modes. Running the GUI, by typing `core` with no options, starts in Edit mode. Nodes are drawn on a blank canvas using the toolbar on the left and configured from right-click menus or by double-clicking them. The GUI does not need to be run as root.

Once editing is complete, pressing the green `Start` button (or choosing `Execute` from the `Experiment` menu) instantiates the topology within the FreeBSD kernel and enters Execute mode. In execute mode, the user can interact with the running emulated machines by double-clicking or right-clicking on them. The editing toolbar disappears and is replaced by an execute toolbar, which provides tools while running the emulation. Pressing the red `Stop` button (or choosing `Terminate` from the `Experiment` menu) will destroy the running emulation and return CORE to Edit mode.

CORE can be started directly in Execute mode by specifying `--start` and a topology file on the command line:

```
core --start ~/.core/configs/myfile.imn
```

Once the emulation is running, the GUI can be closed, and a prompt will appear asking if the emulation should be terminated. The emulation may be left running and the GUI can reconnect to an existing session at a later time.

There is also a **Batch** mode where CORE runs without the GUI and will instantiate a topology from a given file. This is similar to the `--start` option, except that the GUI is not used:

```
core --batch ~/.core/configs/myfile.imn
```

A session running in batch mode can be accessed using the `vcmd` command (or `vimage` on FreeBSD), or the GUI can connect to the session.

The session number is printed in the terminal when batch mode is started. This session number can later be used to stop the batch mode session:

```
core --closebatch 12345
```

The GUI can be run as a normal user on Linux. For FreeBSD, the GUI should be run as root in order to start an emulation.

2.2 Toolbar

The toolbar is a row of buttons that runs vertically along the left side of the CORE GUI window. The toolbar changes depending on the mode of operation.

2.2.1 Editing Toolbar

When CORE is in Edit mode (the default), the vertical Editing Toolbar exists on the left side of the CORE window. Below are brief descriptions for each toolbar item, starting from the top. Most of the tools are grouped into related sub-menus, which appear when you click on their group icon.

- *Selection Tool* - default tool for selecting, moving, configuring nodes
- *Start button* - starts Execute mode, instantiates the emulation
- *Link* - the Link Tool allows network links to be drawn between two nodes by clicking and dragging the mouse
- *Network-layer virtual nodes*
  - *Router* - runs Quagga OSPFv2 and OSPFv3 routing to forward packets
  - *Host* - emulated server machine having a default route, runs SSH server
  - *PC* - basic emulated machine having a default route, runs no processes by default
  - *MDR* - runs Quagga OSPFv3 MDR routing for MANET-optimized routing
  - *PRouter* - physical router represents a real testbed machine, see Section 4.2 [physical], page 35.
  - *Edit* - edit node types button invokes the CORE Node Types dialog. New types of nodes may be created having different icons and names. The default services that are started with each node type can be changed here.
- *Link-layer nodes*
  - *Hub* - the Ethernet hub forwards incoming packets to every connected node
  - *Switch* - the Ethernet switch intelligently forwards incoming packets to attached hosts using an Ethernet address hash table
  - *Wireless LAN* - when routers are connected to this WLAN node, they join a wireless network and an antenna is drawn instead of a connecting line; the WLAN node typically controls connectivity between attached wireless nodes based on the distance between them

  - *RJ45* - with the RJ45 Physical Interface Tool, emulated nodes can be linked to real physical interfaces on the Linux or FreeBSD machine; using this tool, real networks and devices can be physically connected to the live-running emulation (see Section 2.4.1 [RJ45 Tool], page 23)
  - *Tunnel* - the Tunnel Tool allows connecting together more than one CORE emulation using GRE tunnels
- *Annotation Tools*
  - *Marker* - for drawing marks on the canvas
  - *Oval* - for drawing circles on the canvas that appear in the background
  - *Rectangle* - for drawing rectangles on the canvas that appear in the background
  - *Text* - for placing text captions on the canvas

2.2.2 Execution Toolbar

When the Start button is pressed, CORE switches to Execute mode, and the Edit toolbar on the left of the CORE window is replaced with the Execution toolbar. Below are the items on this toolbar, starting from the top.

- *Selection Tool* - in Execute mode, the Selection Tool can be used for moving nodes around the canvas, and double-clicking on a node will open a shell window for that node; right-clicking on a node invokes a pop-up menu of run-time options for that node
- *Stop button* - stops Execute mode, terminates the emulation, returns CORE to edit mode.
- *Observer Widgets Tool* - clicking on this magnifying glass icon invokes a menu for easily selecting an Observer Widget. The icon has a darker gray background when an Observer Widget is active, during which time moving the mouse over a node will pop up an information display for that node (see Section 2.3.6.2 [Observer Widgets], page 21).
- *Marker* - for drawing freehand lines on the canvas, useful during demonstrations
- *Two-node Tool* - click to choose a starting and ending node, and run a one-time `traceroute` between those nodes or a continuous `ping -R` between nodes. The output is displayed in real time in a results box, while the IP addresses are parsed and the complete network path is highlighted on the CORE display.
- *Run Tool* - this tool allows easily running a command on all or a subset of all nodes. A list box allows selecting any of the nodes. A text entry box allows entering any command. The command should return immediately, otherwise the display will block awaiting response. The `ping` command, for example, with no parameters, is not a good idea. The result of each command is displayed in a results box. The first occurrence of the special text `"NODE"` will be replaced with the node name. The command will not be attempted to run on nodes that are not routers, PCs, or hosts, even if they are selected.

2.3 Menubar

The menubar runs along the top of the CORE GUI window and provides access to a variety of features. Some of the menus are detachable, such as the *Widgets* menu, by clicking the dashed line at the top.

2.3.1 File Menu

The File menu contains options for manipulating the `.imn` Section 2.8 [Configuration Files], page 31. Generally, these menu items should not be used in Execute mode (see Section 2.1 [Modes of Operation], page 15.)

- *New* - this starts a new file with an empty canvas.
- *Open* - invokes the File Open dialog box for selecting a new `.imn` topology file to open. You can change the default path used for this dialog in the Section 2.10 [Preferences], page 32 Dialog.
- *Save* - saves the current topology. If you have not yet specified a file name, the Save As dialog box is invoked.
- *Save As* - invokes the Save As dialog box for selecting a new `.imn` topology file for saving the current configuration. Files are saved in the *IMUNES network configuration* file format described in Section 2.8 [Configuration Files], page 31.
- *Open current file in vim* - this opens the current topology file in the `vim` text editor. First you need to save the file. Once the file has been edited with a text editor, you will need to reload the file to see your changes.
- *Print* - this uses the Tcl/Tk postscript command to print the current canvas to a printer. A dialog is invoked where you can specify a printing command, the default being `lpr`. The postscript output is piped to the print command.
- *Save screenshot* - saves the current canvas as a postscript graphic file.
- *Export Python script* - prints Python snippets to the console, for inclusion in a CORE Python script.
- *Recently used files* - above the Quit menu command is a list of recently use files, if any have been opened. You can clear this list in the Section 2.10 [Preferences], page 32 dialog box. You can specify the number of files to keep in this list from the Section 2.10 [Preferences], page 32 dialog. Click on one of the file names listed to open that configuration file.
- *Quit* - the Quit command should be used to exit the CORE GUI. CORE may prompt for termination if you are currently in Execute mode. Preferences and the recently-used files list are saved.

2.3.2 Edit Menu

- *Undo* - attempts to undo the last edit in edit mode.
- *Redo* - attempts to redo an edit that has been undone.
- *Select All* - selects all items on the canvas. Selected items can be moved as a group.
- *Select Adjacent* - select all nodes that are linked to the already selected node(s). For wireless nodes this simply selects the WLAN node(s) that the wireless node belongs to. You can use this by clicking on a node and pressing CTRL+N to select the adjacent nodes.

- *Clear marker* - clears any annotations drawn with the marker tool. Also clears any markings used to indicate a node's status.
- *Preferences...* - invokes the Section 2.10 [Preferences], page 32 dialog box.

2.3.3 Canvas Menu

The canvas menu provides commands for adding, removing, changing, and switching to different editing canvases, see Section 2.5.4 [Multiple Canvases], page 27.

- *New* - creates a new empty canvas at the right of all existing canvases.
- *Manage...* - invokes the *Manage Canvases* dialog box, where canvases may be renamed and reordered, and you can easily switch to one of the canvases by selecting it.
- *Delete* - deletes the current canvas and all items that it contains.
- *Size/scale...* - invokes a Canvas Size and Scale dialog that allows configuring the canvas size, scale, and geographic reference point. The size controls allow changing the width and height of the current canvas, in pixels or meters. The scale allows specifying how many meters are equivalent to 100 pixels. The reference point controls specify the latitude, longitude, and altitude reference point used to convert between geographic and Cartesian coordinate systems. By clicking the *Save as default* option, all new canvases will be created with these properties. The default canvas size can also be changed in the Section 2.10 [Preferences], page 32 dialog box.
- *Wallpaper...* - used for setting the canvas background image, see Section 2.9 [Customizing your Topology's Look], page 31.
- *Previous*, *Next*, *First*, *Last* - used for switching the active canvas to the first, last, or adjacent canvas.

