ns-3 Tutorial

Release ns-3.10

ns-3 project

January 06, 2011

CONTENTS

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

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

This document is written in reStructuredText for Sphinx and is maintained in the `doc/tutorial` directory of ns-3's source code.

CHAPTER

ONE

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.

CHAPTER

TWO

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, Donahoo and Calvert.

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, Makofske and Almeroth. that covers material you may need to understand if you look at the multicast examples in the distribution.

CHAPTER

THREE

GETTING STARTED

3.1 Downloading ns-3

The *ns-3* system as a whole is a fairly complex system and has a number of dependencies on other components. Along with the systems you will most likely deal with every day (the GNU toolchain, Mercurial, you programmer editor) you will need to ensure that a number of additional libraries are present on your system before proceeding. *ns-3* provides a wiki for your reading pleasure that includes pages with many useful hints and tips. One such page is the "Installation" page, http://www.nsnam.org/wiki/index.php/Installation.

The "Prerequisites" section of this wiki page explains which packages are required to support common *ns-3* options, and also provides the commands used to install them for common Linux variants. Cygwin users will have to use the Cygwin installer (if you are a Cygwin user, you used it to install Cygwin).

You may want to take this opportunity to explore the *ns-3* wiki a bit since there really is a wealth of information there.

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 along with the prerequisites mentioned above. 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`.

The current development snapshot (unreleased) of *ns-3* may be found at http://code.nsnam.org/ns-3-dev/. 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-3.10) 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-3.10"` if you want to work with a stable release).

```
./download.py -n ns-3-dev
```

Note that the default for the `-n` option is `ns-3-dev` 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.

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 Ctrl-C to skip
 =>  bzr checkout -rrevno:640 https://launchpad.net/pybindgen pybindgen
Fetch was successful.
```

This was the download script getting the Python bindings generator for you. Note that you will need bazaar (bzr), a version control system, to download PyBindGen. 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. Note that NSC is not supported on OSX or Cygwin and works best with gcc-3.4 or gcc-4.2 or greater series.

After the download.py script completes, you should have several new directories under `~/repos/ns-3-allinone`:

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

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

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 prepackaged 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-3.10.tar.bz2
tar xjf ns-allinone-3.10.tar.bz2
```

If you change into the directory `ns-allinone-3.10` you should see a number of files:

```
build.py      ns-3.10/      pybindgen-0.15.0/    util.py
constants.py  nsc-0.5.2/    README
```

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-3.10` 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/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 -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)
Build examples and samples    : enabled
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 program `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 "./test.py -c core" script,

```
./test.py -c core
```

These tests are run in parallel by waf. You should eventually see a report saying that,

```
47 of 47 tests passed (47 passed, 0 failed, 0 crashed, 0 valgrind errors)
```

This is the important message.

You will also see output from the test runner and the output will actually look something like,

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (1.799s)
PASS: TestSuite ns3-wifi-interference
PASS: TestSuite histogram
PASS: TestSuite sample
PASS: TestSuite ipv4-address-helper
PASS: TestSuite devices-wifi
PASS: TestSuite propagation-loss-model

...
```

```
PASS: TestSuite attributes
PASS: TestSuite config
PASS: TestSuite global-value
PASS: TestSuite command-line
PASS: TestSuite basic-random-number
PASS: TestSuite object
PASS: TestSuite random-number-generators
47 of 47 tests passed (47 passed, 0 failed, 0 crashed, 0 valgrind errors)
```

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

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.

What do I do if I don't see the output?

If you don't see `waf` messages indicating that the build was completed successfully, but do not see the "Hello Simulator" output, chances are that you have switched your build mode to "optimized" in the "Building with Waf" section, but have missed the change back to "debug" mode. All of the console output used in this tutorial uses a special *ns-3* logging component that is useful for printing user messages to the console. Output from this component is automatically disabled when you compile optimized code – it is "optimized out." If you don't see the "Hello Simulator" output, type the following,

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

to tell `waf` to build the debug versions of the *ns-3* programs. You must still build the actual debug version of the code by typing,

```
./waf
```

Now, if you run the `hello-simulator` program, you should see the expected output.

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

CHAPTER

FOUR

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

Change into the `examples/tutorial` 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 `#define``s 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

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.

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.

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.

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.

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.

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.

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/tutorial/first.cc` into the `scratch` directory after changing back into the top level directory.

```
cd ..
cp examples/tutorial/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/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/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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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                                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
-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.

CHAPTER

FIVE

TWEAKING

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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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/ns-3-dev/build
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'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 attribute 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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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. Right before the call to `Simulator::Run ()`, add the following lines of code:

```
AsciiTraceHelper ascii;
pointToPoint.EnableAsciiAll (ascii.CreateFileStream ("myfirst.tr"));
```

Like in many other *ns-3* idioms, this code uses a helper object to help create ASCII traces. The second line contains two nested method calls. The "inside" method, `CreateFileStream()` uses an unnamed object idiom to create a file stream object on the stack (without an object name) and pass it down to the called method. We'll go into this more in the future, but all you have to know at this point is that you are creating an object representing a file named "myfirst.tr" and passing it into `ns-3`. You are telling `ns-3` to deal with the lifetime issues of the created object and also to deal with problems caused by a little-known (intentional) limitation of C++ ofstream objects relating to copy constructors.

The outside call, to `EnableAsciiAll()`, tells the helper 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.

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.

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 packet capture, 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.

```
pointToPoint.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.

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.

Reading output with Wireshark

If you are unfamiliar 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*.

CHAPTER

SIX

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/tutorial} 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/tutorial/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.

```
pointToPoint.EnablePcapAll ("second");
csma.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/tutorial/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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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.

```
pointToPoint.EnablePcap ("second", p2pNodes.Get (0)->GetId (), 0);
csma.EnablePcap ("second", csmaNodes.Get (nCsma)->GetId (), 0, false);
csma.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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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 Models, Attributes and Reality

This is a convenient place to make a small excursion and make an important point. It may or may not be obvious to you, but whenever one is using a simulation, it is important to understand exactly what is being modeled and what is not. It is tempting, for example, to think of the CSMA devices and channels used in the previous section as if they were real Ethernet devices; and to expect a simulation result to directly reflect what will happen in a real Ethernet. This is not the case.

A model is, by definition, an abstraction of reality. It is ultimately the responsibility of the simulation script author to determine the so-called "range of accuracy" and "domain of applicability" of the simulation as a whole, and therefore its constituent parts.

In some cases, like `Csma`, it can be fairly easy to determine what is *not* modeled. By reading the model description (`csma.h`) you can find that there is no collision detection in the CSMA model and decide on how applicable its use will be in your simulation or what caveats you may want to include with your results. In other cases, it can be quite easy to configure behaviors that might not agree with any reality you can go out and buy. It will prove worthwhile to spend some time investigating a few such instances, and how easily you can swerve outside the bounds of reality in your simulations.

As you have seen, *ns-3* provides `Attributes` which a user can easily set to change model behavior. Consider two of the `Attributes` of the `CsmaNetDevice`: `Mtu` and `EncapsulationMode`. The `Mtu` attribute indicates the Maximum Transmission Unit to the device. This is the size of the largest Protocol Data Unit (PDU) that the device can send.

The MTU defaults to 1500 bytes in the `CsmaNetDevice`. This default corresponds to a number found in RFC 894, "A Standard for the Transmission of IP Datagrams over Ethernet Networks." The number is actually derived from the maximum packet size for 10Base5 (full-spec Ethernet) networks – 1518 bytes. If you subtract the DIX encapsulation overhead for Ethernet packets (18 bytes) you will end up with a maximum possible data size (MTU) of 1500 bytes. One can also find that the `MTU` for IEEE 802.3 networks is 1492 bytes. This is because LLC/SNAP encapsulation adds an extra eight bytes of overhead to the packet. In both cases, the underlying hardware can only send 1518 bytes, but the data size is different.

In order to set the encapsulation mode, the `CsmaNetDevice` provides an `Attribute` called `EncapsulationMode` which can take on the values `Dix` or `Llc`. These correspond to Ethernet and LLC/SNAP framing respectively.

If one leaves the `Mtu` at 1500 bytes and changes the encapsulation mode to `Llc`, the result will be a network that encapsulates 1500 byte PDUs with LLC/SNAP framing resulting in packets of 1526 bytes, which would be illegal

in many networks, since they can transmit a maximum of 1518 bytes per packet. This would most likely result in a simulation that quite subtly does not reflect the reality you might be expecting.

Just to complicate the picture, there exist jumbo frames (1500 < MTU <= 9000 bytes) and super-jumbo (MTU > 9000 bytes) frames that are not officially sanctioned by IEEE but are available in some high-speed (Gigabit) networks and NICs. One could leave the encapsulation mode set to `Dix`, and set the `Mtu Attribute` on a `CsmaNetDevice` to 64000 bytes – even though an associated `CsmaChannel DataRate` was set at 10 megabits per second. This would essentially model an Ethernet switch made out of vampire-tapped 1980s-style 10Base5 networks that support super-jumbo datagrams. This is certainly not something that was ever made, nor is likely to ever be made, but it is quite easy for you to configure.

In the previous example, you used the command line to create a simulation that had 100 `Csma` nodes. You could have just as easily created a simulation with 500 nodes. If you were actually modeling that 10Base5 vampire-tap network, the maximum length of a full-spec Ethernet cable is 500 meters, with a minimum tap spacing of 2.5 meters. That means there could only be 200 taps on a real network. You could have quite easily built an illegal network in that way as well. This may or may not result in a meaningful simulation depending on what you are trying to model.

Similar situations can occur in many places in *ns-3* and in any simulator. For example, you may be able to position nodes in such a way that they occupy the same space at the same time, or you may be able to configure amplifiers or noise levels that violate the basic laws of physics.

ns-3 generally favors flexibility, and many models will allow freely setting `Attributes` without trying to enforce any arbitrary consistency or particular underlying spec.

The thing to take home from this is that *ns-3* is going to provide a super-flexible base for you to experiment with. It is up to you to understand what you are asking the system to do and to make sure that the simulations you create have some meaning and some connection with a reality defined by you.

6.3 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/tutorial` directory. This script builds on the `second.cc` script and adds a Wifi network. Go ahead and open `examples/tutorial/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::StaWifiMac",
  "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 that will be created by the helper is specified by Attribute as being of the "ns3::StaWifiMac" type. The use of NqosWifiMacHelper will ensure that the "QosSupported" Attribute for created MAC objects is set false. The combination of these two configurations means that the MAC instance next created will be a non-QoS non-AP station (STA) in an infrastructure BSS (i.e., a BSS with an AP). 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::ApWifiMac",
  "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::ApWifiMac", the latter specifying that a MAC instance configured as an AP should be created, with the helper type implying that the "QosSupported" Attribute should be set to false - disabling 802.11e/WMM-style QoS support at created APs. 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:

```
pointToPoint.EnablePcapAll ("third");
phy.EnablePcap ("third", apDevices.Get (0));
csma.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/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/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 just a sneak peek into the detailed tracing section which is coming up, but this seems a very nice place to get an example in.

