ns-3 Tutorial

ns-3 project
feedback: ns-developers@isi.edu
4 July 2009

This is an `ns-3` tutorial. Primary documentation for the `ns-3` project is available in four forms:

- ns-3 Doxygen/Manual: Documentation of the public APIs of the simulator
- Tutorial (this document)
- Reference Manual: Reference Manual
- ns-3 wiki

This document is written in GNU Texinfo and is to be maintained in revision control on the `ns-3` code server. Both PDF and HTML versions should be available on the server. Changes to the document should be discussed on the ns-developers@isi.edu mailing list.

Table of Contents

1 Introduction

The ns-3 simulator is a discrete-event network simulator targeted primarily for research and educational use. The ns-3 project, started in 2006, is an open-source project developing ns-3.

Primary documentation for the ns-3 project is available in four forms:

- ns-3 Doxygen/Manual: Documentation of the public APIs of the simulator
- Tutorial (this document)
- Reference Manual: Reference Manual
- ns-3 wiki

The purpose of this tutorial is to introduce new ns-3 users to the system in a structured way. It is sometimes difficult for new users to glean essential information from detailed manuals and to convert this information into working simulations. In this tutorial, we will build several example simulations, introducing and explaining key concepts and features as we go.

As the tutorial unfolds, we will introduce the full ns-3 documentation and provide pointers to source code for those interested in delving deeper into the workings of the system.

A few key points are worth noting at the onset:

- Ns-3 is not an extension of ns-2; it is a new simulator. The two simulators are both written in C++ but ns-3 is a new simulator that does not support the ns-2 APIs. Some models from ns-2 have already been ported from ns-2 to ns-3. The project will continue to maintain ns-2 while ns-3 is being built, and will study transition and integration mechanisms.
- Ns-3 is open-source, and the project strives to maintain an open environment for researchers to contribute and share their software.

1.1 For ns-2 Users

For those familiar with ns-2, the most visible outward change when moving to ns-3 is the choice of scripting language. Ns-2 is scripted in OTcl and results of simulations can be visualized using the Network Animator nam. It is not possible to run a simulation in ns-2 purely from C++ (i.e., as a main() program without any OTcl). Moreover, some components of ns-2 are written in C++ and others in OTcl. In ns-3, the simulator is written entirely in C++, with optional Python bindings. Simulation scripts can therefore be written in C++ or in Python. The results of some simulations can be visualized by nam, but new animators are under development. Since ns-3 generates pcap packet trace files, other utilities can be used to analyze traces as well. In this tutorial, we will first concentrate on scripting directly in C++ and interpreting results via trace files.

But there are similarities as well (both, for example, are based on C++ objects, and some code from ns-2 has already been ported to ns-3). We will try to highlight differences between ns-2 and ns-3 as we proceed in this tutorial.

A question that we often hear is "Should I still use ns-2 or move to ns-3?" The answer is that it depends. ns-3 does not have all of the models that ns-2 currently has, but on the

other hand, ns-3 does have new capabilities (such as handling multiple interfaces on nodes correctly, use of IP addressing and more alignment with Internet protocols and designs, more detailed 802.11 models, etc.). ns-2 models can usually be ported to ns-3 (a porting guide is under development). There is active development on multiple fronts for ns-3. The ns-3 developers believe (and certain early users have proven) that ns-3 is ready for active use, and should be an attractive alternative for users looking to start new simulation projects.

1.2 Contributing

Ns-3 is a research and educational simulator, by and for the research community. It will rely on the ongoing contributions of the community to develop new models, debug or maintain existing ones, and share results. There are a few policies that we hope will encourage people to contribute to ns-3 like they have for ns-2:

- Open source licensing based on GNU GPLv2 compatibility;
- wiki;
- Contributed Code page, similar to ns-2's popular Contributed Code page;
- src/contrib directory (we will host your contributed code);
- Open bug tracker;
- Ns-3 developers will gladly help potential contributors to get started with the simulator (please contact one of us).

We realize that if you are reading this document, contributing back to the project is probably not your foremost concern at this point, but we want you to be aware that contributing is in the spirit of the project and that even the act of dropping us a note about your early experience with ns-3 (e.g. "this tutorial section was not clear..."), reports of stale documentation, etc. are much appreciated.

1.3 Tutorial Organization

The tutorial assumes that new users might initially follow a path such as the following:

- Try to download and build a copy;
- Try to run a few sample programs;
- Look at simulation output, and try to adjust it.

As a result, we have tried to organize the tutorial along the above broad sequences of events.

2 Resources

2.1 The Web

There are several important resources of which any ns-3 user must be aware. The main web site is located at http://www.nsnam.org and provides access to basic information about the ns-3 system. Detailed documentation is available through the main web site at http://www.nsnam.org/documents.html. You can also find documents relating to the system architecture from this page.

There is a Wiki that complements the main ns-3 web site which you will find at http://www.nsnam.org/wiki/. You will find user and developer FAQs there, as well as troubleshooting guides, third-party contributed code, papers, etc.

The source code may be found and browsed at http://code.nsnam.org/. There you will find the current development tree in the repository named ns-3-dev. Past releases and experimental repositories of the core developers may also be found there.

2.2 Mercurial

Complex software systems need some way to manage the organization and changes to the underlying code and documentation. There are many ways to perform this feat, and you may have heard of some of the systems that are currently used to do this. The Concurrent Version System (CVS) is probably the most well known.

The ns-3 project uses Mercurial as its source code management system. Although you do not need to know much about Mercurial in order to complete this tutorial, we recommend becoming familiar with Mercurial and using it to access the source code. Mercurial has a web site at http://www.selenic.com/mercurial/, from which you can get binary or source releases of this Software Configuration Management (SCM) system. Selenic (the developer of Mercurial) also provides a tutorial at http://www.selenic.com/mercurial/wiki/index.cgi/Tutorial/, and a QuickStart guide at http://www.selenic.com/mercurial/wiki/index.cgi/QuickStart/.

You can also find vital information about using Mercurial and ns-3 on the main ns-3 web site.

2.3 Waf

Once you have source code downloaded to your local system, you will need to compile that source to produce usable programs. Just as in the case of source code management, there are many tools available to perform this function. Probably the most well known of these tools is make. Along with being the most well known, make is probably the most difficult to use in a very large and highly configurable system. Because of this, many alternatives have been developed. Recently these systems have been developed using the Python language.

The build system Waf is used on the ns-3 project. It is one of the new generation of Python-based build systems. You will not need to understand any Python to build the existing ns-3 system, and will only have to understand a tiny and intuitively obvious subset of Python in order to extend the system in most cases.

For those interested in the gory details of Waf, the main web site can be found at http://code.google.com/p/waf/.

2.4 Development Environment

As mentioned above, scripting in ns-3 is done in C++ or Python. As of ns-3.2, most of the ns-3 API is available in Python, but the models are written in C++ in either case. A working knowledge of C++ and object-oriented concepts is assumed in this document. We will take some time to review some of the more advanced concepts or possibly unfamiliar language features, idioms and design patterns as they appear. We don't want this tutorial to devolve into a C++ tutorial, though, so we do expect a basic command of the language. There are an almost unimaginable number of sources of information on C++ available on the web or in print.

If you are new to C++, you may want to find a tutorial- or cookbook-based book or web site and work through at least the basic features of the language before proceeding. For instance, this tutorial.

The ns-3 system uses several components of the GNU "toolchain" for development. A software toolchain is the set of programming tools available in the given environment. For a quick review of what is included in the GNU toolchain see, http://en.wikipedia.org/wiki/GNU_toolchain. ns-3 uses gcc, GNU binutils, and gdb. However, we do not use the GNU build system tools, neither make nor autotools. We use Waf for these functions.

Typically an ns-3 author will work in Linux or a Linux-like environment. For those running under Windows, there do exist environments which simulate the Linux environment to various degrees. The ns-3 project supports development in the Cygwin environment for these users. See http://www.cygwin.com/ for details on downloading (MinGW is presently not officially supported, although some of the project maintainers to work with it). Cygwin provides many of the popular Linux system commands. It can, however, sometimes be problematic due to the way it actually does its emulation, and sometimes interactions with other Windows software can cause problems.

If you do use Cygwin or MinGW; and use Logitech products, we will save you quite a bit of heartburn right off the bat and encourage you to take a look at the MinGW FAQ.

Search for "Logitech" and read the FAQ entry, "why does make often crash creating a sh.exe.stackdump file when I try to compile my source code." Believe it or not, the Logitech Process Monitor insinuates itself into every DLL in the system when it is running. It can cause your Cygwin or MinGW DLLs to die in mysterious ways and often prevents debuggers from running. Beware of Logitech software when using Cygwin.

Another alternative to Cygwin is to install a virtual machine environment such as VMware server and install a Linux virtual machine.

2.5 Socket Programming

We will assume a basic facility with the Berkeley Sockets API in the examples used in this tutorial. If you are new to sockets, we recommend reviewing the API and some common usage cases. For a good overview of programming TCP/IP sockets we recommend TCP/IP Sockets in C.

There is an associated web site that includes source for the examples in the book, which you can find at: http://cs.baylor.edu/~donahoo/practical/CSockets/.

If you understand the first four chapters of the book (or for those who do not have access to a copy of the book, the echo clients and servers shown in the website above) you will be in good shape to understand the tutorial. There is a similar book on Multicast Sockets, Multicast Sockets. that covers material you may need to understand if you look at the multicast examples in the distribution.

3 Getting Started

3.1 Downloading ns-3

From this point forward, we are going to assume that the reader is working in Linux or a Linux emulation environment (Linux, Cygwin, etc.) and has the GNU toolchain installed and verified. We are also going to assume that you have Mercurial and Waf installed and running on the target system as described in the Getting Started section of the ns-3 web site: http://www.nsnam.org/getting_started.html.

The ns-3 code is available in Mercurial repositories on the server http://code.nsnam.org. You can also download a tarball release at http://www.nsnam.org/releases/, or you can work with repositories using Mercurial. We recommend using Mercurial unless there's a good reason not to. See the end of this section for instructions on how to get a tarball release.

The simplest way to get started using Mercurial repositories is to use the ns-3-allinone environment. This is a set of scripts that manages the downloading and building of various subsystems of ns-3 for you. We recommend that you begin your ns-3 adventures in this environment as it can really simplify your life at this point.

3.1.1 Downloading ns-3 Using Mercurial

One practice is to create a directory called repos in one's home directory under which one can keep local Mercurial repositories. *Hint: we will assume you do this later in the tutorial.* If you adopt that approach, you can get a copy of ns-3-allinone by typing the following into your Linux shell (assuming you have installed Mercurial):

```
cd
mkdir repos
cd repos
hg clone http://code.nsnam.org/ns-3-allinone
```

As the hg (Mercurial) command executes, you should see something like the following displayed,

```
destination directory: ns-3-allinone
requesting all changes
adding changesets
adding manifests
adding file changes
added 31 changesets with 45 changes to 7 files
7 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

After the clone command completes, you should have a directory called ns-3-allinone under your ~/repos directory, the contents of which should look something like the following:

```
build.py*  constants.py  dist.py*  download.py*  README  util.py
```

Notice that you really just downloaded some Python scripts. The next step will be to use those scripts to download and build the ns-3 distribution of your choice.

If you go to the following link: http://code.nsnam.org/, you will see a number of repositories. Many are the private repositories of the ns-3 development team. The repositories

of interest to you will be prefixed with "ns-3". Official releases of `ns-3` will be numbered as `ns-3.<release>.<hotfix>`. For example, a second hotfix to a still hypothetical release nine of `ns-3` would be numbered as `ns-3.9.2`.

We have had a regression testing framework in place since the first release. For each release, a set of output files that define "good behavior" are saved. These known good output files are called reference traces and are associated with a given release by name. For example, in http://code.nsnam.org/ you will find a repository named `ns-3.1` which is the first stable release of `ns-3`. You will also find a separate repository named `ns-3.1-ref-traces` that holds the reference traces for the `ns-3.1` release. It is crucial to keep these files consistent if you want to do any regression testing of your repository. This is a good idea to do at least once to verify everything has built correctly.

The current development snapshot (unreleased) of `ns-3` may be found at http://code.nsnam.org/ns-3-dev/ and the associated reference traces may be found at http://code.nsnam.org/ns-3-dev-ref-traces/. The developers attempt to keep these repository in consistent, working states but they are in a development area with unreleased code present, so you may want to consider staying with an official release if you do not need newly- introduced features.

Since the release numbers are going to be changing, I will stick with the more constant ns-3-dev here in the tutorial, but you can replace the string "ns-3-dev" with your choice of release (e.g., ns-3.5 and ns-3.5-ref-traces) in the text below. You can find the latest version of the code either by inspection of the repository list or by going to the "Getting Started" web page and looking for the latest release identifier.

Go ahead and change into the `ns-3-allinone` directory you created when you cloned that repository. We are now going to use the `download.py` script to pull down the various pieces of `ns-3` you will be using.

Go ahead and type the following into your shell (remember you can substitute the name of your chosen release number instead of `ns-3-dev` – like `"ns-3.5"` and `"ns-3.5-ref-traces"` if you want to work with a stable release).

```
./download.py -n ns-3-dev -r ns-3-dev-ref-traces
```

Note that the default for the `-n` option is `ns-3-dev` and the default for the `-r` option is `ns-3-dev-ref-traces` and so the above is actually redundant. We provide this example to illustrate how to specify alternate repositories. In order to download `ns-3-dev` you can actually use the defaults and simply type,

```
./download.py
```

As the hg (Mercurial) command executes, you should see something like the following,

```
    #
    # Get NS-3
    #

Cloning ns-3 branch
 =>  hg clone http://code.nsnam.org/ns-3-dev ns-3-dev
requesting all changes
adding changesets
adding manifests
adding file changes
```

```
added 4634 changesets with 16500 changes to 1762 files
870 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

This is output by the download script as it fetches the actual `ns-3` code from the repository. Next, you should see something like,

```
    #
    # Get the regression traces
    #

Synchronizing reference traces using Mercurial.
 =>  hg clone http://code.nsnam.org/ns-3-dev-ref-traces ns-3-dev-ref-traces
requesting all changes
adding changesets
adding manifests
adding file changes
added 86 changesets with 1178 changes to 259 files
208 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

This is the download script fetching the reference trace files for you. The download script is smart enough to know that on some platforms various pieces of ns-3 are not supported. On your platform you may not see some of these pieces come down. However, on most platforms, the process should continue with something like,

```
    #
    # Get PyBindGen
    #

Required pybindgen version:  0.10.0.640
Trying to fetch pybindgen; this will fail if no network connection is available.  Hit Ctr
 =>  bzr checkout -rrevno:640 https://launchpad.net/pybindgen pybindgen
Fetch was successful.
```

This was the download script getting the Python bindings generator for you. Next you should see (modulo platform variations) something along the lines of,

```
    #
    # Get NSC
    #

Required NSC version:  nsc-0.5.0
Retrieving nsc from https://secure.wand.net.nz/mercurial/nsc
 =>  hg clone https://secure.wand.net.nz/mercurial/nsc nsc
requesting all changes
adding changesets
adding manifests
adding file changes
added 273 changesets with 17565 changes to 15175 files
10622 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

This part of the process is the script downloading the Network Simulation Cradle for you.

After the clone command completes, you should have several new directories under ~/repos/ns-3-allinone:

```
build.py*     constants.pyc  download.py*  ns-3-dev-ref-traces/  pybindgen/  util.py
constants.py  dist.py*       ns-3-dev/     nsc/                  README      util.pyc
```

Go ahead and change into ns-3-dev under your ~/repos/ns-3-allinone directory. You should see something like the following there:

```
AUTHORS       examples/  regression/    scratch/  waf*
bindings/     LICENSE    regression.py  src/      waf.bat*
CHANGES.html  ns3/       RELEASE_NOTES  utils/    wscript
doc/          README     samples/       VERSION   wutils.py
```

You are now ready to build the ns-3 distribution.

3.1.2 Downloading ns-3 Using a Tarball

The process for downloading ns-3 via tarball is simpler than the Mercurial process since all of the pieces are pre-packaged for you. You just have to pick a release, download it and decompress it.