2.3.4 View Menu

The View menu features items for controlling what is displayed on the drawing canvas.

- *Show* - opens a submenu of items that can be displayed or hidden, such as interface names, addresses, and labels. Use these options to help declutter the display. These options are generally saved in the `.imn` topology files, so scenarios have a more consistent look when copied from one computer to another.
- *Show hidden nodes* - reveal nodes that have been hidden. Nodes are hidden by selecting one or more nodes, right-clicking one and choosing *hide*.
- *Zoom In* - magnifies the display. You can also zoom in by clicking *zoom 100%* label in the status bar, or by pressing the `+` (plus) key.
- *Zoom Out* - reduces the size of the display. You can also zoom out by right-clicking *zoom 100%* label in the status bar or by pressing the `-` (minus) key.

2.3.5 Tools Menu

The tools menu lists different utility functions.

- *Autorearrange all* - automatically arranges all nodes on the canvas. Nodes having a greater number of links are moved to the center. This mode can continue to run while placing nodes. To turn off this autorearrange mode, click on a blank area of the canvas with the select tool, or choose this menu option again.
- *Autorearrange selected* - automatically arranges the selected nodes on the canvas.

- *Align to grid* - moves nodes into a grid formation, starting with the smallest-numbered node in the upper-left corner of the canvas, arranging nodes in vertical columns.
- *Traffic...* - invokes the CORE Traffic Flows dialog box, which allows configuring, starting, and stopping MGEN traffic flows for the emulation.
- *IP addresses...* - invokes the IP Addresses dialog box for configuring which IPv4/IPv6 prefixes are used when automatically addressing new interfaces.
- *MAC addresses...* - invokes the MAC Addresses dialog box for configuring the starting number used as the lowest byte when generating each interface MAC address. This value should be changed when tunneling between CORE emulations to prevent MAC address conflicts.
- *Build hosts file...* - invokes the Build hosts File dialog box for generating '`/etc/hosts`' file entries based on IP addresses used in the emulation.
- *Renumber nodes...* - invokes the Renumber Nodes dialog box, which allows swapping one node number with another in a few clicks.
- *Experimental...* - menu of experimental options, such as a tool to convert ns-2 scripts to IMUNES imn topologies, supporting only basic ns-2 functionality, and a tool for automatically dividing up a topology into partitions.
- *Topology generator* - opens a submenu of topologies to generate. You can first select the type of node that the topology should consist of, or routers will be chosen by default. Nodes may be randomly placed, aligned in grids, or various other topology patterns.
  - *Random* - nodes are randomly placed about the canvas, but are not linked together. This can be used in conjunction with a WLAN node (see Section 2.2.1 [Editing Toolbar], page 16) to quickly create a wireless network.
  - *Grid* - nodes are placed in horizontal rows starting in the upper-left corner, evenly spaced to the right; nodes are not linked to each other.
  - *Connected Grid* - nodes are placed in an N x M (width and height) rectangular grid, and each node is linked to the node above, below, left and right of itself.
  - *Chain* - nodes are linked together one after the other in a chain.
  - *Star* - one node is placed in the center with N nodes surrounding it in a circular pattern, with each node linked to the center node
  - *Cycle* - nodes are arranged in a circular pattern with every node connected to its neighbor to form a closed circular path.
  - *Wheel* - the wheel pattern links nodes in a combination of both Star and Cycle patterns.
  - *Cube* - generate a cube graph of nodes
  - *Clique* - creates a clique graph of nodes, where every node is connected to every other node
  - *Bipartite* - creates a bipartite graph of nodes, having two disjoint sets of vertices.
- *Debugger...* - opens the CORE Debugger window for executing arbitrary Tcl/Tk commands.

2.3.6 Widgets Menu

Widgets are GUI elements that allow interaction with a running emulation. Widgets typically automate the running of commands on emulated nodes to report status information of some type and display this on screen.

2.3.6.1 Periodic Widgets

These Widgets are those available from the main *Widgets* menu. More than one of these Widgets may be run concurrently. An event loop fires once every second that the emulation is running. If one of these Widgets is enabled, its periodic routine will be invoked at this time. Each Widget may have a configuration dialog box which is also accessible from the *Widgets* menu.

Here are some standard widgets:

- *Adjacency* - displays router adjacency states for Quagga's OSPFv2 and OSPFv3 routing protocols. A line is drawn from each router halfway to the router ID of an adjacent router. The color of the line is based on the OSPF adjacency state such as Two-way or Full. Only half of the line is drawn because each router may be in a different adjacency state with respect to the other.
- *Throughput* - displays the kilobits-per-second throughput above each link, using statistics gathered from the ng_pipe Netgraph node that implements each link. If the throughput exceeds a certain threshold, the link will become highlighted. For wireless nodes which broadcast data to all nodes in range, the throughput rate is displayed next to the node and the node will become circled if the threshold is exceeded. *Note: the Throughput Widget will display "0.0 kbps" on all links that have no configured link effects, because of the way link statistics are counted; to fix this, add a small delay or a bandwidth limit to each link.*

2.3.6.2 Observer Widgets

These Widgets are available from the *Observer Widgets* submenu of the *Widgets* menu, and from the Widgets Tool on the toolbar (see Section 2.2.2 [Execution Toolbar], page 17). Only one Observer Widget may be used at a time. Mouse over a node while the experiment is running to pop up an informational display about that node.

Available Observer Widgets include IPv4 and IPv6 routing tables, socket information, list of running processes, and OSPFv2/v3 neighbor information.

Observer Widgets may be edited by the user and rearranged. Choosing *Edit...* from the Observer Widget menu will invoke the Observer Widgets dialog. A list of Observer Widgets is displayed along with up and down arrows for rearranging the list. Controls are available for renaming each widget, for changing the command that is run during mouse over, and for adding and deleting items from the list. Note that specified commands should return immediately to avoid delays in the GUI display. Changes are saved to a '`widgets.conf`' file in the CORE configuration directory.

2.3.7 Experiment Menu

The Experiment Menu has entries for starting, stopping, and managing sessions, in addition to global options such as comments, hooks, node types, and servers.

- *Start* or *Stop* - this starts or stops the emulation, performing the same function as the green Start or red Stop button.
- *Node types...* - invokes the CORE Node Types dialog, performing the same function as the Edit button on the Network-Layer Nodes toolbar.
- *Comments...* - invokes the CORE Experiment Comments window where optional text comments may be specified. These comments are saved at the top of the imn configuration file.
- *Hooks...* - invokes the CORE Experiment Hooks window where scripts may be configured for a particular session state. The top of the window has a list of configured hooks, and buttons on the bottom left allow adding, editing, and removing hook scripts. The new or edit button will open a hook script editing window. A hook script is a shell script invoked on the host (not within a virtual node). The script is started at the session state specified in the drop down:
  1. *definition* - used by the GUI to tell the backend to clear any state.
  2. *configuration* - when the user presses the *Start* button, node, link, and other configuration data is sent to the backend. This state is also reached when the user customizes a service.
  3. *instantiation* - after configuration data has been sent, just before the nodes are created.
  4. *runtime* - all nodes and networks have been built and are running. (This is the same state at which the previously-named *global experiment script* was run.)
  5. *datacollect* - the user has pressed the *Stop* button, but before services have been stopped and nodes have been shut down. This is a good time to collect log files and other data from the nodes.
  6. *shutdown* - all nodes and networks have been shut down and destroyed.
- *Reset node positions* - if you have moved nodes around using the mouse or by using a mobility module, choosing this item will reset all nodes to their original position on the canvas. The node locations are remembered when you first press the Start button.
- *Emulation servers...* - invokes the CORE emulation servers dialog for configuring Section 2.5.5 [Distributed Emulation], page 27.
- *CORE Plugins...* - invokes the CORE Plugins dialog for managing connections with other CORE Plugins, such as the local CORE daemon (cored).
- *Sessions...* - invokes the Sessions dialog for switching between different running experiments. This dialog is presented during startup when an experiment is already running on a node.

2.3.8 Help Menu

- *Info manual* - attempts to load the `info` manual in an xterm, which is an `info` version of this document.
- *Online manual (www)*, *CORE website (www)*, *Mailing list (www)* - these options attempt to open a web browser with the link to the specified web resource.
- *About* - invokes the About dialog box for viewing version information

2.4 Connecting with Physical Networks

CORE's emulated networks run in real time, so they can be connected to live physical networks. The RJ45 tool and the Tunnel tool help with connecting to the real world. These tools are available from the *Link-layer nodes* menu.

When connecting two or more CORE emulations together, MAC address collisions should be avoided. CORE automatically assigns MAC addresses to interfaces when the emulation is started, starting with `00:00:00:aa:00:00` and incrementing the bottom byte. The starting byte should be changed on the second CORE machine using the *MAC addresses...* option from the *Tools* menu.