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
```

CHAPTER

SEVEN

TRACING

7.1 Background

As mentioned in the Using the Tracing System section, the whole point of running an *ns-3* simulation is to generate output for study. You have two basic strategies to work with in *ns-3*: using generic pre-defined bulk output mechanisms and parsing their content to extract interesting information; or somehow developing an output mechanism that conveys exactly (and perhaps only) the information wanted.

Using pre-defined bulk output mechanisms has the advantage of not requiring any changes to *ns-3*, but it does require programming. Often, pcap or NS_LOG output messages are gathered during simulation runs and separately run through scripts that use grep, sed or awk to parse the messages and reduce and transform the data to a manageable form. Programs must be written to do the transformation, so this does not come for free. Of course, if the information of interest in does not exist in any of the pre-defined output mechanisms, this approach fails.

If you need to add some tidbit of information to the pre-defined bulk mechanisms, this can certainly be done; and if you use one of the *ns-3* mechanisms, you may get your code added as a contribution.

ns-3 provides another mechanism, called Tracing, that avoids some of the problems inherent in the bulk output mechanisms. It has several important advantages. First, you can reduce the amount of data you have to manage by only tracing the events of interest to you (for large simulations, dumping everything to disk for post-processing can create I/O bottlenecks). Second, if you use this method, you can control the format of the output directly so you avoid the postprocessing step with sed or awk script. If you desire, your output can be formatted directly into a form acceptable by gnuplot, for example. You can add hooks in the core which can then be accessed by other users, but which will produce no information unless explicitly asked to do so. For these reasons, we believe that the *ns-3* tracing system is the best way to get information out of a simulation and is also therefore one of the most important mechanisms to understand in *ns-3*.

7.1.1 Blunt Instruments

There are many ways to get information out of a program. The most straightforward way is to just directly print the information to the standard output, as in,

```
#include <iostream>
...
void
SomeFunction (void)
{
  uint32_t x = SOME_INTERESTING_VALUE;
  ...
  std::cout << "The value of x is " << x << std::endl;
  ...
}
```

Nobody is going to prevent you from going deep into the core of *ns-3* and adding print statements. This is insanely easy to do and, after all, you have complete control of your own *ns-3* branch. This will probably not turn out to be very satisfactory in the long term, though.

As the number of print statements increases in your programs, the task of dealing with the large number of outputs will become more and more complicated. Eventually, you may feel the need to control what information is being printed in some way; perhaps by turning on and off certain categories of prints, or increasing or decreasing the amount of information you want. If you continue down this path you may discover that you have re-implemented the `NS_LOG` mechanism. In order to avoid that, one of the first things you might consider is using `NS_LOG` itself.

We mentioned above that one way to get information out of *ns-3* is to parse existing NS_LOG output for interesting information. If you discover that some tidbit of information you need is not present in existing log output, you could edit the core of *ns-3* and simply add your interesting information to the output stream. Now, this is certainly better than adding your own print statements since it follows *ns-3* coding conventions and could potentially be useful to other people as a patch to the existing core.

Let's pick a random example. If you wanted to add more logging to the *ns-3* TCP socket (`tcp-socket-base.cc`) you could just add a new message down in the implementation. Notice that in TcpSocketBase::ReceivedAck() there is no log message for the no ack case. You could simply add one, changing the code from:

```
/** Process the newly received ACK */
void
TcpSocketBase::ReceivedAck (Ptr<Packet> packet, const TcpHeader& tcpHeader)
{
  NS_LOG_FUNCTION (this << tcpHeader);

  // Received ACK. Compare the ACK number against highest unacked seqno
  if (0 == (tcpHeader.GetFlags () & TcpHeader::ACK))
    { // Ignore if no ACK flag
    }
  ...
```

to add a new `NS_LOG_LOGIC` in the appropriate statement:

```
/** Process the newly received ACK */
void
TcpSocketBase::ReceivedAck (Ptr<Packet> packet, const TcpHeader& tcpHeader)
{
  NS_LOG_FUNCTION (this << tcpHeader);

  // Received ACK. Compare the ACK number against highest unacked seqno
  if (0 == (tcpHeader.GetFlags () & TcpHeader::ACK))
    { // Ignore if no ACK flag
      NS_LOG_LOGIC ("TcpSocketBase " << this << " no ACK flag");
    }
  ...
```

This may seem fairly simple and satisfying at first glance, but something to consider is that you will be writing code to add the `NS_LOG` statement and you will also have to write code (as in grep, sed or awk scripts) to parse the log output in order to isolate your information. This is because even though you have some control over what is output by the logging system, you only have control down to the log component level.

If you are adding code to an existing module, you will also have to live with the output that every other developer has found interesting. You may find that in order to get the small amount of information you need, you may have to wade through huge amounts of extraneous messages that are of no interest to you. You may be forced to save huge log files to disk and process them down to a few lines whenever you want to do anything.

Since there are no guarantees in *ns-3* about the stability of `NS_LOG` output, you may also discover that pieces of log output on which you depend disappear or change between releases. If you depend on the structure of the output, you may find other messages being added or deleted which may affect your parsing code.

For these reasons, we consider prints to `std::cout` and NS_LOG messages to be quick and dirty ways to get more information out of *ns-3*.

It is desirable to have a stable facility using stable APIs that allow one to reach into the core system and only get the information required. It is desirable to be able to do this without having to change and recompile the core system. Even better would be a system that notified the user when an item of interest changed or an interesting event happened so the user doesn't have to actively poke around in the system looking for things.

The *ns-3* tracing system is designed to work along those lines and is well-integrated with the Attribute and Config subsystems allowing for relatively simple use scenarios.

7.2 Overview

The ns-3 tracing system is built on the concepts of independent tracing sources and tracing sinks; along with 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. A trace source might also indicate when an interesting state change happens in a model. For example, the congestion window of a TCP model is a prime candidate for a trace source.

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 source. The entities that consume trace information are called trace sinks. Trace sources are generators of events and trace sinks are consumers. This explicit division allows for large numbers of trace sources to be scattered around the system in places which model authors believe might be useful.

There can be zero or more consumers of trace events generated by a trace source. One can think of a trace source as a kind of point-to-multipoint information link. Your code looking for trace events from a particular piece of core code could happily coexist with other code doing something entirely different from the same information.

Unless a user connects a trace sink to one of these sources, nothing is output. By using the tracing system, both you and other people at the same trace source are getting exactly what they want and only what they want out of the system. Neither of you are impacting any other user by changing what information is output by the system. If you happen to add a trace source, your work as a good open-source citizen may allow other users to provide new utilities that are perhaps very useful overall, without making any changes to the *ns-3* core.

7.2.1 A Simple Low-Level Example

Let's take a few minutes and walk through a simple tracing example. We are going to need a little background on Callbacks to understand what is happening in the example, so we have to take a small detour right away.

Callbacks

The goal of the Callback system in *ns-3* is to allow one piece of code to call a function (or method in C++) without any specific inter-module dependency. This ultimately means you need some kind of indirection – you treat the address of the called function as a variable. This variable is called a pointer-to-function variable. The relationship between function and pointer-to-function pointer is really no different that that of object and pointer-to-object.

In C the canonical example of a pointer-to-function is a pointer-to-function-returning-integer (PFI). For a PFI taking one int parameter, this could be declared like,

```
int (*pfi)(int arg) = 0;
```

What you get from this is a variable named simply "pfi" that is initialized to the value 0. If you want to initialize this pointer to something meaningful, you have to have a function with a matching signature. In this case, you could provide a function that looks like,

```
int MyFunction (int arg) {}
```

If you have this target, you can initialize the variable to point to your function:

```
pfi = MyFunction;
```

You can then call MyFunction indirectly using the more suggestive form of the call,

```
int result = (*pfi) (1234);
```

This is suggestive since it looks like you are dereferencing the function pointer just like you would dereference any pointer. Typically, however, people take advantage of the fact that the compiler knows what is going on and will just use a shorter form,

```
int result = pfi (1234);
```

This looks like you are calling a function named "pfi," but the compiler is smart enough to know to call through the variable `pfi` indirectly to the function `MyFunction`.

Conceptually, this is almost exactly how the tracing system will work. Basically, a trace source *is* a callback. When a trace sink expresses interest in receiving trace events, it adds a Callback to a list of Callbacks internally held by the trace source. When an interesting event happens, the trace source invokes its `operator()` providing zero or more parameters. The `operator()` eventually wanders down into the system and does something remarkably like the indirect call you just saw. It provides zero or more parameters (the call to "pfi" above passed one parameter to the target function `MyFunction`.

The important difference that the tracing system adds is that for each trace source there is an internal list of Callbacks. Instead of just making one indirect call, a trace source may invoke any number of Callbacks. When a trace sink expresses interest in notifications from a trace source, it basically just arranges to add its own function to the callback list.

If you are interested in more details about how this is actually arranged in *ns-3*, feel free to peruse the Callback section of the manual.

Example Code

We have provided some code to implement what is really the simplest example of tracing that can be assembled. You can find this code in the tutorial directory as `fourth.cc`. Let's walk through it.

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

```
#include "ns3/object.h"
#include "ns3/uinteger.h"
#include "ns3/traced-value.h"
#include "ns3/trace-source-accessor.h"

#include <iostream>

using namespace ns3;
```

Most of this code should be quite familiar to you. As mentioned above, the trace system makes heavy use of the Object and Attribute systems, so you will need to include them. The first two includes above bring in the declarations for those systems explicitly. You could use the core module header, but this illustrates how simple this all really is.

The file, `traced-value.h` brings in the required declarations for tracing of data that obeys value semantics. In general, value semantics just means that you can pass the object around, not an address. In order to use value semantics at all you have to have an object with an associated copy constructor and assignment operator available. We extend the requirements to talk about the set of operators that are pre-defined for plain-old-data (POD) types. Operator=, operator++, operator—, operator+, operator==, etc.

What this all really means is that you will be able to trace changes to a C++ object made using those operators.

Since the tracing system is integrated with Attributes, and Attributes work with Objects, there must be an *ns-3* `Object` for the trace source to live in. The next code snippet declares and defines a simple Object we can work with.

```
class MyObject : public Object
{
public:
  static TypeId GetTypeId (void)
  {
    static TypeId tid = TypeId ("MyObject")
      .SetParent (Object::GetTypeId ())
      .AddConstructor<MyObject> ()
      .AddTraceSource ("MyInteger",
                       "An integer value to trace.",
                       MakeTraceSourceAccessor (&MyObject::m_myInt))
      ;
    return tid;
  }

  MyObject () {}
  TracedValue<int32_t> m_myInt;
};
```

The two important lines of code, above, with respect to tracing are the `.AddTraceSource` and the `TracedValue` declaration of `m_myInt`.

The `.AddTraceSource` provides the "hooks" used for connecting the trace source to the outside world through the config system. The `TracedValue` declaration provides the infrastructure that overloads the operators mentioned above and drives the callback process.

```
void
IntTrace (int32_t oldValue, int32_t newValue)
{
  std::cout << "Traced " << oldValue << " to " << newValue << std::endl;
}
```

This is the definition of the trace sink. It corresponds directly to a callback function. Once it is connected, this function will be called whenever one of the overloaded operators of the `TracedValue` is executed.

We have now seen the trace source and the trace sink. What remains is code to connect the source to the sink.