As mentioned above, one practice is to create a directory called repos in one's home directory under which one can keep local Mercurial repositories. One could also keep a tarballs directory. *Hint: the tutorial will assume you downloaded into a* repos *directory, so remember the placekeeper.* If you adopt the tarballs directory approach, you can get a copy of a release by typing the following into your Linux shell (substitute the appropriate version numbers, of course):

```
cd
mkdir tarballs
cd tarballs
wget http://www.nsnam.org/releases/ns-allinone-3.5.tar.bz2
tar xjf ns-allinone-3.5.tar.bz2
```

If you change into the directory ns-allinone-3.5 you should see a number of files:

```
build.py*     ns-3.5/             nsc-0.5.0/             README
constants.py  ns-3.5-ref-traces/  pybindgen-0.10.0.640/  util.py
```

You are now ready to build the ns-3 distribution.

3.2 Building ns-3

3.2.1 Building with build.py

The first time you build the ns-3 project you should build using the allinone environment. This will get the project configured for you in the most commonly useful way.

Change into the directory you created in the download section above. If you downloaded using Mercurial you should have a directory called ns-3-allinone under your ~/repos directory. If you downloaded using a tarball you should have a directory called something like ns-allinone-3.5 under your ~/tarballs directory. Take a deep breath and type the following:

```
./build.py
```

You will see lots of typical compiler output messages displayed as the build script builds the various pieces you downloaded. Eventually you should see the following magic words:

```
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (2m30.586s)
```

Once the project has built you can say goodbye to your old friends, the `ns-3-allinone` scripts. You got what you needed from them and will now interact directly with Waf and we do it in the `ns-3-dev` directory, not in the `ns-3-allinone` directory. Go ahead and change into the `ns-3-dev` directory (or the directory for the appropriate release you downloaded.

```
cd ns-3-dev
```

3.2.2 Building with Waf

We use Waf to configure and build the `ns-3` project. It's not strictly required at this point, but it will be valuable to take a slight detour and look at how to make changes to the configuration of the project. Probably the most useful configuration change you can make will be to build the optimized version of the code. By default you have configured your project to build the debug version. Let's tell the project to do make an optimized build. To explain to Waf that it should do optimized builds you will need to execute the following command,

```
./waf -d optimized configure
```

This runs Waf out of the local directory (which is provided as a convenience for you). As the build system checks for various dependencies you should see output that looks similar to the following,

```
Checking for program g++                 : ok /usr/bin/g++
Checking for program cpp                 : ok /usr/bin/cpp
Checking for program ar                  : ok /usr/bin/ar
Checking for program ranlib              : ok /usr/bin/ranlib
Checking for g++                         : ok
Checking for program pkg-config          : ok /usr/bin/pkg-config
Checking for regression reference traces  : ok ../ns-3-dev-ref-traces (guessed)
Checking for -Wno-error=deprecated-declarations support : yes
Checking for -Wl,--soname=foo support                   : yes
Checking for header stdlib.h                            : ok
Checking for header signal.h                            : ok
Checking for header pthread.h                           : ok
Checking for high precision time implementation         : 128-bit integer
Checking for header stdint.h                            : ok
Checking for header inttypes.h                          : ok
Checking for header sys/inttypes.h                      : not found
Checking for library rt                                 : ok
Checking for header netpacket/packet.h                  : ok
Checking for pkg-config flags for GSL                   : ok
Checking for header linux/if_tun.h                      : ok
Checking for pkg-config flags for GTK_CONFIG_STORE      : ok
Checking for pkg-config flags for LIBXML2               : ok
Checking for library sqlite3                            : ok
```

```
Checking for NSC location                                     : ok ../nsc (guessed)
Checking for library dl                                       : ok
Checking for NSC supported architecture x86_64                : ok
Checking for program python                                   : ok /usr/bin/python
Checking for Python version >= 2.3                            : ok 2.5.2
Checking for library python2.5                                : ok
Checking for program python2.5-config                         : ok /usr/bin/python2.5-config
Checking for header Python.h                                  : ok
Checking for -fvisibility=hidden support                      : yes
Checking for pybindgen location                               : ok ../pybindgen (guessed)
Checking for Python module pybindgen                          : ok
Checking for pybindgen version                                : ok 0.10.0.640
Checking for Python module pygccxml                           : ok
Checking for pygccxml version                                 : ok 0.9.5
Checking for program gccxml                                   : ok /usr/local/bin/gccxml
Checking for gccxml version                                   : ok 0.9.0
Checking for program sudo                                     : ok /usr/bin/sudo
Checking for program hg                                       : ok /usr/bin/hg
Checking for program valgrind                                 : ok /usr/bin/valgrind
---- Summary of optional NS-3 features:
Threading Primitives          : enabled
Real Time Simulator           : enabled
Emulated Net Device           : enabled
GNU Scientific Library (GSL)  : enabled
Tap Bridge                    : enabled
GtkConfigStore                : enabled
XmlIo                         : enabled
SQlite stats data output      : enabled
Network Simulation Cradle     : enabled
Python Bindings               : enabled
Python API Scanning Support   : enabled
Use sudo to set suid bit      : not enabled (option --enable-sudo not selected)
Static build                  : not enabled (option --enable-static not selected)
'configure' finished successfully (2.870s)
```

Note the last part of the above output. Some ns-3 options are not enabled by default or require support from the underlying system to work properly. For instance, to enable XmlTo, the library libxml-2.0 must be found on the system. If this library were not found, the corresponding `ns-3` feature would not be enabled and a message would be displayed. Note further that there is a feature to use the progarm `sudo` to set the suid bit of certain programs. This is not enabled by default and so this feature is reported as "not enabled."

Now go ahead and switch back to the debug build.

```
./waf -d debug configure
```

The build system is now configured and you can build the debug versions of the `ns-3` programs by simply typing,

```
./waf
```

Some waf commands are meaningful during the build phase and some commands are valid in the configuration phase. For example, if you wanted to use the emulation features of `ns-3` you might want to enable setting the suid bit using sudo as described above. This turns out to be a configuration-time command, and so you could reconfigure using the following command

```
./waf -d debug --enable-sudo configure
```

If you do this, waf will have run sudo to change the socket creator programs of the emulation code to run as root. There are many other configure- and build-time options available in waf. To explore these options, type:

```
./waf --help
```

We'll use some of the testing-related commands in the next section.

Okay, sorry, I made you build the `ns-3` part of the system twice, but now you know how to change the configuration and build optimized code.

3.3 Testing ns-3

You can run the unit tests of the `ns-3` distribution by running the "–check" option,

```
./waf --check
```

These tests are run in parallel by waf, so the summary, "Ran n tests" will appear as soon as all of the tasks are launched, but you should eventually see a report saying that,

```
C++ UNIT TESTS: all 33 tests passed.
```

This is the important message.

You will also see output from the test runner and waf task sequence numbers the output will actually look something like,

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build'
[707/709] get-unit-tests-list
[708/709] run-python-unit-tests
[709/709] print-introspected-doxygen
[710/743] run-unit-test(AddressHelper)
[711/743] run-unit-test(Wifi)
...........
----------------------------------------------------------------------
Ran 11 tests in 0.003s

OK
[712/743] run-unit-test(DcfManager)
[713/743] run-unit-test(MacRxMiddle)
[714/743] run-unit-test(Ipv4ListRouting)

...

[739/743] run-unit-test(RandomVariable)
[740/743] run-unit-test(Object)
[741/743] run-unit-test(Ptr)
[742/743] run-unit-test(Callback)
```

```
[743/743] collect-unit-tests-results
C++ UNIT TESTS: all 33 tests passed.
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (1.799s)
```

This command is typically run by `users` to quickly verify that an `ns-3` distribution has built correctly.

You can also run our regression test suite to ensure that your distribution and toolchain have produced binaries that generate output that is identical to known-good reference output files. You downloaded these reference traces to your machine during the `./download.py` process above. (Warning: The `ns-3.2` and `ns-3.3` releases do not use the `ns-3-allinone` environment and require you to be online when you run regression tests because hey dynamically synchronize the reference traces directory with an online repository immediately prior to the run).

During regression testing Waf will run a number of tests that generate what we call trace files. The content of these trace files are compared with the reference traces. If they are identical, the regression tests report a PASS status. If a regression test fails you will see a FAIL indication along with a pointer to the offending trace file and its associated reference trace file along with a suggestion on diff parameters and options in order to see what has gone awry. If the error was discovered in a pcap file, it will be useful to convert the pcap files to text using tcpdump prior to comparison.

Some regression tests may be SKIPped if the required support is not present.

Note that the regression tests are also run in parallel and so the messages may be interleaved.

To run the regression tests, you provide Waf with the regression flag.

```
./waf --regression
```

You should see messages indicating that many tests are being run and are passing.

```
Entering directory ‘repos/ns-3-allinone/ns-3-dev/build’
[647/669] regression-test (test-csma-bridge)
[648/669] regression-test (test-csma-broadcast)
[649/669] regression-test (test-csma-multicast)
[650/669] regression-test (test-csma-one-subnet)
PASS test-csma-multicast
[651/669] regression-test (test-csma-packet-socket)
PASS test-csma-bridge
...
Regression testing summary:
PASS: 22 of 22 tests passed
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (25.826s)
```

If you want to take a look at an example of what might be checked during a regression test, you can do the following:

```
cd build/debug/regression/traces/second.ref
tcpdump -nn -tt -r second-2-0.pcap
```

The output should be clear to anyone who is familiar with tcpdump or net sniffers. We’ll have much more to say on pcap files later in this tutorial.

Remember to cd back into the top-level `ns-3` directory after you are done:

```
cd ../../../../..
```

3.4 Running a Script

We typically run scripts under the control of Waf. This allows the build system to ensure that the shared library paths are set correctly and that the libraries are available at run time. To run a program, simply use the `--run` option in Waf. Let's run the `ns-3` equivalent of the ubiquitous hello world program by typing the following:

```
./waf --run hello-simulator
```

Waf first checks to make sure that the program is built correctly and executes a build if required. Waf then executes the program, which produces the following output.

```
Hello Simulator
```

Congratulations. You are now an ns-3 user.

If you want to run programs under another tool such as gdb or valgrind, see this wiki entry.

4 Conceptual Overview

The first thing we need to do before actually starting to look at or write ns-3 code is to explain a few core concepts and abstractions in the system. Much of this may appear transparently obvious to some, but we recommend taking the time to read through this section just to ensure you are starting on a firm foundation.

4.1 Key Abstractions

In this section, we'll review some terms that are commonly used in networking, but have a specific meaning in ns-3.

4.1.1 Node

In Internet jargon, a computing device that connects to a network is called a *host* or sometimes an *end system*. Because ns-3 is a *network* simulator, not specifically an *Internet* simulator, we intentionally do not use the term host since it is closely associated with the Internet and its protocols. Instead, we use a more generic term also used by other simulators that originates in Graph Theory — the *node*.

In ns-3 the basic computing device abstraction is called the node. This abstraction is represented in C++ by the class Node. The Node class provides methods for managing the representations of computing devices in simulations.

You should think of a Node as a computer to which you will add functionality. One adds things like applications, protocol stacks and peripheral cards with their associated drivers to enable the computer to do useful work. We use the same basic model in ns-3.

4.1.2 Application

Typically, computer software is divided into two broad classes. *System Software* organizes various computer resources such as memory, processor cycles, disk, network, etc., according to some computing model. System software usually does not use those resources to complete tasks that directly benefit a user. A user would typically run an *application* that acquires and uses the resources controlled by the system software to accomplish some goal.

Often, the line of separation between system and application software is made at the privilege level change that happens in operating system traps. In ns-3 there is no real concept of operating system and especially no concept of privilege levels or system calls. We do, however, have the idea of an application. Just as software applications run on computers to perform tasks in the "real world," ns-3 applications run on ns-3 Nodes to drive simulations in the simulated world.

In ns-3 the basic abstraction for a user program that generates some activity to be simulated is the application. This abstraction is represented in C++ by the class Application. The Application class provides methods for managing the representations of our version of user-level applications in simulations. Developers are expected to specialize the Application class in the object-oriented programming sense to create new applications. In this tutorial, we will use specializations of class Application called UdpEchoClientApplication and UdpEchoServerApplication. As you might expect, these applications compose a client/server application set used to generate and echo simulated network packets

4.1.3 Channel

In the real world, one can connect a computer to a network. Often the media over which data flows in these networks are called *channels*. When you connect your Ethernet cable to the plug in the wall, you are connecting your computer to an Ethernet communication channel. In the simulated world of `ns-3`, one connects a `Node` to an object representing a communication channel. Here the basic communication subnetwork abstraction is called the channel and is represented in C++ by the class `Channel`.

The `Channel` class provides methods for managing communication subnetwork objects and connecting nodes to them. `Channels` may also be specialized by developers in the object oriented programming sense. A `Channel` specialization may model something as simple as a wire. The specialized `Channel` can also model things as complicated as a large Ethernet switch, or three-dimensional space full of obstructions in the case of wireless networks.

We will use specialized versions of the `Channel` called `CsmaChannel`, `PointToPointChannel` and `WifiChannel` in this tutorial. The `CsmaChannel`, for example, models a version of a communication subnetwork that implements a *carrier sense multiple access* communication medium. This gives us Ethernet-like functionality.

4.1.4 Net Device

It used to be the case that if you wanted to connect a computers to a network, you had to buy a specific kind of network cable and a hardware device called (in PC terminology) a *peripheral card* that needed to be installed in your computer. If the peripheral card implemented some networking function, they were called Network Interface Cards, or *NICs*. Today most computers come with the network interface hardware built in and users don't see these building blocks.

A NIC will not work without a software driver to control the hardware. In Unix (or Linux), a piece of peripheral hardware is classified as a *device*. Devices are controlled using *device drivers*, and network devices (NICs) are controlled using *network device drivers* collectively known as *net devices*. In Unix and Linux you refer to these net devices by names such as *eth0*.

In `ns-3` the *net device* abstraction covers both the software driver and the simulated hardware. A net device is "installed" in a `Node` in order to enable the `Node` to communicate with other `Nodes` in the simulation via `Channels`. Just as in a real computer, a `Node` may be connected to more than one `Channel` via multiple `NetDevices`.

The net device abstraction is represented in C++ by the class `NetDevice`. The `NetDevice` class provides methods for managing connections to `Node` and `Channel` objects; and may be specialized by developers in the object-oriented programming sense. We will use the several specialized versions of the `NetDevice` called `CsmaNetDevice`, `PointToPointNetDevice`, and `WifiNetDevice` in this tutorial. Just as an Ethernet NIC is designed to work with an Ethernet network, the `CsmaNetDevice` is designed to work with a `CsmaChannel`; the `PointToPointNetDevice` is designed to work with a `PointToPointChannel` and a `WifiNetNevice` is designed to work with a `WifiChannel`.

4.1.5 Topology Helpers

In a real network, you will find host computers with added (or built-in) NICs. In ns-3 we would say that you will find Nodes with attached NetDevices. In a large simulated network you will need to arrange many connections between Nodes, NetDevices and Channels.

Since connecting NetDevices to Nodes, NetDevices to Channels, assigning IP addresses, etc., are such common tasks in ns-3, we provide what we call *topology helpers* to make this as easy as possible. For example, it may take many distinct ns-3 core operations to create a NetDevice, add a MAC address, install that net device on a Node, configure the node's protocol stack, and then connect the NetDevice to a Channel. Even more operations would be required to connect multiple devices onto multipoint channels and then to connect individual networks together into internetworks. We provide topology helper objects that combine those many distinct operations into an easy to use model for your convenience.

4.2 A First ns-3 Script

If you downloaded the system as was suggested above, you will have a release of ns-3 in a directory called repos under your home directory. Change into that release directory, and you should find a directory structure something like the following:

```
AUTHORS       doc/       README          RELEASE_NOTES  utils/    wscript
bindings/     examples/  regression/     samples/       VERSION   wutils.py
build/        LICENSE    regression.py   scratch/       waf*      wutils.pyc
CHANGES.html  ns3/       regression.pyc  src/           waf.bat*
```

Change into the examples directory. You should see a file named first.cc located there. This is a script that will create a simple point-to-point link between two nodes and echo a single packet between the nodes. Let's take a look at that script line by line, so go ahead and open first.cc in your favorite editor.

4.2.1 Boilerplate

The first line in the file is an emacs mode line. This tells emacs about the formatting conventions (coding style) we use in our source code.