2.4.1 RJ45 Tool

The RJ45 node in CORE represents a physical interface on the real CORE machine. Any real-world network device can be connected to the interface and communicate with the CORE nodes in real time.

The main drawback is that one physical interface is required for each connection. When the physical interface is assigned to CORE, it may not be used for anything else. Another consideration is that the computer or network that you are connecting to must be co-located with the CORE machine.

To place an RJ45 connection, click on the *Link-layer nodes* toolbar and select the *RJ45 Tool* from the submenu. Click on the canvas near the node you want to connect to. This could be a router, hub, switch, or WLAN, for example. Now click on the *Link Tool* and draw a link between the RJ45 and the other node. The RJ45 node will display "UNASSIGNED". Double-click the RJ45 node to assign a physical interface. A list of available interfaces will be shown, and one may be selected by double-clicking its name in the list, or an interface name may be entered into the text box.

When you press the Start button to instantiate your topology, the interface assigned to the RJ45 will be connected to the CORE topology. The interface can no longer be used by the system. For example, if there was an IP address assigned to the physical interface before execution, the address will be removed and control given over to CORE. No IP address is needed; the interface is put into promiscuous mode so it will receive all packets and send them into the emulated world.

Multiple RJ45 nodes can be used within CORE and assigned to the same physical interface if 802.1x VLANs are used. This allows for more RJ45 nodes than physical ports are available, but the (e.g. switching) hardware connected to the physical port must support the VLAN tagging, and the available bandwidth will be shared.

You need to create separate VLAN virtual devices on the Linux or FreeBSD host, and then assign these devices to RJ45 nodes inside of CORE. The VLANning is actually performed outside of CORE, so when the CORE emulated node receives a packet, the VLAN tag will already be removed.

2.4.2 Tunnel Tool

The tunnel tool builds GRE tunnels between CORE emulations or other hosts. Tunneling can be helpful when the number of physical interfaces is limited or when the peer is located on a different network. Also a physical interface does not need to be dedicated to CORE as with the RJ45 tool.

The peer GRE tunnel endpoint may be another CORE machine or a (Linux, FreeBSD, etc.) host that supports GRE tunneling. When placing a Tunnel node, initially the node will display "UNASSIGNED". This text should be replaced with the IP address of the tunnel peer. This is the IP address of the other CORE machine or physical machine, not an IP address of another virtual node.

The GRE key is used to identify flows with GRE tunneling. This allows multiple GRE tunnels to exist between that same pair of tunnel peers. A unique number should be used when multiple tunnels are used with the same peer. When configuring the peer side of the tunnel, ensure that the matching keys are used.

2.4.3 Span Tool

Note: the Span Tool is currently unsupported by the GUI. The description remains here for reference.

The CORE Span Tool allows for two types of connections: connecting multiple CORE emulations together and connecting the emulated network with a virtual interface on a remote machine. Data is tunneled over TCP or UDP connections, allowing the connection to "span" across large networks.

Span is a separate userspace program that runs with CORE. This program has been ported to run on the FreeBSD, Linux, and Windows XP operating systems.

When Span is run on a FreeBSD CORE system, the emulated Netgraph nodes can be connected to Span using the in-kernel Netgraph subsystem. Span employs the ng_socket Netgraph module, providing a kernel hook on one end and userspace socket on the other end. These Netgraph sockets are then connected to other Span daemons running on other systems via TCP or UDP tunnels managed by Span.

Span runs in a different mode on Linux and Windows XP systems. There Span will create a virtual interface, using the TUN/TAP driver, connected to one end of the TCP or UDP tunnels. This provides a workaround for a limited number of interfaces on other systems that you want to connect to CORE.

The drawback of using the Span tool is that packets are copied to userspace and prepended with a tunneling header, which may cause performance and path MTU problems.

To use the Span Tool, select the *Tunnel Tool* from the *Link-layer nodes* submenu on the Editing toolbar. Place a Tunnel node and link it to any CORE node using the Link Tool. Double-click on the Tunnel node and choose your tunneling parameters.

If connecting the tunnel to another CORE system, select "tunnel to another CORE emulation" and enter the IP address of that system. In the "peer hook" text entry box, type in the name of the "local hook" that appears on the other system. By default, the hook is named by by adding the hostname to the node name. Each hook must have a unique name, and there are some limitations such as a maximum length of 16 characters.

If connecting the tunnel node to a virtual interface on a remote system, select "tunnel to the virtual TAP interface of another system" and enter the IP address of that system. In the "peer hook" text entry box, type in the name of TUN/TAP interface on the remote system, which is typically "core0". Again, each hook must have a unique name, so first configure unique device names in Span for the TUN/TAP devices on the peer systems.

The Span Tool must be running before starting any CORE scenario containing tunnels. If tunneling to a virtual interface of another system, Span should be started on that system before starting the scenario (this is because CORE will send the other machine commands for configuring the tunnel.) When tunneling between two CORE emulations, either one may be started first, as each will attempt to connect to the other if a connection does not already exist.

2.5 Building Sample Networks

2.5.1 Wired Networks

Wired networks are created using the *Link Tool* to draw a link between two nodes. This automatically draws a red line representing an Ethernet link and creates new interfaces on network-layer nodes.

Double-click on the link to invoke the *link configuration* dialog box. Here you can change the Bandwidth, Delay, PER (Packet Error Rate), and Duplicate rate parameters for that link. You can also modify the color and width of the link, affecting its display.

Link-layer nodes are provided for modeling wired networks. These do not create a separate network stack when instantiated, but are implemented using bridging (Linux) or Netgraph nodes (FreeBSD). These are the hub, switch, and wireless LAN nodes. The hub copies each packet from the incoming link to every connected link, while the switch behaves more like an Ethernet switch and keeps track of the Ethernet address of the connected peer, forwarding unicast traffic only to the appropriate ports.

The wireless LAN (WLAN) is covered in the next section.

2.5.2 Wireless Networks

The wireless LAN node allows you to build wireless networks where moving nodes around affects the connectivity between them. The wireless LAN, or WLAN, node appears as a small cloud. The WLAN offers several levels of wireless emulation fidelity, depending on your modeling needs.

The WLAN tool can be extended with plugins for different levels of wireless fidelity. The basic on/off range is the default setting available on all platforms. Other plugins offer higher fidelity at the expense of greater complexity and CPU usage. The availability of certain plugins varies depending on platform. See the table below for a brief overview of wireless model types.

Model Type	**Supported Platform(s)**	**Fidelity**	**Description**
Basic on/off	Linux, FreeBSD	Low	Linux Ethernet bridging with ebtables (Linux) or ng_wlan (FreeBSD)
EMANE Plugin	Linux	High	TAP device connected to EMANE emulator with pluggable MAC and PHY radio types

To quickly build a wireless network, you can first place several router nodes onto the canvas. If you have the Quagga MDR software installed, it is recommended that you use the *mdr* node type for reduced routing overhead. Next choose the *wireless LAN* from the

Link-layer nodes submenu. First set the desired WLAN parameters by double-clicking the cloud icon. Then you can link all of the routers by right-clicking on the WLAN and choosing *Link to all routers*.

Linking a router to the WLAN causes a small antenna to appear, but no red link line is drawn. Routers can have multiple wireless links and both wireless and wired links (however, you will need to manually configure route redistribution.) The mdr node type will generate a routing configuration that enables OSPFv3 with MANET extensions. This is a Boeing-developed extension to Quagga's OSPFv3 that reduces flooding overhead and optimizes the flooding procedure for mobile ad-hoc (MANET) networks.

The default configuration of the WLAN is set to use the basic range model, using the *Basic* tab in the WLAN configuration dialog. Having this model selected causes `cored.py` to calculate the distance between nodes based on screen pixels. A numeric range in screen pixels is set for the wireless network using the *Range* slider. When two wireless nodes are within range of each other, a green line is drawn between them and they are linked. Two wireless nodes that are farther than the range pixels apart are not linked. During Execute mode, users may move wireless nodes around by clicking and dragging them, and wireless links will be dynamically made or broken.

The *EMANE* tab lists available EMANE models to use for wireless networking. See the Chapter 5 [EMANE], page 37 chapter for details on using EMANE.

On FreeBSD, the WLAN node is realized using the `ng_wlan` Netgraph node.

2.5.3 Mobility Scripting

CORE has a few ways to script mobility.

- ns-2 scengen script - this is GUI driven, so it will not work from a headless, scripted scenario; the script specifies either absolute positions or waypoints with a velocity.
- CORE API - an external entity can move nodes by sending CORE API Node messages with updated X,Y coordinates; the `coresendmsg.py` utility allows a shell script to generate these messages.
- EMANE events - see Chapter 5 [EMANE], page 37 for details on using EMANE scripts to move nodes around.