```
int
main (int argc, char *argv[])
{
  Ptr<MyObject> myObject = CreateObject<MyObject> ();
  myObject->TraceConnectWithoutContext ("MyInteger", MakeCallback(&IntTrace));

  myObject->m_myInt = 1234;
}
```

Here we first create the Object in which the trace source lives.

The next step, the `TraceConnectWithoutContext`, forms the connection between the trace source and the trace sink. Notice the `MakeCallback` template function. This function does the magic required to create the underlying *ns-3* Callback object and associate it with the function `IntTrace`. TraceConnect makes the association between your provided function and the overloaded `operator()` in the traced variable referred to by the "MyInteger" Attribute. After this association is made, the trace source will "fire" your provided callback function.

The code to make all of this happen is, of course, non-trivial, but the essence is that you are arranging for something that looks just like the `pfi()` example above to be called by the trace source. The declaration of the `TracedValue<int32_t> m_myInt;` in the Object itself performs the magic needed to provide the overloaded operators (++, —, etc.) that will use the `operator()` to actually invoke the Callback with the desired parameters. The `.AddTraceSource` performs the magic to connect the Callback to the Config system, and `TraceConnectWithoutContext` performs the magic to connect your function to the trace source, which is specified by Attribute name.

Let's ignore the bit about context for now.

Finally, the line,

```
myObject->m_myInt = 1234;
```

should be interpreted as an invocation of `operator=` on the member variable `m_myInt` with the integer `1234` passed as a parameter.

It turns out that this operator is defined (by `TracedValue`) to execute a callback that returns void and takes two integer values as parameters — an old value and a new value for the integer in question. That is exactly the function signature for the callback function we provided — `IntTrace`.

To summarize, a trace source is, in essence, a variable that holds a list of callbacks. A trace sink is a function used as the target of a callback. The Attribute and object type information systems are used to provide a way to connect trace sources to trace sinks. The act of "hitting" a trace source is executing an operator on the trace source which fires callbacks. This results in the trace sink callbacks registering interest in the source being called with the parameters provided by the source.

If you now build and run this example,

```
./waf --run fourth
```

you will see the output from the `IntTrace` function execute as soon as the trace source is hit:

```
Traced 0 to 1234
```

When we executed the code, `myObject->m_myInt = 1234;`, the trace source fired and automatically provided the before and after values to the trace sink. The function `IntTrace` then printed this to the standard output. No problem.

7.2.2 Using the Config Subsystem to Connect to Trace Sources

The `TraceConnectWithoutContext` call shown above in the simple example is actually very rarely used in the system. More typically, the `Config` subsystem is used to allow selecting a trace source in the system using what is called a *config path*. We saw an example of this in the previous section where we hooked the "CourseChange" event when we were playing with `third.cc`.

Recall that we defined a trace sink to print course change information from the mobility models of our simulation. It should now be a lot more clear to you what this function is doing.

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

When we connected the "CourseChange" trace source to the above trace sink, we used what is called a "Config Path" to specify the source when we arranged a connection between the pre-defined trace source and the new trace sink:

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

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

Let's try and make some sense of what is sometimes considered relatively mysterious code. For the purposes of discussion, assume that the node number returned by the `GetId()` is "7". In this case, the path above turns out to be,

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

The last segment of a config path must be an `Attribute` of an `Object`. In fact, if you had a pointer to the `Object` that has the "CourseChange" `Attribute` handy, you could write this just like we did in the previous example. You know by now that we typically store pointers to our nodes in a NodeContainer. In the `third.cc` example, the Nodes of interest are stored in the `wifiStaNodes` NodeContainer. In fact, while putting the path together, we used this container to get a Ptr<Node> which we used to call GetId() on. We could have used this Ptr<Node> directly to call a connect method directly:

```
Ptr<Object> theObject = wifiStaNodes.Get (nWifi - 1);
theObject->TraceConnectWithoutContext ("CourseChange", MakeCallback (&CourseChange));
```

In the `third.cc` example, we actually want an additional "context" to be delivered along with the Callback parameters (which will be explained below) so we could actually use the following equivalent code,

```
Ptr<Object> theObject = wifiStaNodes.Get (nWifi - 1);
theObject->TraceConnect ("CourseChange", MakeCallback (&CourseChange));
```

It turns out that the internal code for `Config::ConnectWithoutContext` and `Config::Connect` actually do find a Ptr<Object> and call the appropriate TraceConnect method at the lowest level.

The `Config` functions take a path that represents a chain of `Object` pointers. Each segment of a path corresponds to an Object Attribute. The last segment is the Attribute of interest, and prior segments must be typed to contain or find Objects. The `Config` code parses and "walks" this path until it gets to the final segment of the path. It then interprets the last segment as an `Attribute` on the last Object it found while walking the path. The `Config` functions then call the appropriate `TraceConnect` or `TraceConnectWithoutContext` method on the final Object. Let's see what happens in a bit more detail when the above path is walked.

The leading "/" character in the path refers to a so-called namespace. One of the predefined namespaces in the config system is "NodeList" which is a list of all of the nodes in the simulation. Items in the list are referred to by indices into the list, so "/NodeList/7" refers to the eighth node in the list of nodes created during the simulation. This reference is actually a `Ptr<Node>` and so is a subclass of an `ns3::Object`.

As described in the Object Model section of the *ns-3* manual, we support Object Aggregation. This allows us to form an association between different Objects without any programming. Each Object in an Aggregation can be reached from the other Objects.

The next path segment being walked begins with the "$" character. This indicates to the config system that a `GetObject` call should be made looking for the type that follows. It turns out that the MobilityHelper used in `third.cc` arranges to Aggregate, or associate, a mobility model to each of the wireless Nodes. When you add the "$" you are asking for another Object that has presumably been previously aggregated. You can think of this as switching pointers from the original Ptr<Node> as specified by "/NodeList/7" to its associated mobility model — which is of type "$ns3::MobilityModel". If you are familiar with `GetObject`, we have asked the system to do the following:

```
Ptr<MobilityModel> mobilityModel = node->GetObject<MobilityModel> ()
```

We are now at the last Object in the path, so we turn our attention to the Attributes of that Object. The `MobilityModel` class defines an Attribute called "CourseChange". You can see this by looking at the source code in `src/mobility/mobility-model.cc` and searching for "CourseChange" in your favorite editor. You should find,

```
.AddTraceSource ("CourseChange",
                 "The value of the position and/or velocity vector changed",
                 MakeTraceSourceAccessor (&MobilityModel::m_courseChangeTrace))
```

which should look very familiar at this point.

If you look for the corresponding declaration of the underlying traced variable in `mobility-model.h` you will find

```
TracedCallback<Ptr<const MobilityModel> > m_courseChangeTrace;
```

The type declaration `TracedCallback` identifies `m_courseChangeTrace` as a special list of Callbacks that can be hooked using the Config functions described above.

The `MobilityModel` class is designed to be a base class providing a common interface for all of the specific subclasses. If you search down to the end of the file, you will see a method defined called `NotifyCourseChange()`:

```
void
MobilityModel::NotifyCourseChange (void) const
{
  m_courseChangeTrace(this);
}
```

Derived classes will call into this method whenever they do a course change to support tracing. This method invokes `operator()` on the underlying `m_courseChangeTrace`, which will, in turn, invoke all of the registered Callbacks, calling all of the trace sinks that have registered interest in the trace source by calling a Config function.

So, in the `third.cc` example we looked at, whenever a course change is made in one of the `RandomWalk2dMobilityModel` instances installed, there will be a `NotifyCourseChange()` call which calls up into the `MobilityModel` base class. As seen above, this invokes `operator()` on `m_courseChangeTrace`, which in turn, calls any registered trace sinks. In the example, the only code registering an interest was the code that provided the config path. Therefore, the `CourseChange` function that was hooked from Node number seven will be the only Callback called.

The final piece of the puzzle is the "context". Recall that we saw an output looking something like the following from `third.cc`:

```
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.27897, y = 2.22677
```

The first part of the output is the context. It is simply the path through which the config code located the trace source. In the case we have been looking at there can be any number of trace sources in the system corresponding to any number of nodes with mobility models. There needs to be some way to identify which trace source is actually the one that fired the Callback. An easy way is to request a trace context when you `Config::Connect`.

7.2.3 How to Find and Connect Trace Sources, and Discover Callback Signatures

The first question that inevitably comes up for new users of the Tracing system is, "okay, I know that there must be trace sources in the simulation core, but how do I find out what trace sources are available to me"?

The second question is, "okay, I found a trace source, how do I figure out the config path to use when I connect to it"?

The third question is, "okay, I found a trace source, how do I figure out what the return type and formal arguments of my callback function need to be"?

The fourth question is, "okay, I typed that all in and got this incredibly bizarre error message, what in the world does it mean"?

7.2.4 What Trace Sources are Available?

The answer to this question is found in the *ns-3* Doxygen. Go to the *ns-3* web site "here" and select the "Doxygen (stable)" link "Documentation" on the navigation bar to the left side of the page. Expand the "Modules" book in the NS-3 documentation tree a the upper left by clicking the "+" box. Now, expand the "Core" book in the tree by clicking its "+" box. You should now see three extremely useful links:

- The list of all trace sources
- The list of all attributes
- The list of all global values

The list of interest to us here is "the list of all trace sources". Go ahead and select that link. You will see, perhaps not too surprisingly, a list of all of the trace sources available in the *ns-3* core.

As an example, scroll down to `ns3::MobilityModel`. You will find an entry for

```
CourseChange: The value of the position and/or velocity vector changed
```

You should recognize this as the trace source we used in the `third.cc` example. Perusing this list will be helpful.

7.2.5 What String do I use to Connect?

The easiest way to do this is to grep around in the *ns-3* codebase for someone who has already figured it out, You should always try to copy someone else's working code before you start to write your own. Try something like:

```
find . -name '*.cc' | xargs grep CourseChange | grep Connect
```

and you may find your answer along with working code. For example, in this case, `./ns-3-dev/examples/wireless/mixed-wireless.cc` has something just waiting for you to use:

```
Config::Connect ("/NodeList/*/$ns3::MobilityModel/CourseChange",
  MakeCallback (&CourseChangeCallback));
```

If you cannot find any examples in the distribution, you can find this out from the *ns-3* Doxygen. It will probably be simplest just to walk through the "CourseChanged" example.

Let's assume that you have just found the "CourseChanged" trace source in "The list of all trace sources" and you want to figure out how to connect to it. You know that you are using (again, from the `third.cc` example) an `ns3::RandomWalk2dMobilityModel`. So open the "Class List" book in the NS-3 documentation tree by clicking its "+" box. You will now see a list of all of the classes in *ns-3*. Scroll down until you see the entry for `ns3::RandomWalk2dMobilityModel` and follow that link. You should now be looking at the "ns3::RandomWalk2dMobilityModel Class Reference".

If you now scroll down to the "Member Function Documentation" section, you will see documentation for the `GetTypeId` function. You constructed one of these in the simple tracing example above:

```
static TypeId GetTypeId (void)
{
  static TypeId tid = TypeId ("MyObject")
    .SetParent (Object::GetTypeId ())
    .AddConstructor<MyObject> ()
    .AddTraceSource ("MyInteger",
                     "An integer value to trace.",
                     MakeTraceSourceAccessor (&MyObject::m_myInt))
    ;
  return tid;
}
```

As mentioned above, this is the bit of code that connected the Config and Attribute systems to the underlying trace source. This is also the place where you should start looking for information about the way to connect.

You are looking at the same information for the RandomWalk2dMobilityModel; and the information you want is now right there in front of you in the Doxygen:

```
This object is accessible through the following paths with Config::Set and Config::Connect:

/NodeList/[i]/$ns3::MobilityModel/$ns3::RandomWalk2dMobilityModel
```

The documentation tells you how to get to the `RandomWalk2dMobilityModel` Object. Compare the string above with the string we actually used in the example code:

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

The difference is due to the fact that two `GetObject` calls are implied in the string found in the documentation. The first, for `$ns3::MobilityModel` will query the aggregation for the base class. The second implied `GetObject` call, for `$ns3::RandomWalk2dMobilityModel`, is used to "cast" the base class to the concrete implementation class. The documentation shows both of these operations for you. It turns out that the actual Attribute you are going to be looking for is found in the base class as we have seen.

Look further down in the `GetTypeId` doxygen. You will find,