```
/* -*- Mode:C++; c-file-style:''gnu''; indent-tabs-mode:nil; -*- */
```

This is always a somewhat controversial subject, so we might as well get it out of the way immediately. The ns-3 project, like most large projects, has adopted a coding style to which all contributed code must adhere. If you want to contribute your code to the project, you will eventually have to conform to the ns-3 coding standard as described in the file doc/codingstd.txt or shown on the project web page here.

We recommend that you, well, just get used to the look and feel of ns-3 code and adopt this standard whenever you are working with our code. All of the development team and contributors have done so with various amounts of grumbling. The emacs mode line above makes it easier to get the formatting correct if you use the emacs editor.

The ns-3 simulator is licensed using the GNU General Public License. You will see the appropriate GNU legalese at the head of every file in the ns-3 distribution. Often you will see a copyright notice for one of the institutions involved in the ns-3 project above the GPL text and an author listed below.

```
/*
```

```
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */
```

4.2.2 Module Includes

The code proper starts with a number of include statements.

```
#include "ns3/core-module.h"
#include "ns3/simulator-module.h"
#include "ns3/node-module.h"
#include "ns3/helper-module.h"
```

To help our high-level script users deal with the large number of include files present in the system, we group includes according to relatively large modules. We provide a single include file that will recursively load all of the include files used in each module. Rather than having to look up exactly what header you need, and possibly have to get a number of dependencies right, we give you the ability to load a group of files at a large granularity. This is not the most efficient approach but it certainly makes writing scripts much easier.

Each of the `ns-3` include files is placed in a directory called `ns3` (under the build directory) during the build process to help avoid include file name collisions. The `ns3/core-module.h` file corresponds to the ns-3 module you will find in the directory `src/core` in your downloaded release distribution. If you list this directory you will find a large number of header files. When you do a build, Waf will place public header files in an `ns3` directory under the appropriate `build/debug` or `build/optimized` directory depending on your configuration. Waf will also automatically generate a module include file to load all of the public header files.

Since you are, of course, following this tutorial religiously, you will already have done a

```
./waf -d debug configure
```

in order to configure the project to perform debug builds. You will also have done a

```
./waf
```

to build the project. So now if you look in the directory `../build/debug/ns3` you will find the four module include files shown above. You can take a look at the contents of these files and find that they do include all of the public include files in their respective modules.

4.2.3 Ns3 Namespace

The next line in the `first.cc` script is a namespace declaration.

```
using namespace ns3;
```

The ns-3 project is implemented in a C++ namespace called ns3. This groups all ns-3-related declarations in a scope outside the global namespace, which we hope will help with integration with other code. The C++ using statement introduces the ns-3 namespace into the current (global) declarative region. This is a fancy way of saying that after this declaration, you will not have to type ns3:: scope resolution operator before all of the ns-3 code in order to use it. If you are unfamiliar with namespaces, please consult almost any C++ tutorial and compare the ns3 namespace and usage here with instances of the std namespace and the using namespace std; statements you will often find in discussions of cout and streams.

4.2.4 Logging

The next line of the script is the following,

```
NS_LOG_COMPONENT_DEFINE ("FirstScriptExample");
```

We will use this statement as a convenient place to talk about our Doxygen documentation system. If you look at the project web site, ns-3 project, you will find a link to "Doxygen (ns-3-dev)" in the navigation bar. If you select this link, you will be taken to our documentation page for the current development release. There is also a link to "Doxygen (stable)" that will take you to the documentation for the latest stable release of ns-3.

Along the left side, you will find a graphical representation of the structure of the documentation. A good place to start is the NS-3 Modules "book" in the ns-3 navigation tree. If you expand Modules you will see a list of ns-3 module documentation. The concept of module here ties directly into the module include files discussed above. It turns out that the ns-3 logging subsystem is part of the core module, so go ahead and expand that documentation node. Now, expand the Debugging book and then select the Logging page.

You should now be looking at the Doxygen documentation for the Logging module. In the list of #defines at the top of the page you will see the entry for NS_LOG_COMPONENT_DEFINE. Before jumping in, it would probably be good to look for the "Detailed Description" of the logging module to get a feel for the overall operation. You can either scroll down or select the "More..." link under the collaboration diagram to do this.

Once you have a general idea of what is going on, go ahead and take a look at the specific NS_LOG_COMPONENT_DEFINE documentation. I won't duplicate the documentation here, but to summarize, this line declares a logging component called FirstScriptExample that allows you to enable and disable console message logging by reference to the name.

4.2.5 Main Function

The next lines of the script you will find are,

```
  int
main (int argc, char *argv[])
{
```

This is just the declaration of the main function of your program (script). Just as in any C++ program, you need to define a main function that will be the first function run. There is nothing at all special here. Your ns-3 script is just a C++ program.

The next two lines of the script are used to enable two logging components that are built into the Echo Client and Echo Server applications:

```
LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
LogComponentEnable("UdpEchoServerApplication", LOG_LEVEL_INFO);
```

If you have read over the Logging component documentation you will have seen that there are a number of levels of logging verbosity/detail that you can enable on each component. These two lines of code enable debug logging at the INFO level for echo clients and servers. This will result in the application printing out messages as packets are sent and received during the simulation.

Now we will get directly to the business of creating a topology and running a simulation. We use the topology helper objects to make this job as easy as possible.

4.2.6 Topology Helpers

4.2.6.1 NodeContainer

The next two lines of code in our script will actually create the `ns-3` `Node` objects that will represent the computers in the simulation.

```
NodeContainer nodes;
nodes.Create (2);
```

Let's find the documentation for the `NodeContainer` class before we continue. Another way to get into the documentation for a given class is via the `Classes` tab in the Doxygen pages. If you still have the Doxygen handy, just scroll up to the top of the page and select the `Classes` tab. You should see a new set of tabs appear, one of which is `Class List`. Under that tab you will see a list of all of the `ns-3` classes. Scroll down, looking for `ns3::NodeContainer`. When you find the class, go ahead and select it to go to the documentation for the class.

You may recall that one of our key abstractions is the `Node`. This represents a computer to which we are going to add things like protocol stacks, applications and peripheral cards. The `NodeContainer` topology helper provides a convenient way to create, manage and access any `Node` objects that we create in order to run a simulation. The first line above just declares a NodeContainer which we call `nodes`. The second line calls the `Create` method on the `nodes` object and asks the container to create two nodes. As described in the Doxygen, the container calls down into the `ns-3` system proper to create two `Node` objects and stores pointers to those objects internally.

The nodes as they stand in the script do nothing. The next step in constructing a topology is to connect our nodes together into a network. The simplest form of network we support is a single point-to-point link between two nodes. We'll construct one of those links here.

4.2.6.2 PointToPointHelper

We are constructing a point to point link, and, in a pattern which will become quite familiar to you, we use a topology helper object to do the low-level work required to put the link together. Recall that two of our key abstractions are the `NetDevice` and the `Channel`. In the real world, these terms correspond roughly to peripheral cards and network cables. Typically these two things are intimately tied together and one cannot expect to interchange, for example, Ethernet devices and wireless channels. Our Topology Helpers follow this intimate coupling and therefore you will use a single `PointToPointHelper` to configure and connect `ns-3` `PointToPointNetDevice` and `PointToPointChannel` objects in this script.

The next three lines in the script are,

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

The first line,

```
PointToPointHelper pointToPoint;
```

instantiates a `PointToPointHelper` object on the stack. From a high-level perspective the next line,

```
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
```

tells the `PointToPointHelper` object to use the value "5Mbps" (five megabits per second) as the "DataRate" when it creates a `PointToPointNetDevice` object.

From a more detailed perspective, the string "DataRate" corresponds to what we call an `Attribute` of the `PointToPointNetDevice`. If you look at the Doxygen for class `ns3::PointToPointNetDevice` and find the documentation for the `GetTypeId` method, you will find a list of `Attributes` defined for the device. Among these is the "DataRate" `Attribute`. Most user-visible `ns-3` objects have similar lists of `Attributes`. We use this mechanism to easily configure simulations without recompiling as you will see in a following section.

Similar to the "DataRate" on the `PointToPointNetDevice` you will find a "Delay" `Attribute` associated with the `PointToPointChannel`. The final line,

```
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

tells the `PointToPointHelper` to use the value "2ms" (two milliseconds) as the value of the transmission delay of every point to point channel it subsequently creates.

4.2.6.3 NetDeviceContainer

At this point in the script, we have a `NodeContainer` that contains two nodes. We have a `PointToPointHelper` that is primed and ready to make `PointToPointNetDevices` and wire `PointToPointChannel` objects between them. Just as we used the `NodeContainer` topology helper object to create the `Nodes` for our simulation, we will ask the `PointToPointHelper` to do the work involved in creating, configuring and installing our devices for us. We will need to have a list of all of the NetDevice objects that are created, so we use a NetDeviceContainer to hold them just as we used a NodeContainer to hold the nodes we created. The following two lines of code,

```
NetDeviceContainer devices;
devices = pointToPoint.Install (nodes);
```

will finish configuring the devices and channel. The first line declares the device container mentioned above and the second does the heavy lifting. The `Install` method of the `PointToPointHelper` takes a `NodeContainer` as a parameter. Internally, a `NetDeviceContainer` is created. For each node in the `NodeContainer` (there must be exactly two for a point-to-point link) a `PointToPointNetDevice` is created and saved in the device container. A `PointToPointChannel` is created and the two `PointToPointNetDevices` are attached. When objects are created by the `PointToPointHelper`, the `Attributes` previously set in the helper are used to initialize the corresponding `Attributes` in the created objects.

After executing the `pointToPoint.Install (nodes)` call we will have two nodes, each with an installed point-to-point net device and a single point-to-point channel between them. Both devices will be configured to transmit data at five megabits per second over the channel which has a two millisecond transmission delay.

4.2.6.4 InternetStackHelper

We now have nodes and devices configured, but we don't have any protocol stacks installed on our nodes. The next two lines of code will take care of that.

```
InternetStackHelper stack;
stack.Install (nodes);
```

The `InternetStackHelper` is a topology helper that is to internet stacks what the `PointToPointHelper` is to point-to-point net devices. The `Install` method takes a `NodeContainer` as a parameter. When it is executed, it will install an Internet Stack (TCP, UDP, IP, etc.) on each of the nodes in the node container.

4.2.6.5 Ipv4AddressHelper

Next we need to associate the devices on our nodes with IP addresses. We provide a topology helper to manage the allocation of IP addresses. The only user-visible API is to set the base IP address and network mask to use when performing the actual address allocation (which is done at a lower level inside the helper).

The next two lines of code in our example script, `first.cc`,

```
Ipv4AddressHelper address;
address.SetBase ("10.1.1.0", "255.255.255.0");
```

declare an address helper object and tell it that it should begin allocating IP addresses from the network 10.1.1.0 using the mask 255.255.255.0 to define the allocatable bits. By default the addresses allocated will start at one and increase monotonically, so the first address allocated from this base will be 10.1.1.1, followed by 10.1.1.2, etc. The low level `ns-3` system actually remembers all of the IP addresses allocated and will generate a fatal error if you accidentally cause the same address to be generated twice (which is a very hard to debug error, by the way).

The next line of code,

```
Ipv4InterfaceContainer interfaces = address.Assign (devices);
```

performs the actual address assignment. In `ns-3` we make the association between an IP address and a device using an `Ipv4Interface` object. Just as we sometimes need a list of net devices created by a helper for future reference we sometimes need a list of `Ipv4Interface` objects. The `Ipv4InterfaceContainer` provides this functionality.

Now we have a point-to-point network built, with stacks installed and IP addresses assigned. What we need at this point are applications to generate traffic.

4.2.7 Applications

Another one of the core abstractions of the ns-3 system is the `Application`. In this script we use two specializations of the core `ns-3` class `Application` called `UdpEchoServerApplication` and `UdpEchoClientApplication`. Just as we have in our previous explanations, we use helper objects to help configure and manage the underlying

objects. Here, we use UdpEchoServerHelper and UdpEchoClientHelper objects to make our lives easier.

4.2.7.1 UdpEchoServerHelper

The following lines of code in our example script, first.cc, are used to set up a UDP echo server application on one of the nodes we have previously created.

```
UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
```

The first line of code in the above snippet declares the UdpEchoServerHelper. As usual, this isn't the application itself, it is an object used to help us create the actual applications. One of our conventions is to place *required* Attributes in the helper constructor. In this case, the helper can't do anything useful unless it is provided with a port number that the client also knows about. Rather than just picking one and hoping it all works out, we require the port number as a parameter to the constructor. The constructor, in turn, simply does a SetAttribute with the passed value. If you want, you can set the "Port" Attribute to another value later using SetAttribute.

Similar to many other helper objects, the UdpEchoServerHelper object has an Install method. It is the execution of this method that actually causes the underlying echo server application to be instantiated and attached to a node. Interestingly, the Install method takes a NodeContainter as a parameter just as the other Install methods we have seen. This is actually what is passed to the method even though it doesn't look so in this case. There is a C++ *implicit conversion* at work here that takes the result of nodes.Get (1) (which returns a smart pointer to a node object — Ptr<Node>) and uses that in a constructor for an unnamed NodeContainer that is then passed to Install. If you are ever at a loss to find a particular method signature in C++ code that compiles and runs just fine, look for these kinds of implicit conversions.

We now see that echoServer.Install is going to install a UdpEchoServerApplication on the node found at index number one of the NodeContainer we used to manage our nodes. Install will return a container that holds pointers to all of the applications (one in this case since we passed a NodeContainer containing one node) created by the helper.

Applications require a time to "start" generating traffic and may take an optional time to "stop". We provide both. These times are set using the ApplicationContainer methods Start and Stop. These methods take Time parameters. In this case, we use an *explicit* C++ conversion sequence to take the C++ double 1.0 and convert it to an ns-3 Time object using a Seconds cast. Be aware that the conversion rules may be controlled by the model author, and C++ has its own rules, so you can't always just assume that parameters will be happily converted for you. The two lines,

```
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
```

will cause the echo server application to Start (enable itself) at one second into the simulation and to Stop (disable itself) at ten seconds into the simulation. By virtue of the

fact that we have declared a simulation event (the application stop event) to be executed at ten seconds, the simulation will last *at least* ten seconds.

4.2.7.2 UdpEchoClientHelper

The echo client application is set up in a method substantially similar to that for the server. There is an underlying `UdpEchoClientApplication` that is managed by an `UdpEchoClientHelper`.

```
UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer clientApps = echoClient.Install (nodes.Get (0));
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

For the echo client, however, we need to set five different `Attributes`. The first two `Attributes` are set during construction of the `UdpEchoClientHelper`. We pass parameters that are used (internally to the helper) to set the "RemoteAddress" and "RemotePort" `Attributes` in accordance with our convention to make required `Attributes` parameters in the helper constructors.

Recall that we used an `Ipv4InterfaceContainer` to keep track of the IP addresses we assigned to our devices. The zeroth interface in the `interfaces` container is going to correspond to the IP address of the zeroth node in the `nodes` container. The first interface in the `interfaces` container corresponds to the IP address of the first node in the `nodes` container. So, in the first line of code (from above), we are creating the helper and telling it so set the remote address of the client to be the IP address assigned to the node on which the server resides. We also tell it to arrange to send packets to port nine.

The "MaxPackets" `Attribute` tells the client the maximum number of packets we allow it to send during the simulation. The "Interval" `Attribute` tells the client how long to wait between packets, and the "PacketSize" `Attribute` tells the client how large its packet payloads should be. With this particular combination of `Attributes`, we are telling the client to send one 1024-byte packet.

Just as in the case of the echo server, we tell the echo client to `Start` and `Stop`, but here we start the client one second after the server is enabled (at two seconds into the simulation).

4.2.8 Simulator

What we need to do at this point is to actually run the simulation. This is done using the global function `Simulator::Run`.