Here, the first method using a ns-2 scengen script is described, where the Tcl/Tk GUI animates the nodes. You can create a mobility script using a text editor and associate the script with one of the wireless networks by entering the file name into the *scengen mobility script* text box in the WLAN configuration dialog box.

When the Execute mode is started and one of the WLAN nodes has a mobility script, a mobility script window will appear. This window contains controls for starting, stopping, and resetting the running time for the mobility script. The *loop* checkbox causes the script to play continuously. The *resolution* text box contains the number of milliseconds between each timer event; lower values cause the mobility to appear smoother but consumes greater CPU time.

The format of the script looks like:

```
# nodes: 3, max time: 35.000000, max x: 600.00, max y: 600.00
$node_(2) set X_ 144.0
$node_(2) set Y_ 240.0
```

```
$node_(2) set Z_ 0.00
$ns_ at 1.00 "$node_(2) setdest 130.0 280.0 15.0"
```

The *max time* is taken from the first line (other values are ignored) and represents the total time the script will run before looping or stopping. The next three lines set an initial position for node 2. The last line in the above example causes node 2 to move towards the destination (130, 280) at speed 15. All units are screen coordinates, with speed in units per second.

Examples mobility scripts (and their associated topology files) can be found in the `configs/` directory (see Section 2.8 [Configuration Files], page 31).

2.5.4 Multiple Canvases

CORE supports multiple canvases for organizing emulated nodes. Nodes running on different canvases may be linked together.

To create a new canvas, choose *New* from the *Canvas* menu. A new canvas tab appears in the bottom left corner. Clicking on a canvas tab switches to that canvas. Double-click on one of the tabs to invoke the *Manage Canvases* dialog box. Here, canvases may be renamed and reordered, and you can easily switch to one of the canvases by selecting it.

Each canvas maintains its own set of nodes and annotations. To link between canvases, select a node and right-click on it, choose *Create link to*, choose the target canvas from the list, and from that submenu the desired node. A pseudo-link will be drawn, representing the link between the two nodes on different canvases. Double-clicking on the label at the end of the arrow will jump to the canvas that it links.

2.5.5 Distributed Emulation

A large emulation scenario can be deployed on multiple emulation servers and controlled by a single GUI. The GUI, representing the entire topology, can be run on one of the emulation servers or on a separate machine. Emulations can be distributed on Linux, while tunneling support has not been added yet for FreeBSD.

Each machine that will act as an emulation server needs to have CORE installed. It is not important to have the GUI component but the CORE Python daemon `cored.py` needs to be installed. Set the `listenaddr` line in the '`/etc/core/core.conf`' configuration file so that the CORE Python daemon will respond to commands from other servers:

```
### cored.py configuration options ###
[cored.py]
pidfile = /var/run/coredpy.pid
logfile = /var/log/coredpy.log
listenaddr = 0.0.0.0
```

The `listenaddr` should be set to the address of the interface that should receive CORE API control commands from the other servers; setting `listenaddr = 0.0.0.0` causes the Python daemon to listen on all interfaces. CORE uses TCP port 4038 by default to communicate from the controlling machine (with GUI) to the emulation servers. Make sure that firewall rules are configured as necessary to allow this traffic.

In order to easily open shells on the emulation servers, the servers should be running an SSH server, and public key login should be enabled. This is accomplished by generating an SSH key for your user if you do not already have one (use `ssh-keygen -t rsa`), and then copying your public key to the authorized_keys file on the server (for example, `scp ~/.ssh/id_rsa.pub server:.ssh/authorized_keys`.) When double-clicking on a node during runtime, instead of opening a local shell, the GUI will attempt to SSH to the emulation server to run an interactive shell. The user name used for these remote shells is the same user that is running the CORE GUI.

Servers are configured by choosing *Emulation servers...* from the *Experiment* menu. Servers parameters are configured in the list below and stored in a `servers.conf` file for use in different scenarios. The IP address and port of the server must be specified. The name of each server will be saved in the topology file as each node's location.

The user needs to assign nodes to emulation servers in the scenario. Making no assignment means the node will be emulated locally, on the same machine that the GUI is running. In the configuration window of every node, a drop-down box located between the *Node name* and the *Image* button will select the name of the emulation server. By default, this menu shows *(none)*, indicating that the node will be emulated locally. When entering Execute mode, the CORE GUI will deploy the node on its assigned emulation server.

Another way to assign emulation servers is to select one or more nodes using the select tool (shift-click to select multiple), and right-click one of the nodes and choose *Assign to...*.

The *CORE emulation servers* dialog box may also be used to assign nodes to servers. The assigned server name appears in parenthesis next to the node name. To assign all nodes to one of the servers, click on the server name and then the *all nodes* button. Servers that have assigned nodes are shown in blue in the server list. Another option is to first select a subset of nodes, then open the *CORE emulation servers* box and use the *selected nodes* button.

The emulation server machines should be reachable on the specified port and via SSH. SSH is used when double-clicking a node to open a shell, the GUI will open an SSH prompt to that node's emulation server. Public-key authentication should be configured so that SSH passwords are not needed.

If there is a link between two nodes residing on different servers, the GUI will draw the link with a dashed line, and automatically create necessary tunnels between the nodes when executed. Care should be taken to arrange the topology such that the number of tunnels is minimized. The tunnels carry data between servers to connect nodes as specified in the topology. These tunnels are created using GRE tunneling, similar to the Section 2.4.2 [Tunnel Tool], page 23.

Wireless nodes, i.e. those connected to a WLAN node, can be assigned to different emulation servers and participate in the same wireless network if an EMANE model is used for the WLAN. See Section 5.4 [Distributed EMANE], page 40 for more details.

2.6 Services

CORE uses the concept of services to specify what processes or scripts run on a node when it is started. Layer-3 nodes such as routers and PCs are defined by the services that they run. The Section 1.4 [Quagga Routing Software], page 11, for example, transforms a node into a router.

Services may be customized for each node, or new custom services can be created. New node types can be created each having a different name, icon, and set of default services.

Each service defines the per-node directories, configuration files, startup index, starting commands, validation commands, shutdown commands, and meta-data associated with a node.

2.6.1 Default Services and Node Types

Here are the default node types and their services:

- *router* - zebra, OSFPv2, OSPFv3, vtysh, and IPForward services for IGP link-state routing.
- *host* - DefaultRoute and SSH services, representing an SSH server having a default route when connected directly to a router.
- *PC* - DefaultRoute service for having a default route when connected directly to a router.
- *mdr* - zebra, OSPFv3MDR, vtysh, and IPForward services for wireless-optimized MANET Designated Router routing.
- *prouter* - a physical router, having the same default services as the *router* node type; for incorporating Linux testbed machines into an emulation, the Chapter 4 [Machine Types], page 35 is set to Section 4.2 [physical], page 35.

Configuration files can be automatically generated by each service. For example, CORE automatically generates routing protocol configuration for the router nodes in order to simplify the creation of virtual networks.

To change the services associated with a node, double-click on the node to invoke its configuration dialog and click on the *Services...* button. Services are enabled or disabled by clicking on their names. The button next to each service name allows you to customize all aspects of this service for this node. For example, special route redistribution commands could be inserted in to the Quagga routing configuration associated with the zebra service.

To change the default services associated with a node type, use the Node Types dialog available from the *Edit* button at the end of the Layer-3 nodes toolbar, or choose *Node types...* from the *Experiment* menu. Note that any new services selected are not applied to existing nodes if the nodes have been customized.

The node types are saved in a '`~/.core/nodes.conf`' file, not with the `.imn` file. Keep this in mind when changing the default services for existing node types; it may be better to simply create a new node type. It is recommended that you do not change the default built-in node types. The '`nodes.conf`' file can be copied between CORE machines to save your custom types.

2.6.2 Customizing a Service

A service can be fully customized for a particular node. From the node's configuration dialog, click on the button next to the service name to invoke the service customization dialog for that service. The dialog has three tabs for configuring the different aspects of the service: files, directories, and startup/shutdown.

The Files tab is used to display or edit the configuration files or scripts that are used for this service. Files can be selected from a drop-down list, and their contents are displayed in a text entry below. The file contents are generated by the CORE Services based on the network topology that exists at the time the customization dialog is invoked.

The Directories tab shows the per-node directories for this service. For the default types, CORE nodes share the same filesystem tree, except for these per-node directories that are defined by the services. For example, the `/var/run/quagga` directory needs to be unique for each node running the Zebra service, because Quagga running on each node needs to write separate PID files to that directory.

The Startup/shutdown tab lists commands that are used to start and stop this service. The startup index allows configuring when this service starts relative to the other services enabled for this node; a service with a lower startup index value is started before those with higher values. Because shell scripts generated by the Files tab will not have execute permissions set, the startup commands should include the shell name, with something like `"sh script.sh"`.