```
No TraceSources defined for this type.
TraceSources defined in parent class ns3::MobilityModel:

CourseChange: The value of the position and/or velocity vector changed
Reimplemented from ns3::MobilityModel
```

This is exactly what you need to know. The trace source of interest is found in `ns3::MobilityModel` (which you knew anyway). The interesting thing this bit of Doxygen tells you is that you don't need that extra cast in the config path above to get to the concrete class, since the trace source is actually in the base class. Therefore the additional `GetObject` is not required and you simply use the path:

```
/NodeList/[i]/$ns3::MobilityModel
```

which perfectly matches the example path:

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

7.2.6 What Return Value and Formal Arguments?

The easiest way to do this is to grep around in the *ns-3* codebase for someone who has already figured it out, You should always try to copy someone else's working code. Try something like:

```
find . -name '*.cc' | xargs grep CourseChange | grep Connect
```

and you may find your answer along with working code. For example, in this case, `./ns-3-dev/examples/wireless/mixed-wireless.cc` has something just waiting for you to use. You will find

```
Config::Connect ("/NodeList/*/$ns3::MobilityModel/CourseChange",
  MakeCallback (&CourseChangeCallback));
```

as a result of your grep. The `MakeCallback` should indicate to you that there is a callback function there which you can use. Sure enough, there is:

```
static void
CourseChangeCallback (std::string path, Ptr<const MobilityModel> model)
{
  ...
}
```

Take my Word for It

If there are no examples to work from, this can be, well, challenging to actually figure out from the source code.

Before embarking on a walkthrough of the code, I'll be kind and just tell you a simple way to figure this out: The return value of your callback will always be void. The formal parameter list for a `TracedCallback` can be found from the template parameter list in the declaration. Recall that for our current example, this is in `mobility-model.h`, where we have previously found:

```
TracedCallback<Ptr<const MobilityModel> > m_courseChangeTrace;
```

There is a one-to-one correspondence between the template parameter list in the declaration and the formal arguments of the callback function. Here, there is one template parameter, which is a `Ptr<const MobilityModel>`. This tells you that you need a function that returns void and takes a a `Ptr<const MobilityModel>`. For example,

```
void
CourseChangeCallback (Ptr<const MobilityModel> model)
{
  ...
}
```

That's all you need if you want to `Config::ConnectWithoutContext`. If you want a context, you need to `Config::Connect` and use a Callback function that takes a string context, then the required argument.

```
void
CourseChangeCallback (std::string path, Ptr<const MobilityModel> model)
{
```

```
  ...
}
```

If you want to ensure that your `CourseChangeCallback` is only visible in your local file, you can add the keyword `static` and come up with:

```
static void
CourseChangeCallback (std::string path, Ptr<const MobilityModel> model)
{
  ...
}
```

which is exactly what we used in the `third.cc` example.

The Hard Way

This section is entirely optional. It is going to be a bumpy ride, especially for those unfamiliar with the details of templates. However, if you get through this, you will have a very good handle on a lot of the *ns-3* low level idioms.

So, again, let's figure out what signature of callback function is required for the "CourseChange" Attribute. This is going to be painful, but you only need to do this once. After you get through this, you will be able to just look at a `TracedCallback` and understand it.

The first thing we need to look at is the declaration of the trace source. Recall that this is in `mobility-model.h`, where we have previously found:

```
TracedCallback<Ptr<const MobilityModel> > m_courseChangeTrace;
```

This declaration is for a template. The template parameter is inside the angle-brackets, so we are really interested in finding out what that `TracedCallback<>` is. If you have absolutely no idea where this might be found, grep is your friend.

We are probably going to be interested in some kind of declaration in the *ns-3* source, so first change into the `src` directory. Then, we know this declaration is going to have to be in some kind of header file, so just grep for it using:

```
find . -name '*.h' | xargs grep TracedCallback
```

You'll see 124 lines fly by (I piped this through wc to see how bad it was). Although that may seem like it, that's not really a lot. Just pipe the output through more and start scanning through it. On the first page, you will see some very suspiciously template-looking stuff.

```
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::TracedCallback ()
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::ConnectWithoutContext (c ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::Connect (const CallbackB ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::DisconnectWithoutContext ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::Disconnect (const Callba ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (void) const ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1) const ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1, T2 a2 ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1, T2 a2 ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1, T2 a2 ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1, T2 a2 ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1, T2 a2 ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1, T2 a2 ...
```

It turns out that all of this comes from the header file `traced-callback.h` which sounds very promising. You can then take a look at `mobility-model.h` and see that there is a line which confirms this hunch:

```
#include "ns3/traced-callback.h"
```

Of course, you could have gone at this from the other direction and started by looking at the includes in `mobility-model.h` and noticing the include of `traced-callback.h` and inferring that this must be the file you want.

In either case, the next step is to take a look at `src/core/traced-callback.h` in your favorite editor to see what is happening.

You will see a comment at the top of the file that should be comforting:

```
An ns3::TracedCallback has almost exactly the same API as a normal ns3::Callback but
instead of forwarding calls to a single function (as an ns3::Callback normally does),
it forwards calls to a chain of ns3::Callback.
```

This should sound very familiar and let you know you are on the right track.

Just after this comment, you will find,

```
template<typename T1 = empty, typename T2 = empty,
         typename T3 = empty, typename T4 = empty,
         typename T5 = empty, typename T6 = empty,
         typename T7 = empty, typename T8 = empty>
class TracedCallback
{
  ...
```

This tells you that TracedCallback is a templated class. It has eight possible type parameters with default values. Go back and compare this with the declaration you are trying to understand:

```
TracedCallback<Ptr<const MobilityModel> > m_courseChangeTrace;
```

The `typename T1` in the templated class declaration corresponds to the `Ptr<const MobilityModel>` in the declaration above. All of the other type parameters are left as defaults. Looking at the constructor really doesn't tell you much. The one place where you have seen a connection made between your Callback function and the tracing system is in the `Connect` and `ConnectWithoutContext` functions. If you scroll down, you will see a `ConnectWithoutContext` method here:

```
template<typename T1, typename T2,
         typename T3, typename T4,
         typename T5, typename T6,
         typename T7, typename T8>
void
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::ConnectWithoutContext ...
{
  Callback<void,T1,T2,T3,T4,T5,T6,T7,T8> cb;
  cb.Assign (callback);
  m_callbackList.push_back (cb);
}
```

You are now in the belly of the beast. When the template is instantiated for the declaration above, the compiler will replace `T1` with `Ptr<const MobilityModel>`.

```
void
TracedCallback<Ptr<const MobilityModel>::ConnectWithoutContext ... cb
{
  Callback<void, Ptr<const MobilityModel> > cb;
  cb.Assign (callback);
  m_callbackList.push_back (cb);
}
```

You can now see the implementation of everything we've been talking about. The code creates a Callback of the right type and assigns your function to it. This is the equivalent of the `pfi = MyFunction` we discussed at the start of this section. The code then adds the Callback to the list of Callbacks for this source. The only thing left is to look at the definition of Callback. Using the same grep trick as we used to find `TracedCallback`, you will be able to find that the file `./core/callback.h` is the one we need to look at.

If you look down through the file, you will see a lot of probably almost incomprehensible template code. You will eventually come to some Doxygen for the Callback template class, though. Fortunately, there is some English:

```
This class template implements the Functor Design Pattern.
It is used to declare the type of a Callback:
 - the first non-optional template argument represents
   the return type of the callback.
 - the second optional template argument represents
   the type of the first argument to the callback.
 - the third optional template argument represents
   the type of the second argument to the callback.
 - the fourth optional template argument represents
   the type of the third argument to the callback.
 - the fifth optional template argument represents
   the type of the fourth argument to the callback.
 - the sixth optional template argument represents
   the type of the fifth argument to the callback.
```

We are trying to figure out what the

```
Callback<void, Ptr<const MobilityModel> > cb;
```

declaration means. Now we are in a position to understand that the first (non-optional) parameter, `void`, represents the return type of the Callback. The second (non-optional) parameter, `Ptr<const MobilityModel>` represents the first argument to the callback.

The Callback in question is your function to receive the trace events. From this you can infer that you need a function that returns `void` and takes a `Ptr<const MobilityModel>`. For example,

```
void
CourseChangeCallback (Ptr<const MobilityModel> model)
{
  ...
}
```

That's all you need if you want to `Config::ConnectWithoutContext`. If you want a context, you need to `Config::Connect` and use a Callback function that takes a string context. This is because the `Connect` function will provide the context for you. You'll need:

```
void
CourseChangeCallback (std::string path, Ptr<const MobilityModel> model)
{
  ...
}
```

If you want to ensure that your `CourseChangeCallback` is only visible in your local file, you can add the keyword `static` and come up with:

```
static void
CourseChangeCallback (std::string path, Ptr<const MobilityModel> model)
{
  ...
}
```

which is exactly what we used in the `third.cc` example. Perhaps you should now go back and reread the previous section (Take My Word for It).

If you are interested in more details regarding the implementation of Callbacks, feel free to take a look at the *ns-3* manual. They are one of the most frequently used constructs in the low-level parts of *ns-3*. It is, in my opinion, a quite elegant thing.

7.2.7 What About TracedValue?

Earlier in this section, we presented a simple piece of code that used a `TracedValue<int32_t>` to demonstrate the basics of the tracing code. We just glossed over the way to find the return type and formal arguments for the `TracedValue`. Rather than go through the whole exercise, we will just point you at the correct file, `src/core/traced-value.h` and to the important piece of code:

```
template <typename T>
class TracedValue
{
public:
  ...
  void Set (const T &v) {
    if (m_v != v)
      {
      m_cb (m_v, v);
      m_v = v;
      }
  }
  ...
private:
  T m_v;
  TracedCallback<T,T> m_cb;
};
```

Here you see that the `TracedValue` is templated, of course. In the simple example case at the start of the section, the typename is int32_t. This means that the member variable being traced (`m_v` in the private section of the class) will be an `int32_t m_v`. The `Set` method will take a `const int32_t &v` as a parameter. You should now be able to understand that the `Set` code will fire the `m_cb` callback with two parameters: the first being the current value of the `TracedValue`; and the second being the new value being set.

The callback, `m_cb` is declared as a `TracedCallback<T, T>` which will correspond to a `TracedCallback<int32_t, int32_t>` when the class is instantiated.

Recall that the callback target of a TracedCallback always returns `void`. Further recall that there is a one-to-one correspondence between the template parameter list in the declaration and the formal arguments of the callback function. Therefore the callback will need to have a function signature that looks like:

```
void
MyCallback (int32_t oldValue, int32_t newValue)
{
  ...
}
```

It probably won't surprise you that this is exactly what we provided in that simple example we covered so long ago:

```
void
IntTrace (int32_t oldValue, int32_t newValue)
{
  std::cout << "Traced " << oldValue << " to " << newValue << std::endl;
}
```

7.3 A Real Example

Let's do an example taken from one of the best-known books on TCP around. "TCP/IP Illustrated, Volume 1: The Protocols," by W. Richard Stevens is a classic. I just flipped the book open and ran across a nice plot of both the congestion window and sequence numbers versus time on page 366. Stevens calls this, "Figure 21.10. Value of cwnd and send sequence number while data is being transmitted." Let's just recreate the cwnd part of that plot in *ns-3* using the tracing system and `gnuplot`.

7.3.1 Are There Trace Sources Available?

The first thing to think about is how we want to get the data out. What is it that we need to trace? The first thing to do is to consult "The list of all trace sources" to see what we have to work with. Recall that this is found in the *ns-3* Doxygen in the "Core" Module section. If you scroll through the list, you will eventually find:

```
ns3::TcpNewReno
CongestionWindow: The TCP connection's congestion window
```

It turns out that the *ns-3* TCP implementation lives (mostly) in the file `src/internet-stack/tcp-socket-base.cc` while congestion control variants are in files such as `src/internet-stack/tcp-newreno.cc`. If you don't know this a priori, you can use the recursive grep trick:

```
find . -name '*.cc' | xargs grep -i tcp
```

You will find page after page of instances of tcp pointing you to that file.

If you open `src/internet-stack/tcp-newreno.cc` in your favorite editor, you will see right up at the top of the file, the following declarations:

```
TypeId
TcpNewReno::GetTypeId ()
{
  static TypeId tid = TypeId("ns3::TcpNewReno")
    .SetParent<TcpSocketBase> ()
    .AddConstructor<TcpNewReno> ()
    .AddTraceSource ("CongestionWindow",
                     "The TCP connection's congestion window",
                     MakeTraceSourceAccessor (&TcpNewReno::m_cWnd))
    ;
  return tid;
}
```

This should tell you to look for the declaration of `m_cWnd` in the header file `src/internet-stack/tcp-newreno.h`. If you open this file in your favorite editor, you will find:

```
TracedValue<uint32_t> m_cWnd; //Congestion window
```

You should now understand this code completely. If we have a pointer to the `TcpNewReno`, we can `TraceConnect` to the "CongestionWindow" trace source if we provide an appropriate callback target. This is the same kind of trace source that we saw in the simple example at the start of this section, except that we are talking about `uint32_t` instead of `int32_t`.

We now know that we need to provide a callback that returns void and takes two `uint32_t` parameters, the first being the old value and the second being the new value:

```
void
CwndTrace (uint32_t oldValue, uint32_t newValue)
```

```
{
  ...
}
```

7.3.2 What Script to Use?

It's always best to try and find working code laying around that you can modify, rather than starting from scratch. So the first order of business now is to find some code that already hooks the "CongestionWindow" trace source and see if we can modify it. As usual, grep is your friend:

```
find . -name '*.cc' | xargs grep CongestionWindow
```

This will point out a couple of promising candidates: `examples/tcp/tcp-large-transfer.cc` and `src/test/ns3tcp/ns3tcp-cwnd-test-suite.cc`.

We haven't visited any of the test code yet, so let's take a look there. You will typically find that test code is fairly minimal, so this is probably a very good bet. Open `src/test/ns3tcp/ns3tcp-cwnd-test-suite.cc` in your favorite editor and search for "CongestionWindow". You will find,

```
ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow",
  MakeCallback (&Ns3TcpCwndTestCase1::CwndChange, this));
```

This should look very familiar to you. We mentioned above that if we had a pointer to the `TcpNewReno`, we could `TraceConnect` to the "CongestionWindow" trace source. That's exactly what we have here; so it turns out that this line of code does exactly what we want. Let's go ahead and extract the code we need from this function (`Ns3TcpCwndTestCase1::DoRun (void)`). If you look at this function, you will find that it looks just like an *ns-3* script. It turns out that is exactly what it is. It is a script run by the test framework, so we can just pull it out and wrap it in `main` instead of in `DoRun`. Rather than walk through this, step, by step, we have provided the file that results from porting this test back to a native *ns-3* script – `examples/tutorial/fifth.cc`.

7.3.3 A Common Problem and Solution

The `fifth.cc` example demonstrates an extremely important rule that you must understand before using any kind of `Attribute`: you must ensure that the target of a `Config` command exists before trying to use it. This is no different than saying an object must be instantiated before trying to call it. Although this may seem obvious when stated this way, it does trip up many people trying to use the system for the first time.

Let's return to basics for a moment. There are three basic time periods that exist in any *ns-3* script. The first time period is sometimes called "Configuration Time" or "Setup Time," and is in force during the period when the `main` function of your script is running, but before `Simulator::Run` is called. The second time period is sometimes called "Simulation Time" and is in force during the time period when `Simulator::Run` is actively executing its events. After it completes executing the simulation, `Simulator::Run` will return control back to the `main` function. When this happens, the script enters what can be called "Teardown Time," which is when the structures and objects created during setup and taken apart and released.

Perhaps the most common mistake made in trying to use the tracing system is assuming that entities constructed dynamically during simulation time are available during configuration time. In particular, an *ns-3* `Socket` is a dynamic object often created by `Applications` to communicate between `Nodes`. An *ns-3* `Application` always has a "Start Time" and a "Stop Time" associated with it. In the vast majority of cases, an `Application` will not attempt to create a dynamic object until its `StartApplication` method is called at some "Start Time". This is to ensure that the simulation is completely configured before the app tries to do anything (what would happen if it tried to connect to a node that didn't exist yet during configuration time). The answer to this issue is to 1) create a simulator event that is run after the dynamic object is created and hook the trace when that event is executed; or 2) create the dynamic object at configuration time, hook it then, and give the object to the system to use during simulation time. We took

the second approach in the `fifth.cc` example. This decision required us to create the `MyApp Application`, the entire purpose of which is to take a `Socket` as a parameter.

7.3.4 A fifth.cc Walkthrough

Now, let's take a look at the example program we constructed by dissecting the congestion window test. Open `examples/tutorial/fifth.cc` in your favorite editor. You should see some familiar looking code:

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * 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, Include., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <fstream>
#include "ns3/core-module.h"
#include "ns3/common-module.h"
#include "ns3/simulator-module.h"
#include "ns3/node-module.h"
#include "ns3/helper-module.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("FifthScriptExample");
```

This has all been covered, so we won't rehash it. The next lines of source are the network illustration and a comment addressing the problem described above with `Socket`.

```
// ============================================================================
//
//         node 0                  node 1
//   +----------------+    +----------------+
//   |    ns-3 TCP    |    |    ns-3 TCP    |
//   +----------------+    +----------------+
//   |    10.1.1.1    |    |    10.1.1.2    |
//   +----------------+    +----------------+
//   | point-to-point |    | point-to-point |
//   +----------------+    +----------------+
//           |                      |
//           +----------------------+
//                5 Mbps, 2 ms
//
//
// We want to look at changes in the ns-3 TCP congestion window.  We need
// to crank up a flow and hook the CongestionWindow attribute on the socket
// of the sender.  Normally one would use an on-off application to generate a
// flow, but this has a couple of problems.  First, the socket of the on-off
// application is not created until Application Start time, so we wouldn't be
```

```
// able to hook the socket (now) at configuration time.  Second, even if we
// could arrange a call after start time, the socket is not public so we
// couldn't get at it.
//
// So, we can cook up a simple version of the on-off application that does what
// we want.  On the plus side we don't need all of the complexity of the on-off
// application.  On the minus side, we don't have a helper, so we have to get
// a little more involved in the details, but this is trivial.
//
// So first, we create a socket and do the trace connect on it; then we pass
// this socket into the constructor of our simple application which we then
// install in the source node.
// ============================================================================
//
```

This should also be self-explanatory.

The next part is the declaration of the `MyApp` `Application` that we put together to allow the `Socket` to be created at configuration time.

```
class MyApp : public Application
{
public:

  MyApp ();
  virtual ~MyApp();

  void Setup (Ptr<Socket> socket, Address address, uint32_t packetSize,
    uint32_t nPackets, DataRate dataRate);

private:
  virtual void StartApplication (void);
  virtual void StopApplication (void);

  void ScheduleTx (void);
  void SendPacket (void);

  Ptr<Socket>     m_socket;
  Address         m_peer;
  uint32_t        m_packetSize;
  uint32_t        m_nPackets;
  DataRate        m_dataRate;
  EventId         m_sendEvent;
  bool            m_running;
  uint32_t        m_packetsSent;
};
```

You can see that this class inherits from the *ns-3* `Application` class. Take a look at `src/node/application.h` if you are interested in what is inherited. The `MyApp` class is obligated to override the `StartApplication` and `StopApplication` methods. These methods are automatically called when `MyApp` is required to start and stop sending data during the simulation.

How Applications are Started and Stopped (optional)

It is worthwhile to spend a bit of time explaining how events actually get started in the system. This is another fairly deep explanation, and can be ignored if you aren't planning on venturing down into the guts of the system. It is useful, however, in that the discussion touches on how some very important parts of *ns-3* work and exposes some important idioms. If you are planning on implementing new models, you probably want to understand this section.

The most common way to start pumping events is to start an Application. This is done as the result of the following (hopefully) familar lines of an *ns-3* script:

```
ApplicationContainer apps = ...
apps.Start (Seconds (1.0));
apps.Stop (Seconds (10.0));
```

The application container code (see src/helper/application-container.h if you are interested) loops through its contained applications and calls,

```
app->SetStartTime (startTime);
```

as a result of the apps.Start call and

```
app->SetStopTime (stopTime);
```

as a result of the apps.Stop call.

The ultimate result of these calls is that we want to have the simulator automatically make calls into our Applications to tell them when to start and stop. In the case of MyApp, it inherits from class Application and overrides StartApplication, and StopApplication. These are the functions that will be called by the simulator at the appropriate time. In the case of MyApp you will find that MyApp::StartApplication does the initial Bind, and Connect on the socket, and then starts data flowing by calling MyApp::SendPacket. MyApp::StopApplication stops generating packets by cancelling any pending send events and closing the socket.

One of the nice things about *ns-3* is that you can completely ignore the implementation details of how your Application is "automagically" called by the simulator at the correct time. But since we have already ventured deep into *ns-3* already, let's go for it.

If you look at src/node/application.cc you will find that the SetStartTime method of an Application just sets the member variable m_startTime and the SetStopTime method just sets m_stopTime. From there, without some hints, the trail will probably end.

The key to picking up the trail again is to know that there is a global list of all of the nodes in the system. Whenever you create a node in a simulation, a pointer to that node is added to the global NodeList.

Take a look at src/node/node-list.cc and search for NodeList::Add. The public static implementation calls into a private implementation called NodeListPriv::Add. This is a relatively common idom in *ns-3*. So, take a look at NodeListPriv::Add. There you will find,

```
Simulator::ScheduleWithContext (index, TimeStep (0), &Node::Start, node);
```

This tells you that whenever a Node is created in a simulation, as a side-effect, a call to that node's Start method is scheduled for you that happens at time zero. Don't read too much into that name, yet. It doesn't mean that the node is going to start doing anything, it can be interpreted as an informational call into the Node telling it that the simulation has started, not a call for action telling the Node to start doing something.

So, NodeList::Add indirectly schedules a call to Node::Start at time zero to advise a new node that the simulation has started. If you look in src/node/node.h you will, however, not find a method called Node::Start. It turns out that the Start method is inherited from class Object. All objects in the system can be notified when the simulation starts, and objects of class Node are just one kind of those objects.

Take a look at src/core/object.cc next and search for Object::Start. This code is not as straightforward as you might have expected since *ns-3* Objects support aggregation. The code in Object::Start then loops through all of the objects that have been aggregated together and calls their DoStart method. This is another idiom that is very common in *ns-3*. There is a public API method, that stays constant across implementations, that calls a private implementation method that is inherited and implemented by subclasses. The names are typically something like MethodName for the public API and DoMethodName for the private API.

This tells us that we should look for a `Node::DoStart` method in `src/node/node.cc` for the method that will continue our trail. If you locate the code, you will find a method that loops through all of the devices in the node and then all of the applications in the node calling `device->Start` and `application->Start` respectively.