```
Simulator::Run ();
```

When we previously called the methods,

```
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
...
clientApps.Start (Seconds (2.0));
```

```
clientApps.Stop (Seconds (10.0));
```

we actually scheduled events in the simulator at 1.0 seconds, 2.0 seconds and two events at 10.0 seconds. When `Simulator::Run` is called, the system will begin looking through the list of scheduled events and executing them. First it will run the event at 1.0 seconds, which will enable the echo server application (this event may, in turn, schedule many other events). Then it will run the event scheduled for t=2.0 seconds which will start the echo client application. Again, this event may schedule many more events. The start event implementation in the echo client application will begin the data transfer phase of the simulation by sending a packet to the server.

The act of sending the packet to the server will trigger a chain of events that will be automatically scheduled behind the scenes and which will perform the mechanics of the packet echo according to the various timing parameters that we have set in the script.

Eventually, since we only send one packet (recall the `MaxPackets Attribute` was set to one), the chain of events triggered by that single client echo request will taper off and the simulation will go idle. Once this happens, the remaining events will be the `Stop` events for the server and the client. When these events are executed, there are no further events to process and `Simulator::Run` returns. The simulation is then complete.

All that remains is to clean up. This is done by calling the global function `Simulator::Destroy`. As the helper functions (or low level `ns-3` code) executed, they arranged it so that hooks were inserted in the simulator to destroy all of the objects that were created. You did not have to keep track of any of these objects yourself — all you had to do was to call `Simulator::Destroy` and exit. The `ns-3` system took care of the hard part for you. The remaining lines of our first `ns-3` script, `first.cc`, do just that:

```
  Simulator::Destroy ();
  return 0;
}
```

4.2.9 Building Your Script

We have made it trivial to build your simple scripts. All you have to do is to drop your script into the scratch directory and it will automatically be built if you run Waf. Let's try it. Copy `examples/first.cc` into the `scratch` directory after changing back into the top level directory.

```
cd ..
cp examples/first.cc scratch/myfirst.cc
```

Now build your first example script using waf:

```
./waf
```

You should see messages reporting that your `myfirst` example was built successfully.

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
[614/708] cxx: scratch/myfirst.cc -> build/debug/scratch/myfirst_3.o
[706/708] cxx_link: build/debug/scratch/myfirst_3.o -> build/debug/scratch/myfirst
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (2.357s)
```

You can now run the example (note that if you build your program in the scratch directory you must run it out of the scratch directory):

```
./waf --run scratch/myfirst
```

You should see some output:

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (0.418s)
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.1.2
```

Here you see that the build system checks to make sure that the file has been build and then runs it. You see the logging component on the echo client indicate that it has sent one 1024 byte packet to the Echo Server on 10.1.1.2. You also see the logging component on the echo server say that it has received the 1024 bytes from 10.1.1.1. The echo server silently echoes the packet and you see the echo client log that it has received its packet back from the server.

4.3 Ns-3 Source Code

Now that you have used some of the `ns-3` helpers you may want to have a look at some of the source code that implements that functionality. The most recent code can be browsed on our web server at the following link: http://code.nsnam.org/ns-3-dev. There, you will see the Mercurial summary page for our `ns-3` development tree.

At the top of the page, you will see a number of links,

```
summary | shortlog | changelog | graph | tags | files
```

Go ahead and select the `files` link. This is what the top-level of most of our *repositories* will look:

```
drwxr-xr-x                                [up]
drwxr-xr-x                                bindings python  files
drwxr-xr-x                                doc              files
drwxr-xr-x                                examples         files
drwxr-xr-x                                ns3              files
drwxr-xr-x                                regression       files
drwxr-xr-x                                samples          files
drwxr-xr-x                                scratch          files
drwxr-xr-x                                src              files
drwxr-xr-x                                utils            files
-rw-r--r-- 2009-07-01 12:47 +0200 560     .hgignore        file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 1886    .hgtags          file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 1276    AUTHORS          file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 30961   CHANGES.html      file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 17987   LICENSE          file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 3742    README           file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 16171   RELEASE_NOTES    file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 6       VERSION          file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 10946   regression.py    file | revisions | annotate
-rwxr-xr-x 2009-07-01 12:47 +0200 88110   waf              file | revisions | annotate
-rwxr-xr-x 2009-07-01 12:47 +0200 28      waf.bat          file | revisions | annotate
```

```
-rw-r--r-- 2009-07-01 12:47 +0200 35395  wscript          file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 7673   wutils.py       file | revisions | annotate
```

Our example scripts are in the `examples` directory. If you click on `examples` you will see a list of files. One of the files in that directory is `first.cc`. If you click on `first.cc` you will find the code you just walked through.

The source code is mainly in the `src` directory. You can view source code either by clicking on the directory name or by clicking on the `files` link to the right of the directory name. If you click on the `src` directory, you will be taken to the listing of the `src` subdirectories. If you then click on `core` subdirectory, you will find a list of files. The first file you will find (as of this writing) is `abort.h`. If you click on the `abort.h` link, you will be sent to the source file for `abort.h` which contains useful macros for exiting scripts if abnormal conditions are detected.

The source code for the helpers we have used in this chapter can be found in the `src/helper` directory. Feel free to poke around in the directory tree to get a feel for what is there and the style of `ns-3` programs.

5 Tweaking ns-3

5.1 Using the Logging Module

We have already taken a brief look at the `ns-3` logging module while going over the `first.cc` script. We will now take a closer look and see what kind of use-cases the logging subsystem was designed to cover.

5.1.1 Logging Overview

Many large systems support some kind of message logging facility, and `ns-3` is not an exception. In some cases, only error messages are logged to the "operator console" (which is typically `stderr` in Unix- based systems). In other systems, warning messages may be output as well as more detailed informational messages. In some cases, logging facilities are used to output debug messages which can quickly turn the output into a blur.

`Ns-3` takes the view that all of these verbosity levels are useful and we provide a selectable, multi-level approach to message logging. Logging can be disabled completely, enabled on a component-by-component basis, or enabled globally; and it provides selectable verbosity levels. The `ns-3` log module provides a straightforward, relatively easy to use way to get useful information out of your simulation.

You should understand that we do provide a general purpose mechanism — tracing — to get data out of your models which should be preferred for simulation output (see the tutorial section Using the Tracing System for more details on our tracing system). Logging should be preferred for debugging information, warnings, error messages, or any time you want to easily get a quick message out of your scripts or models.

There are currently seven levels of log messages of increasing verbosity defined in the system.

- NS_LOG_ERROR — Log error messages;
- NS_LOG_WARN — Log warning messages;
- NS_LOG_DEBUG — Log relatively rare, ad-hoc debugging messages;
- NS_LOG_INFO — Log informational messages about program progress;
- NS_LOG_FUNCTION — Log a message describing each function called;
- NS_LOG_LOGIC – Log messages describing logical flow within a function;
- NS_LOG_ALL — Log everything.

We also provide an unconditional logging level that is always displayed, irrespective of logging levels or component selection.

- NS_LOG_UNCOND – Log the associated message unconditionally.

Each level can be requested singly or cumulatively; and logging can be set up using a shell environment variable (NS_LOG) or by logging system function call. As was seen earlier in the tutorial, the logging system has Doxygen documentation and now would be a good time to peruse the Logging Module documentation if you have not done so.

Now that you have read the documentation in great detail, let's use some of that knowledge to get some interesting information out of the `scratch/myfirst.cc` example script you have already built.

5.1.2 Enabling Logging

Let's use the NS_LOG environment variable to turn on some more logging, but first, just to get our bearings, go ahead and run the last script just as you did previously,

```
./waf --run scratch/myfirst
```

You should see the now familiar output of the first `ns-3` example program

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (0.413s)
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.1.2
```

It turns out that the "Sent" and "Received" messages you see above are actually logging messages from the `UdpEchoClientApplication` and `UdpEchoServerApplication`. We can ask the client application, for example, to print more information by setting its logging level via the NS_LOG environment variable.

I am going to assume from here on that you are using an sh-like shell that uses the"VARIABLE=value" syntax. If you are using a csh-like shell, then you will have to convert my examples to the "setenv VARIABLE value" syntax required by those shells.

Right now, the UDP echo client application is responding to the following line of code in `scratch/myfirst.cc`,

```
LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
```

This line of code enables the `LOG_LEVEL_INFO` level of logging. When we pass a logging level flag, we are actually enabling the given level and all lower levels. In this case, we have enabled `NS_LOG_INFO`, `NS_LOG_DEBUG`, `NS_LOG_WARN` and `NS_LOG_ERROR`. We can increase the logging level and get more information without changing the script and recompiling by setting the NS_LOG environment variable like this:

```
export NS_LOG=UdpEchoClientApplication=level_all
```

This sets the shell environment variable `NS_LOG` to the string,

```
UdpEchoClientApplication=level_all
```

The left hand side of the assignment is the name of the logging component we want to set, and the right hand side is the flag we want to use. In this case, we are going to turn on all of the debugging levels for the application. If you run the script with NS_LOG set this way, the `ns-3` logging system will pick up the change and you should see the following output:

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (0.404s)
UdpEchoClientApplication:UdpEchoClient()
UdpEchoClientApplication:SetDataSize(1024)
UdpEchoClientApplication:StartApplication()
UdpEchoClientApplication:ScheduleTransmit()
UdpEchoClientApplication:Send()
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
```

```
UdpEchoClientApplication:HandleRead(0x6241e0, 0x624a20)
Received 1024 bytes from 10.1.1.2
UdpEchoClientApplication:StopApplication()
UdpEchoClientApplication:DoDispose()
UdpEchoClientApplication:~UdpEchoClient()
```

The additional debug information provided by the application is from the NS_LOG_FUNCTION level. This shows every time a function in the application is called during script execution. Note that there are no requirements in the `ns-3` system that models must support any particular logging functionality. The decision regarding how much information is logged is left to the individual model developer. In the case of the echo applications, a good deal of log output is available.

You can now see a log of the function calls that were made to the application. If you look closely you will notice a single colon between the string `UdpEchoClientApplication` and the method name where you might have expected a C++ scope operator (`::`). This is intentional.

The name is not actually a class name, it is a logging component name. When there is a one-to-one correspondence between a source file and a class, this will generally be the class name but you should understand that it is not actually a class name, and there is a single colon there instead of a double colon to remind you in a relatively subtle way to conceptually separate the logging component name from the class name.

It turns out that in some cases, it can be hard to determine which method actually generates a log message. If you look in the text above, you may wonder where the string "`Received 1024 bytes from 10.1.1.2`" comes from. You can resolve this by OR'ing the `prefix_func` level into the `NS_LOG` environment variable. Try doing the following,

```
export 'NS_LOG=UdpEchoClientApplication=level_all|prefix_func'
```

Note that the quotes are required since the vertical bar we use to indicate an OR operation is also a Unix pipe connector.

Now, if you run the script you will see that the logging system makes sure that every message from the given log component is prefixed with the component name.

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build'
'build' finished successfully (0.417s)
UdpEchoClientApplication:UdpEchoClient()
UdpEchoClientApplication:SetDataSize(1024)
UdpEchoClientApplication:StartApplication()
UdpEchoClientApplication:ScheduleTransmit()
UdpEchoClientApplication:Send()
UdpEchoClientApplication:Send(): Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
UdpEchoClientApplication:HandleRead(0x6241e0, 0x624a20)
UdpEchoClientApplication:HandleRead(): Received 1024 bytes from 10.1.1.2
UdpEchoClientApplication:StopApplication()
UdpEchoClientApplication:DoDispose()
UdpEchoClientApplication:~UdpEchoClient()
```

You can now see all of the messages coming from the UDP echo client application are identified as such. The message "Received 1024 bytes from 10.1.1.2" is now clearly identified as coming from the echo client application. The remaining message must be coming from the UDP echo server application. We can enable that component by entering a colon separated list of components in the NS_LOG environment variable.

```
export 'NS_LOG=UdpEchoClientApplication=level_all|prefix_func:
               UdpEchoServerApplication=level_all|prefix_func'
```

Warning: You will need to remove the newline after the : in the example text above which is only there for document formatting purposes.

Now, if you run the script you will see all of the log messages from both the echo client and server applications. You may see that this can be very useful in debugging problems.

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build'
'build' finished successfully (0.406s)
UdpEchoServerApplication:UdpEchoServer()
UdpEchoClientApplication:UdpEchoClient()
UdpEchoClientApplication:SetDataSize(1024)
UdpEchoServerApplication:StartApplication()
UdpEchoClientApplication:StartApplication()
UdpEchoClientApplication:ScheduleTransmit()
UdpEchoClientApplication:Send()
UdpEchoClientApplication:Send(): Sent 1024 bytes to 10.1.1.2
UdpEchoServerApplication:HandleRead(): Received 1024 bytes from 10.1.1.1
UdpEchoServerApplication:HandleRead(): Echoing packet
UdpEchoClientApplication:HandleRead(0x624920, 0x625160)
UdpEchoClientApplication:HandleRead(): Received 1024 bytes from 10.1.1.2
UdpEchoServerApplication:StopApplication()
UdpEchoClientApplication:StopApplication()
UdpEchoClientApplication:DoDispose()
UdpEchoServerApplication:DoDispose()
UdpEchoClientApplication:~UdpEchoClient()
UdpEchoServerApplication:~UdpEchoServer()
```

It is also sometimes useful to be able to see the simulation time at which a log message is generated. You can do this by ORing in the prefix_time bit.

```
export 'NS_LOG=UdpEchoClientApplication=level_all|prefix_func|prefix_time:
               UdpEchoServerApplication=level_all|prefix_func|prefix_time'
```

Again, you will have to remove the newline above. If you run the script now, you should see the following output:

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build'
'build' finished successfully (0.418s)
0s UdpEchoServerApplication:UdpEchoServer()
0s UdpEchoClientApplication:UdpEchoClient()
0s UdpEchoClientApplication:SetDataSize(1024)
1s UdpEchoServerApplication:StartApplication()
```

```
2s UdpEchoClientApplication:StartApplication()
2s UdpEchoClientApplication:ScheduleTransmit()
2s UdpEchoClientApplication:Send()
2s UdpEchoClientApplication:Send(): Sent 1024 bytes to 10.1.1.2
2.00369s UdpEchoServerApplication:HandleRead(): Received 1024 bytes from 10.1.1.1
2.00369s UdpEchoServerApplication:HandleRead(): Echoing packet
2.00737s UdpEchoClientApplication:HandleRead(0x624290, 0x624ad0)
2.00737s UdpEchoClientApplication:HandleRead(): Received 1024 bytes from 10.1.1.2
10s UdpEchoServerApplication:StopApplication()
10s UdpEchoClientApplication:StopApplication()
UdpEchoClientApplication:DoDispose()
UdpEchoServerApplication:DoDispose()
UdpEchoClientApplication:~UdpEchoClient()
UdpEchoServerApplication:~UdpEchoServer()
```

You can see that the constructor for the UdpEchoServer was called at a simulation time of 0 seconds. This is actually happening before the simulation starts, but the time is displayed as zero seconds. The same is true for the UdpEchoClient constructor message.

Recall that the `scratch/first.cc` script started the echo server application at one second into the simulation. You can now see that the `StartApplication` method of the server is, in fact, called at one second. You can also see that the echo client application is started at a simulation time of two seconds as we requested in the script.

You can now follow the progress of the simulation from the `ScheduleTransmit` call in the client that calls `Send` to the `HandleRead` callback in the echo server application. Note that the elapsed time for the packet to be sent across the point-to-point link is 3.69 milliseconds. You see the echo server logging a message telling you that it has echoed the packet and then, after another channel delay, you see the echo client receive the echoed packet in its `HandleRead` method.

There is a lot that is happening under the covers in this simulation that you are not seeing as well. You can very easily follow the entire process by turning on all of the logging components in the system. Try setting the `NS_LOG` variable to the following,

```
export 'NS_LOG=*=level_all|prefix_func|prefix_time'
```

The asterisk above is the logging component wildcard. This will turn on all of the logging in all of the components used in the simulation. I won't reproduce the output here (as of this writing it produces 1265 lines of output for the single packet echo) but you can redirect this information into a file and look through it with your favorite editor if you like,

```
./waf --run scratch/myfirst > log.out 2>&1
```

I personally use this extremely verbose version of logging when I am presented with a problem and I have no idea where things are going wrong. I can follow the progress of the code quite easily without having to set breakpoints and step through code in a debugger. I can just edit up the output in my favorite editor and search around for things I expect, and see things happening that I don't expect. When I have a general idea about what is going wrong, I transition into a debugger for a fine-grained examination of the problem. This kind of output can be especially useful when your script does something completely unexpected. If you are stepping using a debugger you may miss an unexpected excursion completely. Logging the excursion makes it quickly visible.