Shutdown commands optionally terminate the process(es) associated with this service. Generally they send a kill signal to the running process using the `kill` or `killall` commands. If the service does not terminate the running processes using a shutdown command, the processes will be killed when the `vnoded` daemon is terminated (with `kill -9` and the namespace destroyed. It is a good practice to specify shutdown commands, which will allow for proper process termination, and for run-time control of stopping and restarting services in future versions.

Validate commands are executed following the startup commands. A validate command can execute a process or script that should return zero if the service has started successfully, and have a non-zero return value for services that have had a problem starting. For example, the `pidof` command will check if a process is running and return zero when found. When a validate command produces a non-zero return value, an exception is generated, which will cause an error to be displayed in the Section 2.7 [Check Emulation Light], page 30.

2.6.3 Creating new Services

Services can save time required to configure nodes, especially if a number of nodes require similar configuration procedures. New services can be introduced to automate tasks.

The easiest way to capture the configuration of a new process into a service is by using the **UserDefined** service. This is a blank service where any aspect can be customized. The UserDefined service is convenient for testing ideas for a service before adding a new service type.

To introduce new service types, a '`myservices/`' directory exists in the user's CORE configuration directory, at '`~/.core/myservices/`'. A detailed '`README.txt`' file exists in that directory to outline the steps necessary for adding a new service. First, you need to create a small Python file that defines the service; then the `custom_services_dir` entry must be set in the '`/etc/core/core.conf`' configuration file. A sample is provided in the '`myservices/`' directory.

If you have created a new service type that may be useful to others, please consider contributing it to the CORE project.

2.7 Check Emulation Light

The Check Emulation Light, or CEL, is located in the bottom right-hand corner of the status bar in the CORE GUI. This is a yellow icon that indicates one or more problems with the running emulation. Clicking on the CEL will invoke the CEL dialog.

The Check Emulation Light dialog contains a list of exceptions received from the CORE daemon. An exception has a time, severity level, optional node number, and source. When the CEL is blinking, this indicates one or more fatal exceptions. An exception with a fatal severity level indicates that one or more of the basic pieces of emulation could not be created, such as failure to create a bridge or namespace, or the failure to launch EMANE processes for an EMANE-based network.

Clicking on an exception displays details for that exception. If a node number is specified, that node is highlighted on the canvas when the exception is selected. The exception source is a text string to help trace where the exception occurred; "service:UserDefined" for example, would appear for a failed validation command with the UserDefined service.

Buttons are available at the bottom of the dialog for clearing the exception list and for viewing the CORE daemon and node log files.

2.8 Configuration Files

Configurations are saved to `.imn` topology files using the *File* menu. You can easily edit these files with a text editor. Some features are only available by editing the topology file directly. Any time you edit the topology file, you will need to stop the emulation if it were running and reload the file.

Tabs and spacing in the topology files are important. The file starts by listing every node, then links, annotations, canvases, and options. Each entity has a block contained in braces. The first block is indented by four spaces. Within the `network-config` block (and any *custom-*-config* block), the indentation is one tab character.

There are several topology examples included with CORE in the `configs/` directory. This directory can be found in `~/.core/configs`.

2.9 Customizing your Topology's Look

Several annotation tools are provided for changing the way your topology is presented. Captions may be added with the Text tool. Ovals and rectangles may be drawn in the background, helpful for visually grouping nodes together.

During live demonstrations the marker tool may be helpful for drawing temporary annotations on the canvas that may be quickly erased. A size and color palette appears at the bottom of the toolbar when the marker tool is selected. Markings are only temporary and are not saved in the topology file.

The basic node icons can be replaced with a custom image of your choice. Icons appear best when they use the GIF or PNG format with a transparent background. To change a node's icon, double-click the node to invoke its configuration dialog and click on the button to the right of the node name that shows the node's current icon.

A background image for the canvas may be set using the *Wallpaper...* option from the *Canvas* menu. The image may be centered, tiled, or scaled to fit the canvas size. An existing terrain, map, or network diagram could be used as a background, for example, with CORE nodes drawn on top.

2.10 Preferences

The *Preferences* Dialog can be accessed from the Section 2.3.2 [Edit Menu], page 18. There are numerous defaults that can be set with this dialog, which are stored in the `~/.core/prefs.conf` preferences file.

3 Python Scripting

CORE can be used via the Chapter 2 [Using the CORE GUI], page 15 or Python scripting. Writing your own Python scripts offers a rich programming environment with complete control over all aspects of the emulation. This chapter provides a brief introduction to scripting. Most of the documentation is available from sample scripts, or online via interactive Python.

The best starting point is the sample scripts that are included with CORE. If you have a CORE source tree, the example script files can be found under '`core/python/examples/netns/`'. When CORE is installed from packages, the example script files will be in '`/usr/share/core/examples/netns/`' (or the '`/usr/local/...`' prefix when installed from source.) For the most part, the example scripts are self-documenting; see the comments contained within the Python code.

The scripts should be run with root priveleges because they create new network namespaces. In general, a CORE Python script does not connect to the CORE services daemon, '`cored.py`'; in fact, '`cored.py`' is just another Python script that uses the CORE Python modules and exchanges messages with the GUI. To connect the GUI to your scripts, see the included sample scripts that allow for GUI connections.

Here are the basic elements of a CORE Python script:

```
#!/usr/bin/python

from core import pycore

session = pycore.Session(persistent=True)
node1 = session.addobj(cls=pycore.nodes.CoreNode, name="n1")
node2 = session.addobj(cls=pycore.nodes.CoreNode, name="n2")
hub1 = session.addobj(cls=pycore.nodes.HubNode, name="hub1")
node1.newnetif(hub1, ["10.0.0.1/24"])
node2.newnetif(hub1, ["10.0.0.2/24"])

node1.icmd(["ping", "-c", "5", "10.0.0.2"])
session.shutdown()
```

The above script creates a CORE session having two nodes connected with a hub. The first node pings the second node with 5 ping packets; the result is displayed on screen.

A good way to learn about the CORE Python modules is via interactive Python. Scripts can be run using `python -i`. Cut and paste the simple script above and you will have two nodes connected by a hub, with one node running a test ping to the other.

The CORE Python modules are documented with comments in the code. From an interactive Python shell, you can retrieve online help about the various classes and methods; for example `help(pycore.nodes.CoreNode)` or `help(pycore.Session)`.

An interactive development environment (IDE) is available for browsing the CORE source, the Eric Python IDE. CORE has a project file that can be opened by Eric, in the source under '`core/python/CORE.e4p`'. This IDE has a class browser for viewing a tree of classes and methods. It features syntax highlighting, auto-completion, indenting, and more. One feature that is helpful with learning the CORE Python modules is the ability to generate class diagrams; right-click on a class, choose *Diagrams*, and *Class Diagram*.

4 Machine Types

Different node types can be configured in CORE, and each node type has a *machine type* that indicates how the node will be represented at run time. Different machine types allow for different virtualization options.

4.1 netns

The *netns* machine type is the default. This is for nodes that will be backed by Linux network namespaces. See [Linux], page 2 for a brief explanation of netns. This default machine type is very lightweight, providing a minimum amount of virtualization in order to emulate a network. Another reason this is designated as the default machine type is because this virtualization technology typically requires no changes to the kernel; it is available out-of-the-box from the latest mainstream Linux distributions.

4.2 physical

The *physical* machine type is used for nodes that represent a real Linux-based machine that will participate in the emulated network scenario. This is typically used, for example, to incorporate racks of server machines from an emulation testbed. A physical node is one that is running the CORE daemon (`cored.py`), but will not be further partitioned into virtual machines. Services that are run on the physical node do not run in an isolated or virtualized environment, but directly on the operating system.

Physical nodes must be assigned to servers, the same way nodes are assigned to emulation servers with Section 2.5.5 [Distributed Emulation], page 27. The list of available physical nodes currently shares the same dialog box and list as the emulation servers, accessed using the *Emulation Servers...* entry from the *Experiment* menu.

Support for physical nodes is under development and may be improved in future releases. Currently, when any node is linked to a physical node, a dashed line is drawn to indicate network tunneling. A GRE tunneling interface will be created on the physical node and used to tunnel traffic to and from the emulated world.

Double-clicking on a physical node during runtime opens an xterm with an SSH shell to that node. Users should configure public-key SSH login as done with emulation servers.

4.3 xen

The *xen* machine type is only available from an experimental branch of the CORE development tree. When the xen branch has been merged with the main line of CORE development, it may be documented here.

5 EMANE

This chapter describes running CORE with the EMANE emulator.

5.1 What is EMANE?

The Extendable Mobile Ad-hoc Network Emulator (EMANE) allows heterogeneous network emulation using a pluggable MAC and PHY layer architecture. The EMANE framework provides an implementation architecture for modeling different radio interface types in the form of *Network Emulation Modules* (NEMs) and incorporating these modules into a real-time emulation running in a distributed environment.