You may already know that classes `Device` and `Application` both inherit from class `Object` and so the next step will be to look at what happens when `Application::DoStart` is called. Take a look at `src/node/application.cc` and you will find:

```
void
Application::DoStart (void)
{
  m_startEvent = Simulator::Schedule (m_startTime, &Application::StartApplication, this);
  if (m_stopTime != TimeStep (0))
    {
      m_stopEvent = Simulator::Schedule (m_stopTime, &Application::StopApplication, this);
    }
  Object::DoStart ();
}
```

Here, we finally come to the end of the trail. If you have kept it all straight, when you implement an *ns-3* `Application`, your new application inherits from class `Application`. You override the `StartApplication` and `StopApplication` methods and provide mechanisms for starting and stopping the flow of data out of your new `Application`. When a `Node` is created in the simulation, it is added to a global `NodeList`. The act of adding a node to this `NodeList` causes a simulator event to be scheduled for time zero which calls the `Node::Start` method of the newly added `Node` to be called when the simulation starts. Since a `Node` inherits from `Object`, this calls the `Object::Start` method on the `Node` which, in turn, calls the `DoStart` methods on all of the `Objects` aggregated to the `Node` (think mobility models). Since the `Node` `Object` has overridden `DoStart`, that method is called when the simulation starts. The `Node::DoStart` method calls the `Start` methods of all of the `Applications` on the node. Since `Applications` are also `Objects`, this causes `Application::DoStart` to be called. When `Application::DoStart` is called, it schedules events for the `StartApplication` and `StopApplication` calls on the `Application`. These calls are designed to start and stop the flow of data from the `Application`

This has been another fairly long journey, but it only has to be made once, and you now understand another very deep piece of *ns-3*.

The MyApp Application

The `MyApp` `Application` needs a constructor and a destructor, of course:

```
MyApp::MyApp ()
  : m_socket (0),
    m_peer (),
    m_packetSize (0),
    m_nPackets (0),
    m_dataRate (0),
    m_sendEvent (),
    m_running (false),
    m_packetsSent (0)
{
}

MyApp::~MyApp()
{
  m_socket = 0;
}
```

The existence of the next bit of code is the whole reason why we wrote this `Application` in the first place.

```
void
MyApp::Setup (Ptr<Socket> socket, Address address, uint32_t packetSize,
                      uint32_t nPackets, DataRate dataRate)
{
  m_socket = socket;
  m_peer = address;
  m_packetSize = packetSize;
  m_nPackets = nPackets;
  m_dataRate = dataRate;
}
```

This code should be pretty self-explanatory. We are just initializing member variables. The important one from the perspective of tracing is the `Ptr<Socket> socket` which we needed to provide to the application during configuration time. Recall that we are going to create the `Socket` as a `TcpSocket` (which is implemented by `TcpNewReno`) and hook its "CongestionWindow" trace source before passing it to the `Setup` method.

```
void
MyApp::StartApplication (void)
{
  m_running = true;
  m_packetsSent = 0;
  m_socket->Bind ();
  m_socket->Connect (m_peer);
  SendPacket ();
}
```

The above code is the overridden implementation `Application::StartApplication` that will be automatically called by the simulator to start our `Application` running at the appropriate time. You can see that it does a `Socket Bind` operation. If you are familiar with Berkeley Sockets this shouldn't be a surprise. It performs the required work on the local side of the connection just as you might expect. The following `Connect` will do what is required to establish a connection with the TCP at `Address` m_peer. It should now be clear why we need to defer a lot of this to simulation time, since the `Connect` is going to need a fully functioning network to complete. After the `Connect`, the `Application` then starts creating simulation events by calling `SendPacket`.

The next bit of code explains to the `Application` how to stop creating simulation events.

```
void
MyApp::StopApplication (void)
{
  m_running = false;

  if (m_sendEvent.IsRunning ())
    {
      Simulator::Cancel (m_sendEvent);
    }

  if (m_socket)
    {
      m_socket->Close ();
    }
}
```

Every time a simulation event is scheduled, an `Event` is created. If the `Event` is pending execution or executing, its method `IsRunning` will return `true`. In this code, if `IsRunning()` returns true, we `Cancel` the event which removes it from the simulator event queue. By doing this, we break the chain of events that the `Application` is using to keep sending its `Packets` and the `Application` goes quiet. After we quiet the `Application` we `Close` the socket which tears down the TCP connection.

The socket is actually deleted in the destructor when the `m_socket = 0` is executed. This removes the last reference

to the underlying Ptr<Socket> which causes the destructor of that Object to be called.

Recall that `StartApplication` called `SendPacket` to start the chain of events that describes the `Application` behavior.

```
void
MyApp::SendPacket (void)
{
  Ptr<Packet> packet = Create<Packet> (m_packetSize);
  m_socket->Send (packet);

  if (++m_packetsSent < m_nPackets)
    {
      ScheduleTx ();
    }
}
```

Here, you see that `SendPacket` does just that. It creates a `Packet` and then does a `Send` which, if you know Berkeley Sockets, is probably just what you expected to see.

It is the responsibility of the `Application` to keep scheduling the chain of events, so the next lines call `ScheduleTx` to schedule another transmit event (a `SendPacket`) until the `Application` decides it has sent enough.

```
void
MyApp::ScheduleTx (void)
{
  if (m_running)
    {
      Time tNext (Seconds (m_packetSize * 8 / static_cast<double> (m_dataRate.GetBitRate ())));
      m_sendEvent = Simulator::Schedule (tNext, &MyApp::SendPacket, this);
    }
}
```

Here, you see that `ScheduleTx` does exactly that. If the `Application` is running (if `StopApplication` has not been called) it will schedule a new event, which calls `SendPacket` again. The alert reader will spot something that also trips up new users. The data rate of an `Application` is just that. It has nothing to do with the data rate of an underlying `Channel`. This is the rate at which the `Application` produces bits. It does not take into account any overhead for the various protocols or channels that it uses to transport the data. If you set the data rate of an `Application` to the same data rate as your underlying `Channel` you will eventually get a buffer overflow.

The Trace Sinks

The whole point of this exercise is to get trace callbacks from TCP indicating the congestion window has been updated. The next piece of code implements the corresponding trace sink:

```
static void
CwndChange (uint32_t oldCwnd, uint32_t newCwnd)
{
  NS_LOG_UNCOND (Simulator::Now ().GetSeconds () << "\t" << newCwnd);
}
```

This should be very familiar to you now, so we won't dwell on the details. This function just logs the current simulation time and the new value of the congestion window every time it is changed. You can probably imagine that you could load the resulting output into a graphics program (gnuplot or Excel) and immediately see a nice graph of the congestion window behavior over time.

We added a new trace sink to show where packets are dropped. We are going to add an error model to this code also, so we wanted to demonstrate this working.

```
static void
RxDrop (Ptr<const Packet> p)
{
  NS_LOG_UNCOND ("RxDrop at " << Simulator::Now ().GetSeconds ());
}
```

This trace sink will be connected to the "PhyRxDrop" trace source of the point-to-point NetDevice. This trace source fires when a packet is dropped by the physical layer of a `NetDevice`. If you take a small detour to the source (`src/devices/point-to-point/point-to-point-net-device.cc`) you will see that this trace source refers to `PointToPointNetDevice::m_phyRxDropTrace`. If you then look in `src/devices/point-to-point/point-to-point-net-device.h` for this member variable, you will find that it is declared as a `TracedCallback<Ptr<const Packet> >`. This should tell you that the callback target should be a function that returns void and takes a single parameter which is a `Ptr<const Packet>` – just what we have above.

The Main Program

The following code should be very familiar to you by now:

```
int
main (int argc, char *argv[])
{
  NodeContainer nodes;
  nodes.Create (2);

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

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

This creates two nodes with a point-to-point channel between them, just as shown in the illustration at the start of the file.

The next few lines of code show something new. If we trace a connection that behaves perfectly, we will end up with a monotonically increasing congestion window. To see any interesting behavior, we really want to introduce link errors which will drop packets, cause duplicate ACKs and trigger the more interesting behaviors of the congestion window.

ns-3 provides `ErrorModel` objects which can be attached to `Channels`. We are using the `RateErrorModel` which allows us to introduce errors into a `Channel` at a given *rate*.

```
Ptr<RateErrorModel> em = CreateObjectWithAttributes<RateErrorModel> (
  "RanVar", RandomVariableValue (UniformVariable (0., 1.)),
  "ErrorRate", DoubleValue (0.00001));
devices.Get (1)->SetAttribute ("ReceiveErrorModel", PointerValue (em));
```

The above code instantiates a `RateErrorModel` Object. Rather than using the two-step process of instantiating it and then setting Attributes, we use the convenience function `CreateObjectWithAttributes` which allows us to do both at the same time. We set the "RanVar" `Attribute` to a random variable that generates a uniform distribution from 0 to 1. We also set the "ErrorRate" `Attribute`. We then set the resulting instantiated `RateErrorModel` as the error model used by the point-to-point `NetDevice`. This will give us some retransmissions and make our plot a little more interesting.

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

```
Ipv4AddressHelper address;
address.SetBase ("10.1.1.0", "255.255.255.252");
Ipv4InterfaceContainer interfaces = address.Assign (devices);
```

The above code should be familiar. It installs internet stacks on our two nodes and creates interfaces and assigns IP addresses for the point-to-point devices.

Since we are using TCP, we need something on the destination node to receive TCP connections and data. The `PacketSink Application` is commonly used in *ns-3* for that purpose.

```
uint16_t sinkPort = 8080;
Address sinkAddress (InetSocketAddress(interfaces.GetAddress (1), sinkPort));
PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory",
  InetSocketAddress (Ipv4Address::GetAny (), sinkPort));
ApplicationContainer sinkApps = packetSinkHelper.Install (nodes.Get (1));
sinkApps.Start (Seconds (0.));
sinkApps.Stop (Seconds (20.));
```

This should all be familiar, with the exception of,

```
PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory",
  InetSocketAddress (Ipv4Address::GetAny (), sinkPort));
```

This code instantiates a `PacketSinkHelper` and tells it to create sockets using the class `ns3::TcpSocketFactory`. This class implements a design pattern called "object factory" which is a commonly used mechanism for specifying a class used to create objects in an abstract way. Here, instead of having to create the objects themselves, you provide the `PacketSinkHelper` a string that specifies a `TypeId` string used to create an object which can then be used, in turn, to create instances of the Objects created by the factory.

The remaining parameter tells the `Application` which address and port it should `Bind` to.

The next two lines of code will create the socket and connect the trace source.

```
Ptr<Socket> ns3TcpSocket = Socket::CreateSocket (nodes.Get (0),
  TcpSocketFactory::GetTypeId ());
ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow",
  MakeCallback (&CwndChange));
```

The first statement calls the static member function `Socket::CreateSocket` and provides a `Node` and an explicit `TypeId` for the object factory used to create the socket. This is a slightly lower level call than the `PacketSinkHelper` call above, and uses an explicit C++ type instead of one referred to by a string. Otherwise, it is conceptually the same thing.

Once the `TcpSocket` is created and attached to the `Node`, we can use `TraceConnectWithoutContext` to connect the CongestionWindow trace source to our trace sink.

Recall that we coded an `Application` so we could take that `Socket` we just made (during configuration time) and use it in simulation time. We now have to instantiate that `Application`. We didn't go to any trouble to create a helper to manage the `Application` so we are going to have to create and install it "manually". This is actually quite easy:

```
Ptr<MyApp> app = CreateObject<MyApp> ();
app->Setup (ns3TcpSocket, sinkAddress, 1040, 1000, DataRate ("1Mbps"));
nodes.Get (0)->AddApplication (app);
app->Start (Seconds (1.));
app->Stop (Seconds (20.));
```

The first line creates an `Object` of type `MyApp` – our `Application`. The second line tells the `Application` what `Socket` to use, what address to connect to, how much data to send at each send event, how many send events to generate and the rate at which to produce data from those events.

Next, we manually add the `MyApp Application` to the source node and explicitly call the `Start` and `Stop` methods on the `Application` to tell it when to start and stop doing its thing.

We need to actually do the connect from the receiver point-to-point `NetDevice` to our callback now.