5.1.3 Adding Logging to your Code

You can add new logging to your simulations by making calls to the log component via several macros. Let's do so in the `myfirst.cc` script we have in the `scratch` directory.

Recall that we have defined a logging component in that script:

```
NS_LOG_COMPONENT_DEFINE ("FirstScriptExample");
```

You now know that you can enable all of the logging for this component by setting the `NS_LOG` environment variable to the various levels. Let's go ahead and add some logging to the script. The macro used to add an informational level log message is `NS_LOG_INFO`. Go ahead and add one (just before we start creating the nodes) that tells you that the script is "Creating Topology." This is done as in this code snippet,

Open `scratch/myfirst.cc` in your favorite editor and add the line,

```
NS_LOG_INFO ("Creating Topology");
```

right before the lines,

```
NodeContainer nodes;
nodes.Create (2);
```

Now build the script using waf and clear the `NS_LOG` variable to turn off the torrent of logging we previously enabled:

```
./waf
export NS_LOG=
```

Now, if you run the script,

```
./waf --run scratch/myfirst
```

you will *not* see your new message since its associated logging component (`FirstScriptExample`) has not been enabled. In order to see your message you will have to enable the `FirstScriptExample` logging component with a level greater than or equal to `NS_LOG_INFO`. If you just want to see this particular level of logging, you can enable it by,

```
export NS_LOG=FirstScriptExample=info
```

If you now run the script you will see your new "Creating Topology" log message,

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (0.404s)
Creating Topology
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.1.2
```

5.2 Using Command Line Arguments

5.2.1 Overriding Default Attributes

Another way you can change how `ns-3` scripts behave without editing and building is via *command line arguments.* We provide a mechanism to parse command line arguments and automatically set local and global variables based on those arguments.

The first step in using the command line argument system is to declare the command line parser. This is done quite simply (in your main program) as in the following code,

```
  int
main (int argc, char *argv[])
{
  ...

  CommandLine cmd;
  cmd.Parse (argc, argv);

  ...
}
```

This simple two line snippet is actually very useful by itself. It opens the door to the `ns-3` global variable and `Attribute` systems. Go ahead and add that two lines of code to the `scratch/myfirst.cc` script at the start of `main`. Go ahead and build the script and run it, but ask the script for help in the following way,

```
./waf --run "scratch/myfirst --PrintHelp"
```

This will ask Waf to run the `scratch/myfirst` script and pass the command line argument `--PrintHelp` to the script. The quotes are required to sort out which program gets which argument. The command line parser will now see the `--PrintHelp` argument and respond with,

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (0.413s)
TcpL4Protocol:TcpStateMachine()
CommandLine:HandleArgument(): Handle arg name=PrintHelp value=
--PrintHelp: Print this help message.
--PrintGroups: Print the list of groups.
--PrintTypeIds: Print all TypeIds.
--PrintGroup=[group]: Print all TypeIds of group.
--PrintAttributes=[typeid]: Print all attributes of typeid.
--PrintGlobals: Print the list of globals.
```

Let's focus on the `--PrintAttributes` option. We have already hinted at the `ns-3 Attribute` system while walking through the `first.cc` script. We looked at the following lines of code,

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

and mentioned that `DataRate` was actually an `Attribute` of the `PointToPointNetDevice`. Let's use the command line argument parser to take a look at the `Attributes` of the PointToPointNetDevice. The help listing says that we should provide a `TypeId`. This corresponds to the class name of the class to which the `Attributes` belong. In this case it will be `ns3::PointToPointNetDevice`. Let's go ahead and type in,

```
./waf --run "scratch/myfirst --PrintAttributes=ns3::PointToPointNetDevice"
```

The system will print out all of the `Attributes` of this kind of net device. Among the `Attributes` you will see listed is,

```
--ns3::PointToPointNetDevice::DataRate=[32768bps]:
  The default data rate for point to point links
```

This is the default value that will be used when a `PointToPointNetDevice` is created in the system. We overrode this default with the `Attribute` setting in the `PointToPointHelper` above. Let's use the default values for the point-to-point devices and channels by deleting the `SetDeviceAttribute` call and the `SetChannelAttribute` call from the `myfirst.cc` we have in the scratch directory.

Your script should now just declare the `PointToPointHelper` and not do any `set` operations as in the following example,

```
...

NodeContainer nodes;
nodes.Create (2);

PointToPointHelper pointToPoint;

NetDeviceContainer devices;
devices = pointToPoint.Install (nodes);

...
```

Go ahead and build the new script with Waf (`./waf`) and let's go back and enable some logging from the UDP echo server application and turn on the time prefix.

```
export 'NS_LOG=UdpEchoServerApplication=level_all|prefix_time'
```

If you run the script, you should now see the following output,

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build'
'build' finished successfully (0.405s)
0s UdpEchoServerApplication:UdpEchoServer()
1s UdpEchoServerApplication:StartApplication()
Sent 1024 bytes to 10.1.1.2
2.25732s Received 1024 bytes from 10.1.1.1
2.25732s Echoing packet
Received 1024 bytes from 10.1.1.2
10s UdpEchoServerApplication:StopApplication()
UdpEchoServerApplication:DoDispose()
UdpEchoServerApplication:~UdpEchoServer()
```

Recall that the last time we looked at the simulation time at which the packet was received by the echo server, it was at 2.00369 seconds.

```
2.00369s UdpEchoServerApplication:HandleRead(): Received 1024 bytes from 10.1.1.1
```

Now it is receiving the packet at 2.25732 seconds. This is because we just dropped the data rate of the `PointToPointNetDevice` down to its default of 32768 bits per second from five megabits per second.

If we were to provide a new DataRate using the command line, we could speed our simulation up again. We do this in the following way, according to the formula implied by the help item:

```
./waf --run "scratch/myfirst --ns3::PointToPointNetDevice::DataRate=5Mbps"
```

This will set the default value of the DataRate Attribute back to five megabits per second. Are you surprised by the result? It turns out that in order to get the original behavior of the script back, we will have to set the speed-of-light delay of the channel as well. We can ask the command line system to print out the Attributes of the channel just like we did for the net device:

```
./waf --run "scratch/myfirst --PrintAttributes=ns3::PointToPointChannel"
```

We discover the Delay Attribute of the channel is set in the following way:

```
--ns3::PointToPointChannel::Delay=[0ns]:
  Transmission delay through the channel
```

We can then set both of these default values through the command line system,

```
./waf --run "scratch/myfirst
  --ns3::PointToPointNetDevice::DataRate=5Mbps
  --ns3::PointToPointChannel::Delay=2ms"
```

in which case we recover the timing we had when we explicitly set the DataRate and Delay in the script:

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/bu
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/bui
’build’ finished successfully (0.417s)
0s UdpEchoServerApplication:UdpEchoServer()
1s UdpEchoServerApplication:StartApplication()
Sent 1024 bytes to 10.1.1.2
2.00369s Received 1024 bytes from 10.1.1.1
2.00369s Echoing packet
Received 1024 bytes from 10.1.1.2
10s UdpEchoServerApplication:StopApplication()
UdpEchoServerApplication:DoDispose()
UdpEchoServerApplication:~UdpEchoServer()
```

Note that the packet is again received by the server at 2.00369 seconds. We could actually set any of the Attributes used in the script in this way. In particular we could set the UdpEchoClient Attribute MaxPackets to some other value than one.

How would you go about that? Give it a try. Remember you have to comment out the place we override the default Attribute and explicitly set MaxPackets in the script. Then you have to rebuild the script. You will also have to find the syntax for actually setting the new default atribute value using the command line help facility. Once you have this figured out you should be able to control the number of packets echoed from the command line. Since we’re nice folks, we’ll tell you that your command line should end up looking something like,

```
./waf --run "scratch/myfirst
  --ns3::PointToPointNetDevice::DataRate=5Mbps
  --ns3::PointToPointChannel::Delay=2ms
  --ns3::UdpEchoClient::MaxPackets=2"
```

5.2.2 Hooking Your Own Values

You can also add your own hooks to the command line system. This is done quite simply by using the `AddValue` method to the command line parser.

Let's use this facility to specify the number of packets to echo in a completely different way. Let's add a local variable called `nPackets` to the `main` function. We'll initialize it to one to match our previous default behavior. To allow the command line parser to change this value, we need to hook the value into the parser. We do this by adding a call to `AddValue`. Go ahead and change the `scratch/myfirst.cc` script to start with the following code,

```
int
main (int argc, char *argv[])
{
  uint32_t nPackets = 1;

  CommandLine cmd;
  cmd.AddValue("nPackets", "Number of packets to echo", nPackets);
  cmd.Parse (argc, argv);

  ...
```

Scroll down to the point in the script where we set the `MaxPackets Attribute` and change it so that it is set to the variable `nPackets` instead of the constant `1` as is shown below.

```
echoClient.SetAttribute ("MaxPackets", UintegerValue (nPackets));
```

Now if you run the script and provide the `--PrintHelp` argument, you should see your new `User Argument` listed in the help display.

Try,

```
./waf --run "scratch/myfirst --PrintHelp"
```

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (0.403s)
--PrintHelp: Print this help message.
--PrintGroups: Print the list of groups.
--PrintTypeIds: Print all TypeIds.
--PrintGroup=[group]: Print all TypeIds of group.
--PrintAttributes=[typeid]: Print all attributes of typeid.
--PrintGlobals: Print the list of globals.
User Arguments:
    --nPackets: Number of packets to echo
```

If you want to specify the number of packets to echo, you can now do so by setting the `--nPackets` argument in the command line,

```
./waf --run "scratch/myfirst --nPackets=2"
```

You should now see

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (0.404s)
```

```
0s UdpEchoServerApplication:UdpEchoServer()
1s UdpEchoServerApplication:StartApplication()
Sent 1024 bytes to 10.1.1.2
2.25732s Received 1024 bytes from 10.1.1.1
2.25732s Echoing packet
Received 1024 bytes from 10.1.1.2
Sent 1024 bytes to 10.1.1.2
3.25732s Received 1024 bytes from 10.1.1.1
3.25732s Echoing packet
Received 1024 bytes from 10.1.1.2
10s UdpEchoServerApplication:StopApplication()
UdpEchoServerApplication:DoDispose()
UdpEchoServerApplication:~UdpEchoServer()
```

You have now echoed two packets. Pretty easy, isn't it?

You can see that if you are an `ns-3` user, you can use the command line argument system to control global values and `Attributes`. If you are a model author, you can add new `Attributes` to your `Objects` and they will automatically be available for setting by your users through the command line system. If you are a script author, you can add new variables to your scripts and hook them into the command line system quite painlessly.

5.3 Using the Tracing System

The whole point of simulation is to generate output for further study, and the `ns-3` tracing system is a primary mechanism for this. Since `ns-3` is a C++ program, standard facilities for generating output from C++ programs could be used:

```
#include <iostream>
...
int main ()
{
  ...
  std::cout << "The value of x is " << x << std::endl;
  ...
}
```

You could even use the logging module to add a little structure to your solution. There are many well-known problems generated by such approaches and so we have provided a generic event tracing subsystem to address the issues we thought were important.

The basic goals of the `ns-3` tracing system are:

- For basic tasks, the tracing system should allow the user to generate standard tracing for popular tracing sources, and to customize which objects generate the tracing;
- Intermediate users must be able to extend the tracing system to modify the output format generated, or to insert new tracing sources, without modifying the core of the simulator;
- Advanced users can modify the simulator core to add new tracing sources and sinks.

The `ns-3` tracing system is built on the concepts of independent tracing sources and tracing sinks, and a uniform mechanism for connecting sources to sinks. Trace sources are

entities that can signal events that happen in a simulation and provide access to interesting underlying data. For example, a trace source could indicate when a packet is received by a net device and provide access to the packet contents for interested trace sinks.

Trace sources are not useful by themselves, they must be "connected" to other pieces of code that actually do something useful with the information provided by the sink. Trace sinks are consumers of the events and data provided by the trace sources. For example, one could create a trace sink that would (when connected to the trace source of the previous example) print out interesting parts of the received packet.

The rationale for this explicit division is to allow users to attach new types of sinks to existing tracing sources, without requiring editing and recompilation of the core of the simulator. Thus, in the example above, a user could define a new tracing sink in her script and attach it to an existing tracing source defined in the simulation core by editing only the user script.

In this tutorial, we will walk through some pre-defined sources and sinks and show how they may be customized with little user effort. See the ns-3 manual or how-to sections for information on advanced tracing configuration including extending the tracing namespace and creating new tracing sources.

5.3.1 ASCII Tracing

`Ns-3` provides helper functionality that wraps the low-level tracing system to help you with the details involved in configuring some easily understood packet traces. If you enable this functionality, you will see output in a ASCII files — thus the name. For those familiar with `ns-2` output, this type of trace is analogous to the `out.tr` generated by many scripts.

Let's just jump right in and add some ASCII tracing output to our `scratch/myfirst.cc` script.

The first thing you need to do is to add the following include to the top of the script just after the GNU GPL comment:

```
#include <fstream>
```

Then, right before the call to `Simulator::Run ()`, add the following lines of code:

```
std::ofstream ascii;
ascii.open ("myfirst.tr");
PointToPointHelper::EnableAsciiAll (ascii);
```

The first two lines are just vanilla C++ code to open a stream that will be written to a file named "myfirst.tr". See your favorite C++ tutorial if you are unfamiliar with this code. The last line of code in the snippet above tells `ns-3` that you want to enable ASCII tracing on all point-to-point devices in your simulation; and you want the (provided) trace sinks to write out information about packet movement in ASCII format to the stream provided. For those familiar with `ns-2`, the traced events are equivalent to the popular trace points that log `"+"`, `"-"`, `"d"`, and `"r"` events.

You can now build the script and run it from the command line:

```
./waf --run scratch/myfirst
```

Just as you have seen many times before, you will see some messages from Waf and then "'build' finished successfully" with some number of messages from the running program.

When it ran, the program will have created a file named `myfirst.tr`. Because of the way that Waf works, the file is not created in the local directory, it is created at the top-level directory of the repository by default. If you want to control where the traces are saved you can use the `--cwd` option of Waf to specify this. We have not done so, thus we need to change into the top level directory of our repo and take a look at the ASCII trace file `myfirst.tr` in your favorite editor.

5.3.1.1 Parsing Ascii Traces

There's a lot of information there in a pretty dense form, but the first thing to notice is that there are a number of distinct lines in this file. It may be difficult to see this clearly unless you widen your window considerably.

Each line in the file corresponds to a *trace event*. In this case we are tracing events on the *transmit queue* present in every point-to-point net device in the simulation. The transmit queue is a queue through which every packet destined for a point-to-point channel must pass. Note that each line in the trace file begins with a lone character (has a space after it). This character will have the following meaning:

- `+`: An enqueue operation occurred on the device queue;
- `-`: A dequeue operation occurred on the device queue;
- `d`: A packet was dropped, typically because the queue was full;
- `r`: A packet was received by the net device.

Let's take a more detailed view of the first line in the trace file. I'll break it down into sections (indented for clarity) with a two digit reference number on the left side:

```
00 +
01 2
02 /NodeList/0/DeviceList/0/$ns3::PointToPointNetDevice/TxQueue/Enqueue
03 ns3::PppHeader (
04   Point-to-Point Protocol: IP (0x0021))
05   ns3::Ipv4Header (
06     tos 0x0 ttl 64 id 0 protocol 17 offset 0 flags [none]
07     length: 1052 10.1.1.1 > 10.1.1.2)
08     ns3::UdpHeader (
09       length: 1032 49153 > 9)
10       Payload (size=1024)
```

The first line of this expanded trace event (reference number 00) is the operation. We have a `+` character, so this corresponds to an *enqueue* operation on the transmit queue. The second line (reference 01) is the simulation time expressed in seconds. You may recall that we asked the `UdpEchoClientApplication` to start sending packets at two seconds. Here we see confirmation that this is, indeed, happening.

The next line of the example trace (reference 02) tell us which trace source originated this event (expressed in the tracing namespace). You can think of the tracing namespace somewhat like you would a filesystem namespace. The root of the namespace is the `NodeList`. This corresponds to a container managed in the `ns-3` core code that contains all of the nodes that are created in a script. Just as a filesystem may have directories under the root, we may have node numbers in the `NodeList`. The string `/NodeList/0` therefore refers to

the zeroth node in the `NodeList` which we typically think of as "node 0". In each node there is a list of devices that have been installed. This list appears next in the namespace. You can see that this trace event comes from `DeviceList/0` which is the zeroth device installed in the node.