EMANE is developed by U.S. Naval Research Labs (NRL) Code 5522 and CenGen Inc., who maintain these websites:

- `http://cs.itd.nrl.navy.mil/work/emane/index.php`
- `http://labs.cengen.com/emane/`

Instead of building Linux Ethernet bridging networks with CORE, higher-fidelity wireless networks can be emulated using EMANE bound to virtual devices. CORE emulates layers 3 and above (network, session, application) with its virtual network stacks and process space for protocols and applications, while EMANE emulates layers 1 and 2 (physical and data link) using its pluggable PHY and MAC models.

The interface between CORE and EMANE is a TAP device. CORE builds the virtual node using Linux network namespaces, and installs the TAP device into the namespace. EMANE binds a userspace socket to the device, on the host before it is pushed into the namespace, for sending and receiving data. The *Virtual Transport* is the EMANE component responsible for connecting with the TAP device.

EMANE models are configured through CORE's WLAN configuration dialog. A corresponding EmaneModel Python class is sub-classed for each supported EMANE model, to provide configuration items and their mapping to XML files. This way new models can be easily supported. When CORE starts the emulation, it generates the appropriate XML files that specify the EMANE NEM configuration, and launches the EMANE daemons.

Some EMANE models support location information to determine when packets should be dropped. EMANE has an event system where location events are broadcast to all NEMs. CORE can generate these location events when nodes are moved on the canvas. The canvas size and scale dialog has controls for mapping the X,Y coordinate system to a latitude, longitude geographic system that EMANE uses. When specified in the '`core.conf`' configuration file, CORE can also subscribe to EMANE location events and move the nodes on the canvas as they are moved in the EMANE emulation. This would occur when an Emulation Script Generator, for example, is running a mobility script.

5.2 EMANE Configuration

CORE and EMANE currently work together only on the Linux network namespaces platform. The normal CORE installation instructions should be followed from see Chapter 1 [Installation], page 5.

The CORE configuration file '`/etc/core/core.conf`' has options specific to EMANE. Namely, the `emane_models` line contains a comma-separated list of EMANE models that will be available. Each model has a corresponding Python file containing the EmaneModel subclass. The default 'core.conf' file is shown below:

```
### cored.py configuration options ###
[cored.py]
pidfile = /var/run/coredpy.pid
logfile = /var/log/coredpy.log
listenaddr = localhost
port = 4038
numthreads = 1
verbose = False
# EMANE configuration
emane_platform_port = 8101
emane_transform_port = 8201
emane_event_monitor = False
emane_models = RfPipe, Ieee80211abg
```

EMANE can be installed from RPM packages or from source. See the EMANE website for full details. If you do not want to install all of the EMANE packages, the typical packages are EMANE, Utilities, IEEE 802.11abg Model, RFPipe Model, DTD, Transport Daemon, Virtual Transport, Event Service, Emulation Script Generator, ACE, ACE gperf, and the add-ons Emane Event Service Library, python-EventService and python-Location bindings.

Here are quick instructions for installing all EMANE packages:

```
# install dependencies
yum -y install openssl-devel perl-XML-Simple perl-XML-LibXML
# download and install EMANE 0.7.2
wget http://labs.cengen.com/emane/download/RPMS/F14/0.7.2/i386/\
emane-bundle-0.7.2.fc14.i386.tgz
mkdir emane-0.7.2
cd emane-0.7.2
tar xzf ../emane-bundle-0.7.2.fc14.i386.tgz
# ACE libs must be installed first
rpm -ivh ace*
rm -f ace*
rpm -ivh *
```

If you have an EMANE event generator (e.g. mobility or pathloss scripts) and want to have CORE subscribe to EMANE location events, set the following line in the '/etc/core/core.conf' configuration file:

```
emane_event_monitor = True
```

Do not set the above option to True if you want to manually drag nodes around on the canvas to update their location in EMANE.

Another common issue is if installing EMANE from source, the default configure prefix will place the DTD files in '/usr/local/share/emane/dtd' while CORE expects them in '/usr/share/emane/dtd'. A symbolic link will fix this:

```
sudo ln -s /usr/local/share/emane /usr/share/emane
```

5.3 Single PC with EMANE

This section describes running CORE and EMANE on a single machine. This is the default mode of operation when building an EMANE network with CORE. The OTA manager interface is off and the virtual nodes use the loopback device for communicating with one another. This prevents your emulation experiment from sending data on your local network and interfering with other EMANE users.

EMANE is configured through a WLAN node, because it is all about emulating wireless radio networks. Once a node is linked to a WLAN cloud configured with an EMANE model, the radio interface on that node may also be configured separately (apart from the cloud.)

Double-click on a WLAN node to invoke the WLAN configuration dialog. Click the *EMANE* tab; when EMANE has been properly installed, EMANE wireless modules should be listed in the *EMANE Models* list. (You may need to restart the CORE services if they were running prior to installing the EMANE Python bindings.) Click on a model name to enable it.

When an EMANE model is selected in the *EMANE Models* list, clicking on the *model options* button causes the GUI to query the CORE services for configuration items. Each model will have different parameters, refer to the EMANE documentation for an explanation of each item. The defaults values are presented in the dialog. Clicking *Apply* and *Apply* again will store the EMANE model selections.

The *EMANE options* button allows specifying some global parameters for EMANE, some of which are necessary for distributed operation, see Section 5.4 [Distributed EMANE], page 40.

The RF-PIPE and IEEE 802.11abg models use a Universal PHY that supports geographic location information for determining pathloss between nodes. A default latitude and longitude location is provided by CORE and this location-based pathloss is enabled by default; this is the *pathloss mode* setting for the Universal PHY. Moving a node on the canvas while the emulation is running generates location events for EMANE. To view or change the geographic location or scale of the canvas use the *Canvas Size and Scale* dialog available from the *Canvas* menu.

Clicking the green *Start* button launches the emulation and causes TAP devices to be created in the virtual nodes that are linked to the EMANE WLAN. These devices appear with interface names such as eth0, eth1, etc. The EMANE daemons should now be running on the host:

```
> ps -aef | grep emane
root   10472   1  1 12:57 ?   00:00:00 emane --logl 0 platform.xml
root   10526   1  1 12:57 ?   00:00:00 emanetransportd --logl 0 tr
```

The above example shows the `emane` and `emanetransportd` daemons started by CORE. To view the configuration generated by CORE, look in the '`/tmp/pycore.nnnnn/`' session directory for a '`platform.xml`' file and other XML files. One easy way to view this information is by double-clicking one of the virtual nodes, and typing `cd ..` in the shell to go up to the session directory.

When EMANE is used to network together CORE nodes, no Ethernet bridging device is used. The Virtual Transport creates a TAP device that is installed into the network namespace container, so no corresponding device is visible on the host.

5.4 Distributed EMANE

Running CORE and EMANE distributed among two or more emulation servers is similar to running on a single machine. There are a few key configuration items that need to be set in order to be successful, and those are outlined here.

Because EMANE uses a multicast channel to disseminate data to all NEMs, it is a good idea to maintain separate networks for data and control. The control network may be a shared laboratory network, for example, but you do not want multicast traffic on the data network to interfere with other EMANE users. The examples described here will use `eth0` as a control interface and `eth1` as a data interface, although using separate interfaces is not strictly required.

Each machine that will act as an emulation server needs to have CORE and EMANE installed. Refer to the Section 2.5.5 [Distributed Emulation], page 27 section for configuring CORE.

The IP addresses of the available servers are configured from the CORE emulation servers dialog box (choose *Experiment* then *Emulation servers...*) described in Section 2.5.5 [Distributed Emulation], page 27. This list of servers is stored in a '`~/.core/servers.conf`' file. The dialog shows available servers, some or all of which may be assigned to nodes on the canvas.

Nodes need to be assigned to emulation servers as described in Section 2.5.5 [Distributed Emulation], page 27. Select several nodes, right-click them, and choose *Assign to* and the name of the desired server. When a node is not assigned to any emulation server, it will be emulated locally. The local machine that the GUI connects with is considered the "master" machine, which in turn connects to the other emulation server "slaves". Public key SSH should be configured from the master to the slaves as mentioned in the Section 2.5.5 [Distributed Emulation], page 27 section.

The EMANE models can be configured as described in Section 5.3 [Single PC with EMANE], page 39. When the Available Plugins dialog box is open for configuring EMANE models, enable the *Emulation Server - emane* item. Click on this item in the *Active capabilities* list and click the *Configure...* button. This brings up the emane configuration dialog. The *enable OTA Manager channel* should be set to *on*. The *OTA Manager device* and *Event Service device* should be set to something other than the loopback *lo* device. For example, if eth0 is your control device and eth1 is for data, set the OTA Manager device to eth1 and the Event Service device to eth0. Click *Apply*, *OK*, and *Apply* to save these settings.

Now when the Start button is used to instantiate the emulation, the local CORE Python daemon will connect to other emulation servers that have been assigned to nodes. Each server will have its own session directory where the '`platform.xml`' file and other EMANE XML files are generated. The NEM IDs are automatically coordinated across servers so there is no overlap. Each server also gets its own Platform ID.