```
devices.Get (1)->TraceConnectWithoutContext("PhyRxDrop", MakeCallback (&RxDrop));
```

It should now be obvious that we are getting a reference to the receiving `Node NetDevice` from its container and connecting the trace source defined by the attribute "PhyRxDrop" on that device to the trace sink `RxDrop`.

Finally, we tell the simulator to override any `Applications` and just stop processing events at 20 seconds into the simulation.

```
  Simulator::Stop (Seconds(20));
  Simulator::Run ();
  Simulator::Destroy ();

  return 0;
}
```

Recall that as soon as `Simulator::Run` is called, configuration time ends, and simulation time begins. All of the work we orchestrated by creating the `Application` and teaching it how to connect and send data actually happens during this function call.

As soon as `Simulator::Run` returns, the simulation is complete and we enter the teardown phase. In this case, `Simulator::Destroy` takes care of the gory details and we just return a success code after it completes.

7.3.5 Running fifth.cc

Since we have provided the file `fifth.cc` for you, if you have built your distribution (in debug mode since it uses NS_LOG – recall that optimized builds optimize out NS_LOGs) it will be waiting for you to run.

```
./waf --run fifth
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone-dev/ns-3-dev/build
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone-dev/ns-3-dev/build'
'build' finished successfully (0.684s)
1.20919 1072
1.21511 1608
1.22103 2144
...
1.2471  8040
1.24895 8576
1.2508  9112
RxDrop at 1.25151
...
```

You can probably see immediately a downside of using prints of any kind in your traces. We get those extraneous waf messages printed all over our interesting information along with those RxDrop messages. We will remedy that soon, but I'm sure you can't wait to see the results of all of this work. Let's redirect that output to a file called `cwnd.dat`:

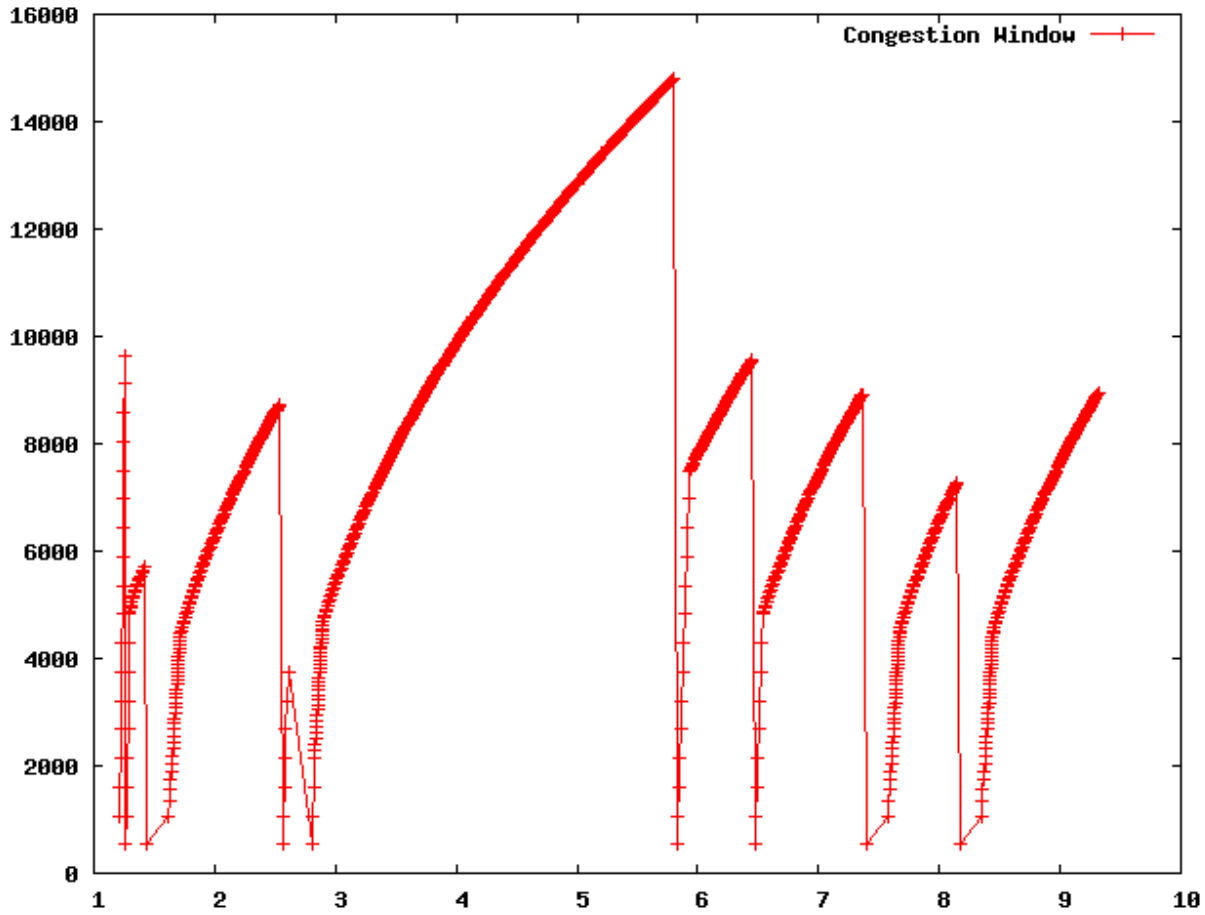
```
./waf --run fifth > cwnd.dat 2>&1
```

Now edit up "cwnd.dat" in your favorite editor and remove the waf build status and drop lines, leaving only the traced data (you could also comment out the `TraceConnectWithoutContext("PhyRxDrop", MakeCallback (&RxDrop));` in the script to get rid of the drop prints just as easily.

You can now run gnuplot (if you have it installed) and tell it to generate some pretty pictures:

```
gnuplot> set terminal png size 640,480
gnuplot> set output "cwnd.png"
gnuplot> plot "cwnd.dat" using 1:2 title 'Congestion Window' with linespoints
gnuplot> exit
```

You should now have a graph of the congestion window versus time sitting in the file "cwnd.png" that looks like:



7.3.6 Using Mid-Level Helpers

In the previous section, we showed how to hook a trace source and get hopefully interesting information out of a simulation. Perhaps you will recall that we called logging to the standard output using `std::cout` a "Blunt Instrument" much earlier in this chapter. We also wrote about how it was a problem having to parse the log output in order to isolate interesting information. It may have occurred to you that we just spent a lot of time implementing an example that exhibits all of the problems we purport to fix with the *ns-3* tracing system! You would be correct. But, bear with us. We're not done yet.

One of the most important things we want to do is to is to have the ability to easily control the amount of output coming out of the simulation; and we also want to save those data to a file so we can refer back to it later. We can use the mid-level trace helpers provided in *ns-3* to do just that and complete the picture.

We provide a script that writes the cwnd change and drop events developed in the example `fifth.cc` to disk in separate files. The cwnd changes are stored as a tab-separated ASCII file and the drop events are stored in a pcap file. The changes to make this happen are quite small.

A sixth.cc Walkthrough

Let's take a look at the changes required to go from `fifth.cc` to `sixth.cc`. Open `examples/tutorial/fifth.cc` in your favorite editor. You can see the first change by searching for CwndChange. You will find that we have changed the signatures for the trace sinks and have added a single line to each sink that writes the traced information to a stream representing a file.

```
static void
CwndChange (Ptr<OutputStreamWrapper> stream, uint32_t oldCwnd, uint32_t newCwnd)
{
  NS_LOG_UNCOND (Simulator::Now ().GetSeconds () << "\t" << newCwnd);
  *stream->GetStream () << Simulator::Now ().GetSeconds () << "\t" << oldCwnd << "\t" << newCwnd << s
}

static void
RxDrop (Ptr<PcapFileWrapper> file, Ptr<const Packet> p)
{
  NS_LOG_UNCOND ("RxDrop at " << Simulator::Now ().GetSeconds ());
  file->Write(Simulator::Now(), p);
}
```

We have added a "stream" parameter to the `CwndChange` trace sink. This is an object that holds (keeps safely alive) a C++ output stream. It turns out that this is a very simple object, but one that manages lifetime issues for the stream and solves a problem that even experienced C++ users run into. It turns out that the copy constructor for ostream is marked private. This means that ostreams do not obey value semantics and cannot be used in any mechanism that requires the stream to be copied. This includes the *ns-3* callback system, which as you may recall, requires objects that obey value semantics. Further notice that we have added the following line in the `CwndChange` trace sink implementation:

```
*stream->GetStream () << Simulator::Now ().GetSeconds () << "\t" << oldCwnd << "\t" << newCwnd << std
```

This would be very familiar code if you replaced `*stream->GetStream ()` with `std::cout`, as in:

```
std::cout << Simulator::Now ().GetSeconds () << "\t" << oldCwnd << "\t" << newCwnd << std::endl;
```

This illustrates that the `Ptr<OutputStreamWrapper>` is really just carrying around a `std::ofstream` for you, and you can use it here like any other output stream.

A similar situation happens in `RxDrop` except that the object being passed around (a `Ptr<PcapFileWrapper>`) represents a pcap file. There is a one-liner in the trace sink to write a timestamp and the contents of the packet being dropped to the pcap file:

```
file->Write(Simulator::Now(), p);
```

Of course, if we have objects representing the two files, we need to create them somewhere and also cause them to be passed to the trace sinks. If you look in the `main` function, you will find new code to do just that:

```
AsciiTraceHelper asciiTraceHelper;
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("sixth.cwnd");
ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow", MakeBoundCallback (&CwndChange, stream)

...