The next string, `$ns3::PointToPointNetDevice` tells you what kind of device is in the zeroth position of the device list for node zero. Recall that the operation `+` found at reference 00 meant that an enqueue operation happened on the transmit queue of the device. This is reflected in the final segments of the "trace path" which are `TxQueue/Enqueue`.

The remaining lines in the trace should be fairly intuitive. References 03-04 indicate that the packet is encapsulated in the point-to-point protocol. References 05-07 show that the packet has an IP version four header and has originated from IP address 10.1.1.1 and is destined for 10.1.1.2. References 08-09 show that this packet has a UDP header and, finally, reference 10 shows that the payload is the expected 1024 bytes.

The next line in the trace file shows the same packet being dequeued from the transmit queue on the same node.

The Third line in the trace file shows the packet being received by the net device on the node with the echo server. I have reproduced that event below.

```
00 r
01 2.25732
02 /NodeList/1/DeviceList/0/$ns3::PointToPointNetDevice/MacRx
03   ns3::Ipv4Header (
04     tos 0x0 ttl 64 id 0 protocol 17 offset 0 flags [none]
05     length: 1052 10.1.1.1 > 10.1.1.2)
06     ns3::UdpHeader (
07       length: 1032 49153 > 9)
08       Payload (size=1024)
```

Notice that the trace operation is now `r` and the simulation time has increased to 2.25732 seconds. If you have been following the tutorial steps closely this means that you have left the `DataRate` of the net devices and the channel `Delay` set to their default values. This time should be familiar as you have seen it before in a previous section.

The trace source namespace entry (reference 02) has changed to reflect that this event is coming from node 1 (`/NodeList/1`) and the packet reception trace source (`/MacRx`). It should be quite easy for you to follow the progress of the packet through the topology by looking at the rest of the traces in the file.

5.3.2 PCAP Tracing

The `ns-3` device helpers can also be used to create trace files in the `.pcap` format. The acronym pcap (usually written in lower case) stands for *p*acket *cap*ture, and is actually an API that includes the definition of a `.pcap` file format. The most popular program that can read and display this format is Wireshark (formerly called Ethereal). However, there are many traffic trace analyzers that use this packet format. We encourage users to exploit the many tools available for analyzing pcap traces. In this tutorial, we concentrate on viewing pcap traces with tcpdump.

The code used to enable pcap tracing is a one-liner.

```
PointToPointHelper::EnablePcapAll ("myfirst");
```

Go ahead and insert this line of code after the ASCII tracing code we just added to `scratch/myfirst.cc`. Notice that we only passed the string "myfirst," and not "myfirst.pcap" or something similar. This is because the parameter is a prefix, not a complete file name. The helper will actually create a trace file for every point-to-point device in the simulation. The file names will be built using the prefix, the node number, the device number and a ".pcap" suffix.

In our example script, we will eventually see files named "myfirst-0-0.pcap" and "myfirst-1-0.pcap" which are the pcap traces for node 0-device 0 and node 1-device 0, respectively.

Once you have added the line of code to enable pcap tracing, you can run the script in the usual way:

```
./waf --run scratch/myfirst
```

If you look at the top level directory of your distribution, you should now see three log files: `myfirst.tr` is the ASCII trace file we have previously examined. `myfirst-0-0.pcap` and `myfirst-1-0.pcap` are the new pcap files we just generated.

5.3.2.1 Reading output with tcpdump

The easiest thing to do at this point will be to use `tcpdump` to look at the `pcap` files.

```
tcpdump -nn -tt -r myfirst-0-0.pcap
reading from file myfirst-0-0.pcap, link-type PPP (PPP)
2.000000 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 1024
2.514648 IP 10.1.1.2.9 > 10.1.1.1.49153: UDP, length 1024

tcpdump -nn -tt -r myfirst-1-0.pcap
reading from file myfirst-1-0.pcap, link-type PPP (PPP)
2.257324 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 1024
2.257324 IP 10.1.1.2.9 > 10.1.1.1.49153: UDP, length 1024
```

You can see in the dump of `myfirst-0-0.pcap` (the client device) that the echo packet is sent at 2 seconds into the simulation. If you look at the second dump (`myfirst-1-0.pcap`) you can see that packet being received at 2.257324 seconds. You see the packet being echoed back at 2.257324 seconds in the second dump, and finally, you see the packet being received back at the client in the first dump at 2.514648 seconds.

5.3.2.2 Reading output with Wireshark

If you are unfamilar with Wireshark, there is a web site available from which you can download programs and documentation: http://www.wireshark.org/.

Wireshark is a graphical user interface which can be used for displaying these trace files. If you have Wireshark available, you can open each of the trace files and display the contents as if you had captured the packets using a *packet sniffer*.

6 Building Topologies

6.1 Building a Bus Network Topology

In this section we are going to expand our mastery of ns-3 network devices and channels to cover an example of a bus network. Ns-3 provides a net device and channel we call CSMA (Carrier Sense Multiple Access).

The ns-3 CSMA device models a simple network in the spirit of Ethernet. A real Ethernet uses CSMA/CD (Carrier Sense Multiple Access with Collision Detection) scheme with exponentially increasing backoff to contend for the shared transmission medium. The ns-3 CSMA device and channel models only a subset of this.

Just as we have seen point-to-point topology helper objects when constructing point-to-point topologies, we will see equivalent CSMA topology helpers in this section. The appearance and operation of these helpers should look quite familiar to you.

We provide an example script in our examples directory. This script builds on the first.cc script and adds a CSMA network to the point-to-point simulation we've already considered. Go ahead and open examples/second.cc in your favorite editor. You will have already seen enough ns-3 code to understand most of what is going on in this example, but we will go over the entire script and examine some of the output.

Just as in the first.cc example (and in all ns-3 examples) the file begins with an emacs mode line and some GPL boilerplate.

The actual code begins by loading module include files just as was done in the first.cc example.

```
#include "ns3/core-module.h"
#include "ns3/simulator-module.h"
#include "ns3/node-module.h"
#include "ns3/helper-module.h"
```

One thing that can be surprisingly useful is a small bit of ASCII art that shows a cartoon of the network topology constructed in the example. You will find a similar "drawing" in most of our examples.

In this case, you can see that we are going to extend our point-to-point example (the link between the nodes n0 and n1 below) by hanging a bus network off of the right side. Notice that this is the default network topology since you can actually vary the number of nodes created on the LAN. If you set nCsma to one, there will be a total of two nodes on the LAN (CSMA channel) — one required node and one "extra" node. By default there are three "extra" nodes as seen below:

```
// Default Network Topology
//
//       10.1.1.0
// n0 -------------- n1   n2   n3   n4
//    point-to-point  |    |    |    |
//                    ================
//                      LAN 10.1.2.0
```

Then the ns-3 namespace is used and a logging component is defined. This is all just as it was in first.cc, so there is nothing new yet.

```
using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("SecondScriptExample");
```

The main program begins with a slightly different twist. We use a verbose flag to determine whether or not the `UdpEchoClientApplication` and `UdpEchoServerApplication` logging components are enabled. This flag defaults to true (the logging components are enabled) but allows us to turn off logging during regression testing of this example.

You will see some familiar code that will allow you to change the number of devices on the CSMA network via command line argument. We did something similar when we allowed the number of packets sent to be changed in the section on command line arguments. The last line makes sure you have at least one "extra" node.

The code consists of variations of previously covered API so you should be entirely comfortable with the following code at this point in the tutorial.

```
bool verbose = true;
uint32_t nCsma = 3;

CommandLine cmd;
cmd.AddValue (``nCsma'', ``Number of \"extra\" CSMA nodes/devices'', nCsma);
cmd.AddValue (``verbose'', ``Tell echo applications to log if true'', verbose);

cmd.Parse (argc,argv);

if (verbose)
  {
    LogComponentEnable(``UdpEchoClientApplication'', LOG_LEVEL_INFO);
    LogComponentEnable(``UdpEchoServerApplication'', LOG_LEVEL_INFO);
  }

nCsma = nCsma == 0 ? 1 : nCsma;
```

The next step is to create two nodes that we will connect via the point-to-point link. The `NodeContainer` is used to do this just as was done in `first.cc`.

```
NodeContainer p2pNodes;
p2pNodes.Create (2);
```

Next, we declare another `NodeContainer` to hold the nodes that will be part of the bus (CSMA) network. First, we just instantiate the container object itself.

```
NodeContainer csmaNodes;
csmaNodes.Add (p2pNodes.Get (1));
csmaNodes.Create (nCsma);
```

The next line of code `Gets` the first node (as in having an index of one) from the point-to-point node container and adds it to the container of nodes that will get CSMA devices. The node in question is going to end up with a point-to-point device *and* a CSMA device. We then create a number of "extra" nodes that compose the remainder of the CSMA network. Since we already have one node in the CSMA network – the one that will have both a point-to-point and CSMA net device, the number of "extra" nodes means the number nodes you desire in the CSMA section minus one.

The next bit of code should be quite familiar by now. We instantiate a `PointToPointHelper` and set the associated default `Attributes` so that we create a five megabit per second transmitter on devices created using the helper and a two millisecond delay on channels created by the helper.

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));

NetDeviceContainer p2pDevices;
p2pDevices = pointToPoint.Install (p2pNodes);
```

We then instantiate a `NetDeviceContainer` to keep track of the point-to-point net devices and we `Install` devices on the point-to-point nodes.

We mentioned above that you were going to see a helper for CSMA devices and channels, and the next lines introduce them. The `CsmaHelper` works just like a `PointToPointHelper`, but it creates and connects CSMA devices and channels. In the case of a CSMA device and channel pair, notice that the data rate is specified by a *channel* `Attribute` instead of a device `Attribute`. This is because a real CSMA network does not allow one to mix, for example, 10Base-T and 100Base-T devices on a given channel. We first set the data rate to 100 megabits per second, and then set the speed-of-light delay of the channel to 6560 nano-seconds (arbitrarily chosen as 1 nanosecond per foot over a 100 meter segment). Notice that you can set an `Attribute` using its native data type.

```
CsmaHelper csma;
csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));

NetDeviceContainer csmaDevices;
csmaDevices = csma.Install (csmaNodes);
```

Just as we created a `NetDeviceContainer` to hold the devices created by the `PointToPointHelper` we create a `NetDeviceContainer` to hold the devices created by our `CsmaHelper`. We call the `Install` method of the `CsmaHelper` to install the devices into the nodes of the `csmaNodes NodeContainer`.

We now have our nodes, devices and channels created, but we have no protocol stacks present. Just as in the `first.cc` script, we will use the `InternetStackHelper` to install these stacks.

```
InternetStackHelper stack;
stack.Install (p2pNodes.Get (0));
stack.Install (csmaNodes);
```

Recall that we took one of the nodes from the `p2pNodes` container and added it to the `csmaNodes` container. Thus we only need to install the stacks on the remaining `p2pNodes` node, and all of the nodes in the `csmaNodes` container to cover all of the nodes in the simulation.

Just as in the `first.cc` example script, we are going to use the `Ipv4AddressHelper` to assign IP addresses to our device interfaces. First we use the network 10.1.1.0 to create the two addresses needed for our two point-to-point devices.

```
Ipv4AddressHelper address;
address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer p2pInterfaces;
p2pInterfaces = address.Assign (p2pDevices);
```

Recall that we save the created interfaces in a container to make it easy to pull out addressing information later for use in setting up the applications.

We now need to assign IP addresses to our CSMA device interfaces. The operation works just as it did for the point-to-point case, except we now are performing the operation on a container that has a variable number of CSMA devices — remember we made the number of CSMA devices changeable by command line argument. The CSMA devices will be associated with IP addresses from network number 10.1.2.0 in this case, as seen below.

```
address.SetBase ("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer csmaInterfaces;
csmaInterfaces = address.Assign (csmaDevices);
```

Now we have a topology built, but we need applications. This section is going to be fundamentally similar to the applications section of `first.cc` but we are going to instantiate the server on one of the nodes that has a CSMA device and the client on the node having only a point-to-point device.

First, we set up the echo server. We create a `UdpEchoServerHelper` and provide a required `Attribute` value to the constructor which is the server port number. Recall that this port can be changed later using the `SetAttribute` method if desired, but we require it to be provided to the constructor.

```
UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (csmaNodes.Get (nCsma));
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
```

Recall that the `csmaNodes` `NodeContainer` contains one of the nodes created for the point-to-point network and `nCsma` "extra" nodes. What we want to get at is the last of the "extra" nodes. The zeroth entry of the `csmaNodes` container will be the point-to-point node. The easy way to think of this, then, is if we create one "extra" CSMA node, then it will be at index one of the `csmaNodes` container. By induction, if we create `nCsma` "extra" nodes the last one will be at index `nCsma`. You see this exhibited in the `Get` of the first line of code.

The client application is set up exactly as we did in the `first.cc` example script. Again, we provide required `Attributes` to the `UdpEchoClientHelper` in the constructor (in this case the remote address and port). We tell the client to send packets to the server we just installed on the last of the "extra" CSMA nodes. We install the client on the leftmost point-to-point node seen in the topology illustration.

```
UdpEchoClientHelper echoClient (csmaInterfaces.GetAddress (nCsma), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer clientApps = echoClient.Install (p2pNodes.Get (0));
```

```
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

Since we have actually built an internetwork here, we need some form of internetwork routing. `ns-3` provides what we call global routing to help you out. Global routing takes advantage of the fact that the entire internetwork is accessible in the simulation and runs through the all of the nodes created for the simulation — it does the hard work of setting up routing for you without having to configure routers.

Basically, what happens is that each node behaves as if it were an OSPF router that communicates instantly and magically with all other routers behind the scenes. Each node generates link advertisements and communicates them directly to a global route manager which uses this global information to construct the routing tables for each node. Setting up this form of routing is a one-liner:

```
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
```

Next we enable pcap tracing. The first line of code to enable pcap tracing in the point-to-point helper should be familiar to you by now. The second line enables pcap tracing in the CSMA helper and there is an extra parameter you haven't encountered yet.

```
PointToPointHelper::EnablePcapAll ("second");
CsmaHelper::EnablePcap ("second", csmaDevices.Get (1), true);
```

The CSMA network is a multi-point-to-point network. This means that there can (and are in this case) multiple endpoints on a shared medium. Each of these endpoints has a net device associated with it. There are two basic alternatives to gathering trace information from such a network. One way is to create a trace file for each net device and store only the packets that are emitted or consumed by that net device. Another way is to pick one of the devices and place it in promiscuous mode. That single device then "sniffs" the network for all packets and stores them in a single pcap file. This is how `tcpdump`, for example, works. That final parameter tells the CSMA helper whether or not to arrange to capture packets in promiscuous mode.

In this example, we are going to select one of the devices on the CSMA network and ask it to perform a promiscuous sniff of the network, thereby emulating what `tcpdump` would do. If you were on a Linux machine you might do something like `tcpdump -i eth0` to get the trace. In this case, we specify the device using `csmaDevices.Get(1)`, which selects the first device in the container. Setting the final parameter to true enables promiscuous captures.

The last section of code just runs and cleans up the simulation just like the `first.cc` example.

```
  Simulator::Run ();
  Simulator::Destroy ();
  return 0;
}
```

In order to run this example, copy the `second.cc` example script into the scratch directory and use waf to build just as you did with the `first.cc` example. If you are in the top-level directory of the repository you just type,

```
cp examples/second.cc scratch/mysecond.cc
./waf
```

Warning: We use the file `second.cc` as one of our regression tests to verify that it works exactly as we think it should in order to make your tutorial experience a positive one. This means that an executable named `second` already exists in the project. To avoid any confusion about what you are executing, please do the renaming to `mysecond.cc` suggested above.

If you are following the tutorial religiously (you are, aren't you) you will still have the NS_LOG variable set, so go ahead and clear that variable and run the program.

```
export NS_LOG=
./waf --run scratch/mysecond
```

Since we have set up the UDP echo applications to log just as we did in `first.cc`, you will see similar output when you run the script.