Instead of using the loopback device for disseminating multicast EMANE events, an Ethernet device is used as specified in the *configure emane* dialog. EMANE's Event Service can be run with mobility or pathloss scripts as described in Section 5.3 [Single PC with EMANE], page 39. If CORE is not subscribed to location events, it will generate them as nodes are moved on the canvas.

Double-clicking on a node during runtime will cause the GUI to attempt to SSH to the emulation server for that node and run an interactive shell. The public key SSH configuration should be tested with all emulation servers prior to starting the emulation.

6 ns-3

This chapter describes running CORE with the ns-3 simulator.

6.1 What is ns-3?

ns-3 is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use[1].

CORE can run in conjunction with ns-3 to simulate some types of networks. CORE network namespace virtual nodes can have virtual TAP interfaces installed using the simulator for communication. The simulator needs to run at wall clock time with the real-time scheduler.

Users simulate networks with ns-3 by writing C++ or Python scripts that import the ns-3 library. Simulation models are objects instantiated in these scripts. Combining the CORE Python modules with ns-3 Python bindings allow a script to easily set up and manage an emulation + simulation environment.

6.2 ns-3 Scripting

Currently, ns-3 is supported at the Python scripting level, not within the GUI. If you have a copy of the CORE source, look under '`core/python/ns3/examples/`' for example scripts; a CORE installation package puts these under '`/usr/share/core/examples/corens3`'.

To run these scripts, install CORE so the CORE Python libraries are accessible, and download and build ns-3. This has been tested using ns-3 3.11, 3.12.1, and 3.13. Open a waf shell as root, so that network namespaces may be instantiated by the script.

```
> cd ns-allinone-3.13/ns-3.13
> sudo ./waf shell
# # use '/usr/local' below if installed from source
# cd /usr/share/core/examples/corens3/
# python -i ns3wifi.py
running ns-3 simulation for 600 seconds

>>> print session
<corens3.obj.Ns3Session object at 0x1963e50>
>>>
```

The interactive Python shell allows some interaction with the Python objects for the emulation.

In another terminal, nodes can be accessed using `vcmd`:

```
vcmd -c /tmp/pycore.10781/n1 -- bash
root@n1:/tmp/pycore.10781/n1.conf#
root@n1:/tmp/pycore.10781/n1.conf# 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_req=1 ttl=64 time=7.99 ms
64 bytes from 10.0.0.3: icmp_req=2 ttl=64 time=3.73 ms
```

[1] http://www.nsnam.org

```
64 bytes from 10.0.0.3: icmp_req=3 ttl=64 time=3.60 ms
^C
--- 10.0.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 3.603/5.111/7.993/2.038 ms
root@n1:/tmp/pycore.10781/n1.conf#
```

The ping packets shown above are traversing an ns-3 ad-hoc Wifi simulated network.

To clean up the experiment, use the Session.shutdown() method from the Python terminal.

```
>>> print session
<corens3.obj.Ns3Session object at 0x1963e50>
>>>
>>> session.shutdown()
>>>
```

A CORE/ns-3 Python script will instantiate an Ns3Session, which is a CORE Session having CoreNs3Nodes, an ns-3 MobilityHelper, and a fixed duration. The CoreNs3Node inherits from both the CoreNode and the ns-3 Node classes – it is a network namespace having an associated simulator object. The CORE TunTap interface is used, represented by a ns-3 TapBridge in `CONFIGURE_LOCAL` mode, where ns-3 creates and configures the tap device. An event is scheduled to install the taps at time 0.

6.3 Under Development

Support for ns-3 is new in this release and still under active development. Improved support may be found in the development snapshots available on the web.

The following limitations will be addressed in future releases:

- GUI configuration and control - currently ns-3 networks can only be instantiated from a Python script.
- Location - the ns-3 mobility model governs the node location, and this is not yet integrated with the GUI display (if you were to connect the CORE GUI with the running Python script). Dragging a node on the canvas would not affect the simulated node position.
- Model support - currently the WiFi model is supported. The WiMAX and 3GPP LTE models have been experimented with, but are not currently working with the TapBridge device.

7 Performance

The top question about the performance of CORE is often *how many nodes can it handle?* The answer depends on several factors:

- Hardware - the number and speed of processors in the computer, the available processor cache, RAM memory, and front-side bus speed may greatly affect overall performance.
- Operating system version - Linux or FreeBSD, and the specific kernel versions used will affect overall performance.
- Active processes - all nodes share the same CPU resources, so if one or more nodes is performing a CPU-intensive task, overall performance will suffer.
- Network traffic - the more packets that are sent around the virtual network increases the amount of CPU usage.
- GUI usage - widgets that run periodically, mobility scenarios, and other GUI interactions generally consume CPU cycles that may be needed for emulation.

On a typical single-CPU Xeon 3.0GHz server machine with 2GB RAM running FreeBSD 9.0, we have found it reasonable to run 30-75 nodes running OSPFv2 and OSPFv3 routing. On this hardware CORE can instantiate 100 or more nodes, but at that point it becomes critical as to what each of the nodes is doing.

Because this software is primarily a network emulator, the more appropriate question is *how much network traffic can it handle?* On the same 3.0GHz server described above, running FreeBSD 4.11, about 300,000 packets-per-second can be pushed through the system. The number of hops and the size of the packets is less important. The limiting factor is the number of times that the operating system needs to handle a packet. The 300,000 pps figure represents the number of times the system as a whole needed to deal with a packet. As more network hops are added, this increases the number of context switches and decreases the throughput seen on the full length of the network path.

For a more detailed study of performance in CORE, refer to the following publications:

- J. Ahrenholz, T. Goff, and B. Adamson, Integration of the CORE and EMANE Network Emulators, Proceedings of the IEEE M ilitary Communications Conference 2011, November 2011.
- Ahrenholz, J., Comparison of CORE Network Emulation Platforms, Proceedings of the IEEE Military Communications Conference 2010, pp. 864-869, November 2010.
- J. Ahrenholz, C. Danilov, T. Henderson, and J.H. Kim, CORE: A real-time network emulator, Proceedings of IEEE MILCOM Conference, 2008.

8 Developer's Guide

This section contains advanced usage information, intended for developers and others who are comfortable with the command line.

8.1 Coding Standard

The coding standard and style guide for the CORE project are posted online. Please refer to the coding standard posted on the CORE Wiki.

8.2 Source Code Guide

The CORE source consists of several different programming languages for historical reasons. Current development focuses on the Python modules and daemon. Here is a brief description of the source directories.

These are being actively developed as of CORE 4.3:

- *gui* - Tcl/Tk GUI. This uses Tcl/Tk because of its roots with the IMUNES project.
- *python* - Python modules are found in the '`python/core`' directory, the daemon under '`python/sbin/cored.py`', and Python extension modules for Linux Network Namespace support are in '`python/src`'.
- *doc* - Documentation for the manual lives here in texinfo format.
- *packaging* - Control files and script for building CORE packages are here.

These directories are not so actively developed:

- *kernel* - patches and modules mostly related to FreeBSD.
- *span* - a C daemon for building user-space tunnels to CORE emulations. In CORE 4.1 this was deprecated in favor of kernel-based GRE tunneling.
- *coreapi* - a C library used by the `cored` and `coreexecd` daemons, and `coreapisend` utility. The more up-to-date Python implementation of the CORE API can be found under '`python/core/api`'.
- *cored* - contains the `cored` and `coreexecd` C daemons supporting OpenVZ virtualization and enhanced link effects under FreeBSD.

8.3 The CORE API

The CORE API is used between different components of CORE for communication. The GUI communicates with the CORE Services using the API. One emulation server communicates with another using the API. The API also allows other systems to interact with the CORE emulation. The API allows another system to add, remove, or modify nodes and links, and enables executing commands on the emulated systems. On FreeBSD, the API is used for enhancing the wireless LAN calculations. Wireless link parameters are updated on-the-fly based on node positions.

CORE listens on a local TCP port for API messages. The other system could be software running locally or another machine accessible across the network.

The CORE API is currently specified in a separate document, available from the CORE website.

8.4 Linux network namespace Commands

Linux network namespace containers are often managed using the *Linux Container Tools* or *lxc-tools* package. The lxc-tools website is available here http://lxc.sourceforge.net/ for more information. CORE does not use these management utilities, but includes its own set of tools for instantiating and configuring network namespace containers. This section describes these tools.

The `vnoded` daemon is the program used to create a new namespace, and listen on a control channel for commands that may instantiate other processes. This daemon runs as PID 1 in the container. It is launched automatically by the CORE services. The control channel is a UNIX domain socket usually named '`/tmp/pycore.23098/n3`', for node 3 running on CORE session 23098, for example. Root privileges are required for creating a new namespace.

The `vcmd` program is used for running commands in a Linux network namespace container. This utility is run automatically by the CORE services for setting up a node and running processes within it. This program has two required arguments, the control channel name, and the command line to be run within the namespace. This command does not need to run with root privileges.