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (0.415s)
Sent 1024 bytes to 10.1.2.4
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.2.4
```

Recall that the first message, "`Sent 1024 bytes to 10.1.2.4`," is the UDP echo client sending a packet to the server. In this case, the server is on a different network (10.1.2.0). The second message, "`Received 1024 bytes from 10.1.1.1`," is from the UDP echo server, generated when it receives the echo packet. The final message, "`Received 1024 bytes from 10.1.2.4`," is from the echo client, indicating that it has received its echo back from the server.

If you now go and look in the top level directory, you will find three trace files:

```
second-0-0.pcap  second-1-0.pcap  second-2-0.pcap
```

Let's take a moment to look at the naming of these files. They all have the same form, `<name>-<node>-<device>.pcap`. For example, the first file in the listing is `second-0-0.pcap` which is the pcap trace from node zero, device zero. This is the point-to-point net device on node zero. The file `second-1-0.pcap` is the pcap trace for device zero on node one, also a point-to-point net device; and the file `second-2-0.pcap` is the pcap trace for device zero on node two.

If you refer back to the topology illustration at the start of the section, you will see that node zero is the leftmost node of the point-to-point link and node one is the node that has both a point-to-point device and a CSMA device. You will see that node two is the first "extra" node on the CSMA network and its device zero was selected as the device to capture the promiscuous-mode trace.

Now, let's follow the echo packet through the internetwork. First, do a tcpdump of the trace file for the leftmost point-to-point node — node zero.

```
tcpdump -nn -tt -r second-0-0.pcap
```

You should see the contents of the pcap file displayed:

```
reading from file second-0-0.pcap, link-type PPP (PPP)
2.000000 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
2.007602 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

The first line of the dump indicates that the link type is PPP (point-to-point) which we expect. You then see the echo packet leaving node zero via the device associated with IP address 10.1.1.1 headed for IP address 10.1.2.4 (the rightmost CSMA node). This packet will move over the point-to-point link and be received by the point-to-point net device on node one. Let's take a look:

```
tcpdump -nn -tt -r second-1-0.pcap
```

You should now see the pcap trace output of the other side of the point-to-point link:

```
reading from file second-1-0.pcap, link-type PPP (PPP)
2.003686 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
2.003915 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

Here we see that the link type is also PPP as we would expect. You see the packet from IP address 10.1.1.1 (that was sent at 2.000000 seconds) headed toward IP address 10.1.2.4 appear on this interface. Now, internally to this node, the packet will be forwarded to the CSMA interface and we should see it pop out on that device headed for its ultimate destination.

Remember that we selected node 2 as the promiscuous sniffer node for the CSMA network so let's then look at second-2-0.pcap and see if its there.

```
tcpdump -nn -tt -r second-2-0.pcap
```

You should now see the promiscuous dump of node two, device zero:

```
reading from file second-2-0.pcap, link-type EN10MB (Ethernet)
2.003696 arp who-has 10.1.2.4 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1
2.003707 arp reply 10.1.2.4 is-at 00:00:00:00:00:06
2.003801 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
2.003811 arp who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.4
2.003822 arp reply 10.1.2.1 is-at 00:00:00:00:00:03
2.003915 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

As you can see, the link type is now "Ethernet". Something new has appeared, though. The bus network needs `ARP`, the Address Resolution Protocol. Node one knows it needs to send the packet to IP address 10.1.2.4, but it doesn't know the MAC address of the corresponding node. It broadcasts on the CSMA network (ff:ff:ff:ff:ff:ff) asking for the device that has IP address 10.1.2.4. In this case, the rightmost node replies saying it is at MAC address 00:00:00:00:00:06. Note that node two is not directly involved in this exchange, but is sniffing the network and reporting all of the traffic it sees.

This exchange is seen in the following lines,

```
2.003696 arp who-has 10.1.2.4 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1
2.003707 arp reply 10.1.2.4 is-at 00:00:00:00:00:06
```

Then node one, device one goes ahead and sends the echo packet to the UDP echo server at IP address 10.1.2.4.

```
2.003801 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
```

The server receives the echo request and turns the packet around trying to send it back to the source. The server knows that this address is on another network that it reaches via IP address 10.1.2.1. This is because we initialized global routing and it has figured all of this out for us. But, the echo server node doesn't know the MAC address of the first CSMA node, so it has to ARP for it just like the first CSMA node had to do.

```
2.003811 arp who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.4
2.003822 arp reply 10.1.2.1 is-at 00:00:00:00:00:03
```

The server then sends the echo back to the forwarding node.

```
2.003915 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

Looking back at the rightmost node of the point-to-point link,

```
tcpdump -nn -tt -r second-1-0.pcap
```

You can now see the echoed packet coming back onto the point-to-point link as the last line of the trace dump.

```
reading from file second-1-0.pcap, link-type PPP (PPP)
2.003686 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
2.003915 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

Lastly, you can look back at the node that originated the echo

```
tcpdump -nn -tt -r second-0-0.pcap
```

and see that the echoed packet arrives back at the source at 2.007602 seconds,

```
reading from file second-0-0.pcap, link-type PPP (PPP)
2.000000 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
2.007602 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

Finally, recall that we added the ability to control the number of CSMA devices in the simulation by command line argument. You can change this argument in the same way as when we looked at changing the number of packets echoed in the `first.cc` example. Try running the program with the number of "extra" devices set to four:

```
./waf --run "scratch/mysecond --nCsma=4"
```

You should now see,

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (0.405s)
Sent 1024 bytes to 10.1.2.5
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.2.5
```

Notice that the echo server has now been relocated to the last of the CSMA nodes, which is 10.1.2.5 instead of the default case, 10.1.2.4.

It is possible that you may not be satisfied with a trace file generated by a bystander in the CSMA network. You may really want to get a trace from a single device and you may not be interested in any other traffic on the network. You can do this fairly easily/

Let's take a look at `scratch/mysecond.cc` and add that code enabling us to be more specific. `ns-3` helpers provide methods that take a node number and device number as parameters. Go ahead and replace the `EnablePcap` calls with the calls below.

```
PointToPointHelper::EnablePcap ("second", p2pNodes.Get (0)->GetId (), 0);
CsmaHelper::EnablePcap ("second", csmaNodes.Get (nCsma)->GetId (), 0, false);
CsmaHelper::EnablePcap ("second", csmaNodes.Get (nCsma-1)->GetId (), 0, false);
```

We know that we want to create a pcap file with the base name "second" and we also know that the device of interest in both cases is going to be zero, so those parameters are not really interesting.

In order to get the node number, you have two choices: first, nodes are numbered in a monotonically increasing fashion starting from zero in the order in which you created them. One way to get a node number is to figure this number out "manually" by contemplating the order of node creation. If you take a look at the network topology illustration at the beginning of the file, we did this for you and you can see that the last CSMA node is going to be node number `nCsma + 1`. This approach can become annoyingly difficult in larger simulations.

An alternate way, which we use here, is to realize that the `NodeContainers` contain pointers to `ns-3` `Node` Objects. The `Node` Object has a method called `GetId` which will return that node's ID, which is the node number we seek. Let's go take a look at the Doxygen for the `Node` and locate that method, which is further down in the `ns-3` core code than we've seen so far; but sometimes you have to search diligently for useful things.

Go to the Doxygen documentation for your release (recall that you can find it on the project web site). You can get to the `Node` documentation by looking through at the "Classes" tab and scrolling down the "Class List" until you find `ns3::Node`. Select `ns3::Node` and you will be taken to the documentation for the `Node` class. If you now scroll down to the `GetId` method and select it, you will be taken to the detailed documentation for the method. Using the `GetId` method can make determining node numbers much easier in complex topologies.

Let's clear the old trace files out of the top-level directory to avoid confusion about what is going on,

```
rm *.pcap
rm *.tr
```

If you build the new script and run the simulation setting `nCsma` to 100,

```
./waf --run "scratch/mysecond --nCsma=100"
```

you will see the following output:

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (0.407s)
Sent 1024 bytes to 10.1.2.101
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.2.101
```

Note that the echo server is now located at 10.1.2.101 which corresponds to having 100 "extra" CSMA nodes with the echo server on the last one. If you list the pcap files in the top level directory you will see,

```
second-0-0.pcap  second-100-0.pcap  second-101-0.pcap
```

The trace file `second-0-0.pcap` is the "leftmost" point-to-point device which is the echo packet source. The file `second-101-0.pcap` corresponds to the rightmost CSMA device which is where the echo server resides. You may have noticed that the final parameter on the call to enable pcap tracing on the echo server node was false. This means that the trace gathered on that node was in non-promiscuous mode.

To illustrate the difference between promiscuous and non-promiscuous traces, we also requested a non-promiscuous trace for the next-to-last node. Go ahead and take a look at the `tcpdump` for `second-100-0.pcap`.

```
tcpdump -nn -tt -r second-100-0.pcap
```

You can now see that node 100 is really a bystander in the echo exchange. The only packets that it receives are the ARP requests which are broadcast to the entire CSMA network.

```
reading from file second-100-0.pcap, link-type EN10MB (Ethernet)
2.003696 arp who-has 10.1.2.101 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1
2.003811 arp who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.101
```

Now take a look at the `tcpdump` for `second-101-0.pcap`.

```
tcpdump -nn -tt -r second-101-0.pcap
```

You can now see that node 101 is really the participant in the echo exchange.

```
reading from file second-101-0.pcap, link-type EN10MB (Ethernet)
2.003696 arp who-has 10.1.2.101 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1
2.003696 arp reply 10.1.2.101 is-at 00:00:00:00:00:67
2.003801 IP 10.1.1.1.49153 > 10.1.2.101.9: UDP, length 1024
2.003801 arp who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.101
2.003822 arp reply 10.1.2.1 is-at 00:00:00:00:00:03
2.003822 IP 10.1.2.101.9 > 10.1.1.1.49153: UDP, length 1024
```

6.2 Building a Wireless Network Topology

In this section we are going to further expand our knowledge of `ns-3` network devices and channels to cover an example of a wireless network. `Ns-3` provides a set of 802.11 models that attempt to provide an accurate MAC-level implementation of the 802.11 specification and a "not-so-slow" PHY-level model of the 802.11a specification.

Just as we have seen both point-to-point and CSMA topology helper objects when constructing point-to-point topologies, we will see equivalent `Wifi` topology helpers in this section. The appearance and operation of these helpers should look quite familiar to you.

We provide an example script in our `examples` directory. This script builds on the `second.cc` script and adds a Wifi network. Go ahead and open `examples/third.cc` in your favorite editor. You will have already seen enough `ns-3` code to understand most of what is going on in this example, but there are a few new things, so we will go over the entire script and examine some of the output.

Just as in the `second.cc` example (and in all `ns-3` examples) the file begins with an emacs mode line and some GPL boilerplate.

Take a look at the ASCII art (reproduced below) that shows the default network topology constructed in the example. You can see that we are going to further extend our example by hanging a wireless network off of the left side. Notice that this is a default network topology since you can actually vary the number of nodes created on the wired and wireless networks. Just as in the `second.cc` script case, if you change `nCsma`, it will give you a number of "extra" CSMA nodes. Similarly, you can set `nWifi` to control how many `STA` (station) nodes are created in the simulation. There will always be one `AP` (access point) node on the wireless network. By default there are three "extra" CSMA nodes and three wireless `STA` nodes.

The code begins by loading module include files just as was done in the `second.cc` example. There are a couple of new includes corresponding to the Wifi module and the mobility module which we will discuss below.

```
#include "ns3/core-module.h"
#include "ns3/simulator-module.h"
#include "ns3/node-module.h"
#include "ns3/helper-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
```

The network topology illustration follows:

```
// Default Network Topology
//
//   Wifi 10.1.3.0
//                  AP
//  *    *    *    *
//  |    |    |    |    10.1.1.0
// n5   n6   n7   n0 -------------- n1   n2   n3   n4
//                   point-to-point  |    |    |    |
//                                   ================
//                                     LAN 10.1.2.0
```

You can see that we are adding a new network device to the node on the left side of the point-to-point link that becomes the access point for the wireless network. A number of wireless STA nodes are created to fill out the new 10.1.3.0 network as shown on the left side of the illustration.

After the illustration, the `ns-3` namespace is `used` and a logging component is defined. This should all be quite familiar by now.

```
using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("ThirdScriptExample");
```

The main program begins just like `second.cc` by adding some command line parameters for enabling or disabling logging components and for changing the number of devices created.

```
bool verbose = true;
uint32_t nCsma = 3;
uint32_t nWifi = 3;

CommandLine cmd;
cmd.AddValue (``nCsma'', ``Number of \"extra\" CSMA nodes/devices'', nCsma);
cmd.AddValue (``nWifi'', ``Number of wifi STA devices'', nWifi);
cmd.AddValue (``verbose'', ``Tell echo applications to log if true'', verbose);

cmd.Parse (argc,argv);

if (verbose)
  {
    LogComponentEnable(``UdpEchoClientApplication'', LOG_LEVEL_INFO);
```

```
    LogComponentEnable(``UdpEchoServerApplication'', LOG_LEVEL_INFO);
  }
```

Just as in all of the previous examples, the next step is to create two nodes that we will connect via the point-to-point link.

```
NodeContainer p2pNodes;
p2pNodes.Create (2);
```

Next, we see an old friend. We instantiate a `PointToPointHelper` and set the associated default `Attributes` so that we create a five megabit per second transmitter on devices created using the helper and a two millisecond delay on channels created by the helper. We then `Intall` the devices on the nodes and the channel between them.

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));

NetDeviceContainer p2pDevices;
p2pDevices = pointToPoint.Install (p2pNodes);
```

Next, we declare another `NodeContainer` to hold the nodes that will be part of the bus (CSMA) network.

```
NodeContainer csmaNodes;
csmaNodes.Add (p2pNodes.Get (1));
csmaNodes.Create (nCsma);
```

The next line of code `Gets` the first node (as in having an index of one) from the point-to-point node container and adds it to the container of nodes that will get CSMA devices. The node in question is going to end up with a point-to-point device and a CSMA device. We then create a number of "extra" nodes that compose the remainder of the CSMA network.

We then instantiate a `CsmaHelper` and set its `Attributes` as we did in the previous example. We create a `NetDeviceContainer` to keep track of the created CSMA net devices and then we `Install` CSMA devices on the selected nodes.

```
CsmaHelper csma;
csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));

NetDeviceContainer csmaDevices;
csmaDevices = csma.Install (csmaNodes);
```

Next, we are going to create the nodes that will be part of the Wifi network. We are going to create a number of "station" nodes as specified by the command line argument, and we are going to use the "leftmost" node of the point-to-point link as the node for the access point.

```
NodeContainer wifiStaNodes;
wifiStaNodes.Create (nWifi);
NodeContainer wifiApNode = p2pNodes.Get (0);
```

The next bit of code constructs the wifi devices and the interconnection channel between these wifi nodes. First, we configure the PHY and channel helpers:

```
YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
YansWifiPhyHelper phy = YansWifiPhyHelper::Default ();
```

For simplicity, this code uses the default PHY layer configuration and channel models which are documented in the API doxygen documentation for the `YansWifiChannelHelper::Default` and `YansWifiPhyHelper::Default` methods. Once these objects are created, we create a channel object and associate it to our PHY layer object manager to make sure that all the PHY layer objects created by the `YansWifiPhyHelper` share the same underlying channel, that is, they share the same wireless medium and can communication and interfere:

```
phy.SetChannel (channel.Create ());
```

Once the PHY helper is configured, we can focus on the MAC layer. Here we choose to work with non-Qos MACs so we use a NqosWifiMacHelper object to set MAC parameters.

```
WifiHelper wifi = WifiHelper::Default ();
wifi.SetRemoteStationManager ("ns3::AarfWifiManager");

NqosWifiMacHelper mac = NqosWifiMacHelper::Default ();
```

The `SetRemoteStationManager` method tells the helper the type of rate control algorithm to use. Here, it is asking the helper to use the AARF algorithm — details are, of course, available in Doxygen.

Next, we configure the type of MAC, the SSID of the infrastructure network we want to setup and make sure that our stations don't perform active probing:

```
Ssid ssid = Ssid ("ns-3-ssid");
mac.SetType ("ns3::NqstaWifiMac",
  "Ssid", SsidValue (ssid),
  "ActiveProbing", BooleanValue (false));
```

This code first creates an 802.11 service set identifier (SSID) object that will be used to set the value of the "Ssid" `Attribute` of the MAC layer implementation. The particular kind of MAC layer is specified by `Attribute` as being of the "ns3::NqstaWifiMac" type. This means that the MAC will use a "non-QoS station" (nqsta) state machine. Finally, the "ActiveProbing" `Attribute` is set to false. This means that probe requests will not be sent by MACs created by this helper.

Once all the station-specific parameters are fully configured, both at the MAC and PHY layers, we can invoke our now-familiar `Install` method to create the wifi devices of these stations:

```
NetDeviceContainer staDevices;
staDevices = wifi.Install (phy, mac, wifiStaNodes);
```

We have configured Wifi for all of our STA nodes, and now we need to configure the AP (access point) node. We begin this process by changing the default `Attributes` of the `NqosWifiMacHelper` to reflect the requirements of the AP.

```
mac.SetType ("ns3::NqapWifiMac",
  "Ssid", SsidValue (ssid),
  "BeaconGeneration", BooleanValue (true),
  "BeaconInterval", TimeValue (Seconds (2.5)));
```

In this case, the NqosWifiMacHelper is going to create MAC layers of the "ns3::NqapWifiMac" (Non-Qos Access Point) type. We set the "BeaconGeneration" Attribute to true and also set an interval between beacons of 2.5 seconds.

The next lines create the single AP which shares the same set of PHY-level Attributes (and channel) as the stations:

```
NetDeviceContainer apDevices;
apDevices = wifi.Install (phy, mac, wifiApNode);
```

Now, we are going to add mobility models. We want the STA nodes to be mobile, wandering around inside a bounding box, and we want to make the AP node stationary. We use the MobilityHelper to make this easy for us. First, we instantiate a MobilityHelper object and set some Attributes controlling the "position allocator" functionality.

```
MobilityHelper mobility;

mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
  "MinX", DoubleValue (0.0),
  "MinY", DoubleValue (0.0),
  "DeltaX", DoubleValue (5.0),
  "DeltaY", DoubleValue (10.0),
  "GridWidth", UintegerValue (3),
  "LayoutType", StringValue ("RowFirst"));
```

This code tells the mobility helper to use a two-dimensional grid to initially place the STA nodes. Feel free to explore the Doxygen for class ns3::GridPositionAllocator to see exactly what is being done.

We have arranged our nodes on an initial grid, but now we need to tell them how to move. We choose the RandomWalk2dMobilityModel which has the nodes move in a random direction at a random speed around inside a bounding box.

```
mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",
  "Bounds", RectangleValue (Rectangle (-50, 50, -50, 50)));
```

We now tell the MobilityHelper to install the mobility models on the STA nodes.

```
mobility.Install (wifiStaNodes);
```

We want the access point to remain in a fixed position during the simulation. We accomplish this by setting the mobility model for this node to be the ns3::ConstantPositionMobilityModel:

```
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (wifiApNode);
```

We now have our nodes, devices and channels created, and mobility models chosen for the Wifi nodes, but we have no protocol stacks present. Just as we have done previously many times, we will use the InternetStackHelper to install these stacks.

```
InternetStackHelper stack;
stack.Install (csmaNodes);
stack.Install (wifiApNode);
stack.Install (wifiStaNodes);
```

Just as in the second.cc example script, we are going to use the Ipv4AddressHelper to assign IP addresses to our device interfaces. First we use the network 10.1.1.0 to create the

two addresses needed for our two point-to-point devices. Then we use network 10.1.2.0 to assign addresses to the CSMA network and then we assign addresses from network 10.1.3.0 to both the STA devices and the AP on the wireless network.

```
Ipv4AddressHelper address;

address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer p2pInterfaces;
p2pInterfaces = address.Assign (p2pDevices);

address.SetBase ("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer csmaInterfaces;
csmaInterfaces = address.Assign (csmaDevices);

address.SetBase ("10.1.3.0", "255.255.255.0");
address.Assign (staDevices);
address.Assign (apDevices);
```

We put the echo server on the "rightmost" node in the illustration at the start of the file. We have done this before.

```
UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (csmaNodes.Get (nCsma));
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
```

And we put the echo client on the last STA node we created, pointing it to the server on the CSMA network. We have also seen similar operations before.

```
UdpEchoClientHelper echoClient (csmaInterfaces.GetAddress (nCsma), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer clientApps =
  echoClient.Install (wifiStaNodes.Get (nWifi - 1));
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

Since we have built an internetwork here, we need to enable internetwork routing just as we did in the `second.cc` example script.

```
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
```

One thing that can surprise some users is the fact that the simulation we just created will never "naturally" stop. This is because we asked the wireless access point to generate beacons. It will generate beacons forever, and this will result in simulator events being scheduled into the future indefinitely, so we must tell the simulator to stop even though it may have beacon generation events scheduled. The following line of code tells the simulator to stop so that we don't simulate beacons forever and enter what is essentially an endless loop.

```
Simulator::Stop (Seconds (10.0));
```

We create just enough tracing to cover all three networks:

```
PointToPointHelper::EnablePcapAll ("third");
phy.EnablePcap ("third", apDevices.Get (0));
CsmaHelper::EnablePcap ("third", csmaDevices.Get (0), true);
```

These three lines of code will start pcap tracing on both of the point-to-point nodes that serves as our backbone, will start a promiscuous (monitor) mode trace on the Wifi network, and will start a promiscuous trace on the CSMA network. This will let us see all of the traffic with a minimum number of trace files.

Finally, we actually run the simulation, clean up and then exit the program.

```
  Simulator::Run ();
  Simulator::Destroy ();
  return 0;
}
```

In order to run this example, you have to copy the `third.cc` example script into the scratch directory and use Waf to build just as you did with the `second.cc` example. If you are in the top-level directory of the repository you would type,

```
cp examples/third.cc scratch/mythird.cc
./waf
./waf --run scratch/mythird
```

Again, since we have set up the UDP echo applications just as we did in the `second.cc` script, you will see similar output.

```
Waf: Entering directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone-3.5-tutorial/ns-3-dev/build’
’build’ finished successfully (0.407s)
Sent 1024 bytes to 10.1.2.4
Received 1024 bytes from 10.1.3.3
Received 1024 bytes from 10.1.2.4
```

Recall that the first message, “`Sent 1024 bytes to 10.1.2.4`,” is the UDP echo client sending a packet to the server. In this case, the client is on the wireless network (10.1.3.0). The second message, “`Received 1024 bytes from 10.1.3.3`,” is from the UDP echo server, generated when it receives the echo packet. The final message, “`Received 1024 bytes from 10.1.2.4`,” is from the echo client, indicating that it has received its echo back from the server.

If you now go and look in the top level directory, you will find four trace files from this simulation, two from node zero and two from node one:

```
third-0-0.pcap  third-0-1.pcap  third-1-0.pcap  third-1-1.pcap
```

The file “third-0-0.pcap” corresponds to the point-to-point device on node zero – the left side of the “backbone”. The file “third-1-0.pcap” corresponds to the point-to-point device on node one – the right side of the “backbone”. The file “third-0-1.pcap” will be the promiscuous (monitor mode) trace from the Wifi network and the file “third-1-1.pcap” will be the promiscuous trace from the CSMA network. Can you verify this by inspecting the code?

Since the echo client is on the Wifi network, let’s start there. Let’s take a look at the promiscuous (monitor mode) trace we captured on that network.

```
tcpdump -nn -tt -r third-0-1.pcap
```

You should see some wifi-looking contents you haven't seen here before:

```
reading from file third-0-1.pcap, link-type IEEE802_11 (802.11)
0.000025 Beacon () [6.0* 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit] IBSS
0.000263 Assoc Request () [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.000279 Acknowledgment RA:00:00:00:00:00:07
0.000357 Assoc Response AID(0) :: Succesful
0.000501 Acknowledgment RA:00:00:00:00:00:0a
0.000748 Assoc Request () [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.000764 Acknowledgment RA:00:00:00:00:00:08
0.000842 Assoc Response AID(0) :: Succesful
0.000986 Acknowledgment RA:00:00:00:00:00:0a
0.001242 Assoc Request () [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.001258 Acknowledgment RA:00:00:00:00:00:09
0.001336 Assoc Response AID(0) :: Succesful
0.001480 Acknowledgment RA:00:00:00:00:00:0a
2.000112 arp who-has 10.1.3.4 (ff:ff:ff:ff:ff:ff) tell 10.1.3.3
2.000128 Acknowledgment RA:00:00:00:00:00:09
2.000206 arp who-has 10.1.3.4 (ff:ff:ff:ff:ff:ff) tell 10.1.3.3
2.000487 arp reply 10.1.3.4 is-at 00:00:00:00:00:0a
2.000659 Acknowledgment RA:00:00:00:00:00:0a
2.002169 IP 10.1.3.3.49153 > 10.1.2.4.9: UDP, length 1024
2.002185 Acknowledgment RA:00:00:00:00:00:09
2.009771 arp who-has 10.1.3.3 (ff:ff:ff:ff:ff:ff) tell 10.1.3.4
2.010029 arp reply 10.1.3.3 is-at 00:00:00:00:00:09
2.010045 Acknowledgment RA:00:00:00:00:00:09
2.010231 IP 10.1.2.4.9 > 10.1.3.3.49153: UDP, length 1024
2.011767 Acknowledgment RA:00:00:00:00:00:0a
2.500000 Beacon () [6.0* 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit] IBSS
5.000000 Beacon () [6.0* 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit] IBSS
7.500000 Beacon () [6.0* 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit] IBSS
```

You can see that the link type is now 802.11 as you would expect. You can probably understand what is going on and find the IP echo request and response packets in this trace. We leave it as an exercise to completely parse the trace dump.

Now, look at the pcap file of the right side of the point-to-point link,

```
tcpdump -nn -tt -r third-0-0.pcap
```

Again, you should see some familiar looking contents:

```
reading from file third-0-0.pcap, link-type PPP (PPP)
2.002169 IP 10.1.3.3.49153 > 10.1.2.4.9: UDP, length 1024
2.009771 IP 10.1.2.4.9 > 10.1.3.3.49153: UDP, length 1024
```

This is the echo packet going from left to right (from Wifi to CSMA) and back again across the point-to-point link.

Now, look at the pcap file of the right side of the point-to-point link,

```
tcpdump -nn -tt -r third-1-0.pcap
```

Again, you should see some familiar looking contents:

```
reading from file third-1-0.pcap, link-type PPP (PPP)
2.005855 IP 10.1.3.3.49153 > 10.1.2.4.9: UDP, length 1024
2.006084 IP 10.1.2.4.9 > 10.1.3.3.49153: UDP, length 1024
```

This is also the echo packet going from left to right (from Wifi to CSMA) and back again across the point-to-point link with slightly different timings as you might expect.

The echo server is on the CSMA network, let's look at the promiscuous trace there:

```
tcpdump -nn -tt -r third-1-1.pcap
```

You should see some familiar looking contents:

```
reading from file third-1-1.pcap, link-type EN10MB (Ethernet)
2.005855 arp who-has 10.1.2.4 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1
2.005877 arp reply 10.1.2.4 is-at 00:00:00:00:00:06
2.005877 IP 10.1.3.3.49153 > 10.1.2.4.9: UDP, length 1024
2.005980 arp who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.4
2.005980 arp reply 10.1.2.1 is-at 00:00:00:00:00:03
2.006084 IP 10.1.2.4.9 > 10.1.3.3.49153: UDP, length 1024
```

This should be easily understood. If you've forgotten, go back and look at the discussion in `second.cc`. This is the same sequence.

Now, we spent a lot of time setting up mobility models for the wireless network and so it would be a shame to finish up without even showing that the STA nodes are actually moving around during the simulation. Let's do this by hooking into the `MobilityModel` course change trace source. This is usually considered a fairly advanced topic, but let's just go for it.

As mentioned in the "Tweaking ns-3" section, the `ns-3` tracing system is divided into trace sources and trace sinks, and we provide functions to connect the two. We will use the mobility model predefined course change trace source to originate the trace events. We will need to write a trace sink to connect to that source that will display some pretty information for us. Despite its reputation as being difficult, it's really quite simple. Just before the main program of the `scratch/mythird.cc` script, add the following function:

```
void
CourseChange (std::string context, Ptr<const MobilityModel> model)
{
  Vector position = model->GetPosition ();
  NS_LOG_UNCOND (context <<
    " x = " << position.x << ", y = " << position.y);
}
```

This code just pulls the position information from the mobility model and unconditionally logs the x and y position of the node. We are going to arrange for this function to be called every time the wireless node with the echo client changes its position. We do this using the `Config::Connect` function. Add the following lines of code to the script just before the `Simulator::Run` call.

```
std::ostringstream oss;
oss <<
  "/NodeList/" << wifiStaNodes.Get (nWifi - 1)->GetId () <<
  "/$ns3::MobilityModel/CourseChange";
```

```
Config::Connect (oss.str (), MakeCallback (&CourseChange));
```

What we do here is to create a string containing the tracing namespace path of the event to which we want to connect. First, we have to figure out which node it is we want using the `GetId` method as described earlier. In the case of the default number of CSMA and wireless nodes, this turns out to be node seven and the tracing namespace path to the mobility model would look like,

```
/NodeList/7/$ns3::MobilityModel/CourseChange
```

Based on the discussion in the tracing section, you may infer that this trace path references the seventh node in the global NodeList. It specifies what is called an aggregated object of type `ns3::MobilityModel`. The dollar sign prefix implies that the MobilityModel is aggregated to node seven. The last component of the path means that we are hooking into the "CourseChange" event of that model.

We make a connection between the trace source in node seven with our trace sink by calling `Config::Connect` and passing this namespace path. Once this is done, every course change event on node seven will be hooked into our trace sink, which will in turn print out the new position.

If you now run the simulation, you will see the course changes displayed as they happen.

```
Build finished successfully (00:00:01)
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10, y = 0
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.41539, y = -0.811313
/NodeList/7/$ns3::MobilityModel/CourseChange x = 8.46199, y = -1.11303
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.52738, y = -1.46869
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.67099, y = -1.98503
/NodeList/7/$ns3::MobilityModel/CourseChange x = 5.6835, y = -2.14268
/NodeList/7/$ns3::MobilityModel/CourseChange x = 4.70932, y = -1.91689
Sent 1024 bytes to 10.1.2.4
Received 1024 bytes from 10.1.3.3
Received 1024 bytes from 10.1.2.4
/NodeList/7/$ns3::MobilityModel/CourseChange x = 5.53175, y = -2.48576
/NodeList/7/$ns3::MobilityModel/CourseChange x = 4.58021, y = -2.17821
/NodeList/7/$ns3::MobilityModel/CourseChange x = 4.18915, y = -1.25785
/NodeList/7/$ns3::MobilityModel/CourseChange x = 4.7572, y = -0.434856
/NodeList/7/$ns3::MobilityModel/CourseChange x = 4.62404, y = 0.556238
/NodeList/7/$ns3::MobilityModel/CourseChange x = 4.74127, y = 1.54934
/NodeList/7/$ns3::MobilityModel/CourseChange x = 5.73934, y = 1.48729
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.18521, y = 0.59219
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.58121, y = 1.51044
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.27897, y = 2.22677
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.42888, y = 1.70014
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.40519, y = 1.91654
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.51981, y = 1.45166
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.34588, y = 2.01523
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.81046, y = 2.90077
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.89186, y = 3.29596
```

```
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.46617, y = 2.47732
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.05492, y = 1.56579
/NodeList/7/$ns3::MobilityModel/CourseChange x = 8.00393, y = 1.25054
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.00968, y = 1.35768
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.33503, y = 2.30328
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.18682, y = 3.29223
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.96865, y = 2.66873
```

If you are feeling brave, there is a list of all trace sources in the ns-3 Doxygen which you can find in the "Modules" tab. Under the "core" section, you will find a link to "The list of all trace sources.". You may find it interesting to try and hook some of these traces yourself. Additionally in the "Modules" documentation, there is a link to "The list of all attributes.". You can set the default value of any of these `Attributes` via the command line as we have previously discussed.

We have just scratched the surface of `ns-3` in this tutorial, but we hope we have hopefully covered enough to get you started doing useful work.

– The `ns-3` development team.