When you double-click on a node in a running emulation, CORE will open a shell window for that node using a command such as:

```
xterm -sb -right -T "CORE: n1" -e vcmd -c /tmp/pycore.50160/n1 -- bash
```

Similarly, the IPv4 routes Observer Widget will run a command to display the routing table using a command such as:

```
vcmd -c /tmp/pycore.50160/n1 -- /sbin/ip -4 ro
```

A script named `core-cleanup.sh` is provided to clean up any running CORE emulations. It will attempt to kill any remaining vnoded processes, kill any EMANE processes, remove the '`/tmp/pycore.*`' session directories, and remove any bridges or `ebtables` rules. With a `-d` option, it will also kill any running CORE Python daemon.

The `netns` command is not used by CORE directly. This utility can be used to run a command in a new network namespace for testing purposes. It does not open a control channel for receiving further commands.

Here are some other Linux commands that are useful for managing the Linux network namespace emulation.

```
# view the Linux bridging setup
brctl show
# view the netem rules used for applying link effects
tc qdisc show
# view the rules that make the wireless LAN work
ebtables -L
```

Below is a transcript of creating two emulated nodes and connecting them together with a wired link:

```
# create node 1 namespace container
vnoded -c /tmp/n1.ctl -l /tmp/n1.log -p /tmp/n1.pid
# create a virtual Ethernet (veth) pair, installing one end into node 1
ip link add name n1.0.1 type veth peer name n1.0
```

```
ip link set n1.0 netns `cat /tmp/n1.pid`
vcmd -c /tmp/n1.ctl -- ip link set n1.0 name eth0
vcmd -c /tmp/n1.ctl -- ifconfig eth0 10.0.0.1/24

# create node 2 namespace container
vnoded -c /tmp/n2.ctl -l /tmp/n2.log -p /tmp/n2.pid
# create a virtual Ethernet (veth) pair, installing one end into node 2
ip link add name n2.0.1 type veth peer name n2.0
ip link set n2.0 netns `cat /tmp/n2.pid`
vcmd -c /tmp/n2.ctl -- ip link set n2.0 name eth0
vcmd -c /tmp/n2.ctl -- ifconfig eth0 10.0.0.2/24

# bridge together nodes 1 and 2 using the other end of each veth pair
brctl addbr b.1.1
brctl setfd b.1.1 0
brctl addif b.1.1 n1.0.1
brctl addif b.1.1 n2.0.1
ip link set n1.0.1 up
ip link set n2.0.1 up
ip link set b.1.1 up

# display connectivity and ping from node 1 to node 2
brctl show
vcmd -c /tmp/n1.ctl -- ping 10.0.0.2
```

The above example script can be found as '`twonodes.sh`' in the '`examples/netns`' directory. Use `core-cleanup.sh` to clean up after the script.

8.5 FreeBSD Commands

8.5.1 FreeBSD Kernel Commands

The FreeBSD kernel emulation controlled by CORE is realized through several userspace commands. The CORE GUI itself could be thought of as a glorified script that dispatches these commands to build and manage the kernel emulation.

- **vimage** - the vimage command, short for "virtual image", is used to create lightweight virtual machines and execute commands within the virtual image context. On a FreeBSD CORE machine, see the `vimage(8)` man page for complete details. The vimage command comes from the VirtNet project which virtualizes the FreeBSD network stack.
- **ngctl** - the ngctl command, short for "netgraph control", creates Netgraph nodes and hooks, connects them together, and allows for various interactions with the Netgraph nodes. See the `ngctl(8)` man page for complete details. The ngctl command is built-in to FreeBSD because the Netgraph system is part of the kernel.

Both commands must be run as root. Some example usage of the `vimage` command follows below.

```
vimage # displays the current virtual image
```

```
vimage -l # lists running virtual images
vimage e0_n0 ps aux # list the processes running on node 0
for i in 1 2 3 4 5
do # execute a command on all nodes
  vimage e0_n$i sysctl -w net.inet.ip.redirect=0
done
```

The `ngctl` command is more complex, due to the variety of Netgraph nodes available and each of their options.

```
ngctl l # list active Netgraph nodes
ngctl show e0_n8:  # display node hook information
ngctl msg e0_n0-n1: getstats # get pkt count statistics from a pipe node
ngctl shutdown \[0x0da3\]: # shut down unnamed node using hex node ID
```

There are many other combinations of commands not shown here. See the online manual (man) pages for complete details.

Below is a transcript of creating two emulated nodes, `router0` and `router1`, and connecting them together with a link:

```
# create node 0
vimage -c e0_n0
vimage e0_n0 hostname router0
ngctl mkpeer eiface ether ether
vimage -i e0_n0 ngeth0 eth0
vimage e0_n0 ifconfig eth0 link 40:00:aa:aa:00:00
vimage e0_n0 ifconfig lo0 inet localhost
vimage e0_n0 sysctl net.inet.ip.forwarding=1
vimage e0_n0 sysctl net.inet6.ip6.forwarding=1
vimage e0_n0 ifconfig eth0 mtu 1500

# create node 1
vimage -c e0_n1
vimage e0_n1 hostname router1
ngctl mkpeer eiface ether ether
vimage -i e0_n1 ngeth1 eth0
vimage e0_n1 ifconfig eth0 link 40:00:aa:aa:0:1
vimage e0_n1 ifconfig lo0 inet localhost
vimage e0_n1 sysctl net.inet.ip.forwarding=1
vimage e0_n1 sysctl net.inet6.ip6.forwarding=1
vimage e0_n1 ifconfig eth0 mtu 1500

# create a link between n0 and n1
ngctl mkpeer eth0@e0_n0: pipe ether upper
ngctl name eth0@e0_n0:ether e0_n0-n1
ngctl connect e0_n0-n1: eth0@e0_n1: lower ether
ngctl msg e0_n0-n1: setcfg \
  {{ bandwidth=100000000 delay=0  upstream={ BER=0 dupl
icate=0 }  downstream={ BER=0 duplicate=0 } }}
ngctl msg e0_n0-n1: setcfg {{ downstream={ fifo=1 } }}
```

```
ngctl msg e0_n0-n1: setcfg {{ downstream={ droptail=1 } }}
ngctl msg e0_n0-n1: setcfg {{ downstream={ queuelen=50 } }}
ngctl msg e0_n0-n1: setcfg {{ upstream={ fifo=1 } }}
ngctl msg e0_n0-n1: setcfg {{ upstream={ droptail=1 } }}
ngctl msg e0_n0-n1: setcfg {{ upstream={ queuelen=50 } }}
```

Other FreeBSD commands that may be of interest:

- **kldstat**, **kldload**, **kldunload** - list, load, and unload FreeBSD kernel modules
- **sysctl** - display and modify various pieces of kernel state
- **pkg_info**, **pkg_add**, **pkg_delete** - list, add, or remove FreeBSD software packages.
- **vtysh** - start a Quagga CLI for router configuration

8.5.2 Netgraph Nodes

Each Netgraph node implements a protocol or processes data in some well-defined manner (see the `netgraph(4)` man page). The netgraph source code is located in `/usr/src/sys/netgraph`. There you might discover additional nodes that implement some desired functionality, that have not yet been included in CORE. Using certain kernel commands, you can likely include these types of nodes into your CORE emulation.

The following Netgraph nodes are used by CORE:

- **ng_bridge** - switch node performs Ethernet bridging
- **ng_cisco** - Cisco HDLC serial links
- **ng_eiface** - virtual Ethernet interface that is assigned to each virtual machine
- **ng_ether** - physical Ethernet devices, used by the RJ45 tool
- **ng_hub** - hub node
- **ng_pipe** - used for wired Ethernet links, imposes packet delay, bandwidth restrictions, and other link characteristics
- **ng_socket** - socket used by `ngctl` and `span` utilities
- **ng_wlan** - wireless LAN node

9 Acknowledgments

The CORE project was derived from the open source IMUNES project from the University of Zagreb in 2004. In 2006, changes for CORE were released back to that project, some items of which were adopted. Marko Zec <zec@fer.hr> is the primary developer from the University of Zagreb responsible for the IMUNES (GUI) and VirtNet (kernel) projects. Ana Kukec and Miljenko Mikuc are known contributors.

Jeff Ahrenholz <jeffrey.m.ahrenholz@boeing.com> has been the primary Boeing developer of CORE, and has written this manual. Tom Goff <thomas.goff@boeing.com> designed the Python framework and has made significant contributions. Claudiu Danilov <claudiu.b.danilov@boeing.com>, Gary Pei <guangyu.pei@boeing.com>, Phil Spagnolo, and Ian Chakeres have contributed code to CORE. Dan Mackley <daniel.c.mackley@boeing.com> helped develop the CORE API, originally to interface with a simulator. Jae Kim <jae.h.kim@boeing.com> and Tom Henderson <thomas.r.henderson@boeing.com> have supervised the project and provided direction.

Command Index

Concept Index

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z