PcapHelper pcapHelper;
Ptr<PcapFileWrapper> file = pcapHelper.CreateFile ("sixth.pcap", std::ios::out, PcapHelper::DLT_PPP);
devices.Get (1)->TraceConnectWithoutContext("PhyRxDrop", MakeBoundCallback (&RxDrop, file));
```

In the first section of the code snippet above, we are creating the ASCII trace file, creating an object responsible for managing it and using a variant of the callback creation function to arrange for the object to be passed to the sink. Our

ASCII trace helpers provide a rich set of functions to make using text (ASCII) files easy. We are just going to illustrate the use of the file stream creation function here.

The `CreateFileStream{}` function is basically going to instantiate a std::ofstream object and create a new file (or truncate an existing file). This ofstream is packaged up in an *ns-3* object for lifetime management and copy constructor issue resolution.

We then take this *ns-3* object representing the file and pass it to `MakeBoundCallback()`. This function creates a callback just like `MakeCallback()`, but it "binds" a new value to the callback. This value is added to the callback before it is called.

Essentially, `MakeBoundCallback(&CwndChange, stream)` causes the trace source to add the additional "stream" parameter to the front of the formal parameter list before invoking the callback. This changes the required signature of the `CwndChange` sink to match the one shown above, which includes the "extra" parameter `Ptr<OutputStreamWrapper> stream`.

In the second section of code in the snippet above, we instantiate a `PcapHelper` to do the same thing for our pcap trace file that we did with the `AsciiTraceHelper`. The line of code,

```
Ptr<PcapFileWrapper> file = pcapHelper.CreateFile ("sixth.pcap", "w", PcapHelper::DLT_PPP);
```

creates a pcap file named "sixth.pcap" with file mode "w". This means that the new file is to truncated if an existing file with that name is found. The final parameter is the "data link type" of the new pcap file. These are the same as the pcap library data link types defined in `bpf.h` if you are familar with pcap. In this case, `DLT_PPP` indicates that the pcap file is going to contain packets prefixed with point to point headers. This is true since the packets are coming from our point-to-point device driver. Other common data link types are DLT_EN10MB (10 MB Ethernet) appropriate for csma devices and DLT_IEEE802_11 (IEEE 802.11) appropriate for wifi devices. These are defined in `src/helper/trace-helper.h"` if you are interested in seeing the list. The entries in the list match those in `bpf.h` but we duplicate them to avoid a pcap source dependence.

A *ns-3* object representing the pcap file is returned from `CreateFile` and used in a bound callback exactly as it was in the ascii case.

An important detour: It is important to notice that even though both of these objects are declared in very similar ways,

```
Ptr<PcapFileWrapper> file ...
Ptr<OutputStreamWrapper> stream ...
```

The underlying objects are entirely different. For example, the Ptr<PcapFileWrapper> is a smart pointer to an *ns-3* Object that is a fairly heaviweight thing that supports `Attributes` and is integrated into the config system. The Ptr<OutputStreamWrapper>, on the other hand, is a smart pointer to a reference counted object that is a very lightweight thing. Remember to always look at the object you are referencing before making any assumptions about the "powers" that object may have.

For example, take a look at `src/common/pcap-file-object.h` in the distribution and notice,

```
class PcapFileWrapper : public Object
```

that class `PcapFileWrapper` is an *ns-3* Object by virtue of its inheritance. Then look at `src/common/output-stream-wrapper.h` and notice,

```
class OutputStreamWrapper : public SimpleRefCount<OutputStreamWrapper>
```

that this object is not an *ns-3* Object at all, it is "merely" a C++ object that happens to support intrusive reference counting.

The point here is that just because you read Ptr<something> it does not necessarily mean that "something" is an *ns-3* Object on which you can hang *ns-3* `Attributes`, for example.

Now, back to the example. If you now build and run this example,

```
./waf --run sixth
```

you will see the same messages appear as when you ran "fifth", but two new files will appear in the top-level directory of your *ns-3* distribution.

```
sixth.cwnd  sixth.pcap
```

Since "sixth.cwnd" is an ASCII text file, you can view it with `cat` or your favorite file viewer.

```
1.20919 536     1072
1.21511 1072    1608
...
9.30922 8893    8925
9.31754 8925    8957
```

You have a tab separated file with a timestamp, an old congestion window and a new congestion window suitable for directly importing into your plot program. There are no extraneous prints in the file, no parsing or editing is required.

Since "sixth.pcap" is a pcap file, you can fiew it with `tcpdump`.

```
reading from file ../../sixth.pcap, link-type PPP (PPP)
1.251507 IP 10.1.1.1.49153 > 10.1.1.2.8080: . 17689:18225(536) ack 1 win 65535
1.411478 IP 10.1.1.1.49153 > 10.1.1.2.8080: . 33808:34312(504) ack 1 win 65535
...
7.393557 IP 10.1.1.1.49153 > 10.1.1.2.8080: . 781568:782072(504) ack 1 win 65535
8.141483 IP 10.1.1.1.49153 > 10.1.1.2.8080: . 874632:875168(536) ack 1 win 65535
```

You have a pcap file with the packets that were dropped in the simulation. There are no other packets present in the file and there is nothing else present to make life difficult.

It's been a long journey, but we are now at a point where we can appreciate the *ns-3* tracing system. We have pulled important events out of the middle of a TCP implementation and a device driver. We stored those events directly in files usable with commonly known tools. We did this without modifying any of the core code involved, and we did this in only 18 lines of code:

```
static void
CwndChange (Ptr<OutputStreamWrapper> stream, uint32_t oldCwnd, uint32_t newCwnd)
{
  NS_LOG_UNCOND (Simulator::Now ().GetSeconds () << "\t" << newCwnd);
  *stream->GetStream () << Simulator::Now ().GetSeconds () << "\t" << oldCwnd << "\t" << newCwnd << s
}

...

AsciiTraceHelper asciiTraceHelper;
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("sixth.cwnd");
ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow", MakeBoundCallback (&CwndChange, stream)

...

static void
RxDrop (Ptr<PcapFileWrapper> file, Ptr<const Packet> p)
{
  NS_LOG_UNCOND ("RxDrop at " << Simulator::Now ().GetSeconds ());
  file->Write(Simulator::Now(), p);
}

...

PcapHelper pcapHelper;
```

```
Ptr<PcapFileWrapper> file = pcapHelper.CreateFile ("sixth.pcap", "w", PcapHelper::DLT_PPP);
devices.Get (1)->TraceConnectWithoutContext("PhyRxDrop", MakeBoundCallback (&RxDrop, file));
```

7.4 Using Trace Helpers

The *ns-3* trace helpers provide a rich environment for configuring and selecting different trace events and writing them to files. In previous sections, primarily "Building Topologies," we have seen several varieties of the trace helper methods designed for use inside other (device) helpers.

Perhaps you will recall seeing some of these variations:

```
pointToPoint.EnablePcapAll ("second");
pointToPoint.EnablePcap ("second", p2pNodes.Get (0)->GetId (), 0);
csma.EnablePcap ("third", csmaDevices.Get (0), true);
pointToPoint.EnableAsciiAll (ascii.CreateFileStream ("myfirst.tr"));
```

What may not be obvious, though, is that there is a consistent model for all of the trace-related methods found in the system. We will now take a little time and take a look at the "big picture".

There are currently two primary use cases of the tracing helpers in *ns-3*: Device helpers and protocol helpers. Device helpers look at the problem of specifying which traces should be enabled through a node, device pair. For example, you may want to specify that pcap tracing should be enabled on a particular device on a specific node. This follows from the *ns-3* device conceptual model, and also the conceptual models of the various device helpers. Following naturally from this, the files created follow a <prefix>-<node>-<device> naming convention.

Protocol helpers look at the problem of specifying which traces should be enabled through a protocol and interface pair. This follows from the *ns-3* protocol stack conceptual model, and also the conceptual models of internet stack helpers. Naturally, the trace files should follow a <prefix>-<protocol>-<interface> naming convention.

The trace helpers therefore fall naturally into a two-dimensional taxonomy. There are subtleties that prevent all four classes from behaving identically, but we do strive to make them all work as similarly as possible; and whenever possible there are analogs for all methods in all classes.

```
                 | pcap | ascii |
-----------------+------+-------|
Device Helper    |      |       |
-----------------+------+-------|
Protocol Helper  |      |       |
-----------------+------+-------|
```

We use an approach called a `mixin` to add tracing functionality to our helper classes. A `mixin` is a class that provides functionality to that is inherited by a subclass. Inheriting from a mixin is not considered a form of specialization but is really a way to collect functionality.

Let's take a quick look at all four of these cases and their respective `mixins`.

7.4.1 Pcap Tracing Device Helpers

The goal of these helpers is to make it easy to add a consistent pcap trace facility to an *ns-3* device. We want all of the various flavors of pcap tracing to work the same across all devices, so the methods of these helpers are inherited by device helpers. Take a look at `src/helper/trace-helper.h` if you want to follow the discussion while looking at real code.

The class `PcapHelperForDevice` is a `mixin` provides the high level functionality for using pcap tracing in an *ns-3* device. Every device must implement a single virtual method inherited from this class.

```
virtual void EnablePcapInternal (std::string prefix, Ptr<NetDevice> nd, bool promiscuous, bool explic
```

The signature of this method reflects the device-centric view of the situation at this level. All of the public methods inherited from class 2‘‘PcapUserHelperForDevice‘‘ reduce to calling this single device-dependent implementation method. For example, the lowest level pcap method,

```
void EnablePcap (std::string prefix, Ptr<NetDevice> nd, bool promiscuous = false, bool explicitFilena
```

will call the device implementation of `EnablePcapInternal` directly. All other public pcap tracing methods build on this implementation to provide additional user-level functionality. What this means to the user is that all device helpers in the system will have all of the pcap trace methods available; and these methods will all work in the same way across devices if the device implements `EnablePcapInternal` correctly.

Pcap Tracing Device Helper Methods

```
void EnablePcap (std::string prefix, Ptr<NetDevice> nd, bool promiscuous = false, bool explicitFilena
void EnablePcap (std::string prefix, std::string ndName, bool promiscuous = false, bool explicitFile
void EnablePcap (std::string prefix, NetDeviceContainer d, bool promiscuous = false);
void EnablePcap (std::string prefix, NodeContainer n, bool promiscuous = false);
void EnablePcap (std::string prefix, uint32_t nodeid, uint32_t deviceid, bool promiscuous = false);
void EnablePcapAll (std::string prefix, bool promiscuous = false);
```

In each of the methods shown above, there is a default parameter called `promiscuous` that defaults to false. This parameter indicates that the trace should not be gathered in promiscuous mode. If you do want your traces to include all traffic seen by the device (and if the device supports a promiscuous mode) simply add a true parameter to any of the calls above. For example,

```
Ptr<NetDevice> nd;
...
helper.EnablePcap ("prefix", nd, true);
```

will enable promiscuous mode captures on the `NetDevice` specified by `nd`.

The first two methods also include a default parameter called `explicitFilename` that will be discussed below.

You are encouraged to peruse the Doxygen for class `PcapHelperForDevice` to find the details of these methods; but to summarize ...

You can enable pcap tracing on a particular node/net-device pair by providing a `Ptr<NetDevice>` to an `EnablePcap` method. The `Ptr<Node>` is implicit since the net device must belong to exactly one `Node`. For example,

```
Ptr<NetDevice> nd;
...
helper.EnablePcap ("prefix", nd);
```

You can enable pcap tracing on a particular node/net-device pair by providing a `std::string` representing an object name service string to an `EnablePcap` method. The `Ptr<NetDevice>` is looked up from the name string. Again, the `<Node>` is implicit since the named net device must belong to exactly one `Node`. For example,

```
Names::Add ("server" ...);
Names::Add ("server/eth0" ...);
...
helper.EnablePcap ("prefix", "server/ath0");
```

You can enable pcap tracing on a collection of node/net-device pairs by providing a `NetDeviceContainer`. For each `NetDevice` in the container the type is checked. For each device of the proper type (the same type as is managed

by the device helper), tracing is enabled. Again, the <Node> is implicit since the found net device must belong to exactly one Node. For example,

```
NetDeviceContainer d = ...;
...
helper.EnablePcap ("prefix", d);
```

You can enable pcap tracing on a collection of node/net-device pairs by providing a NodeContainer. For each Node in the NodeContainer its attached NetDevices are iterated. For each NetDevice attached to each node in the container, the type of that device is checked. For each device of the proper type (the same type as is managed by the device helper), tracing is enabled.

```
NodeContainer n;
...
helper.EnablePcap ("prefix", n);
```

You can enable pcap tracing on the basis of node ID and device ID as well as with explicit Ptr. Each Node in the system has an integer node ID and each device connected to a node has an integer device ID.

```
helper.EnablePcap ("prefix", 21, 1);
```

Finally, you can enable pcap tracing for all devices in the system, with the same type as that managed by the device helper.

```
helper.EnablePcapAll ("prefix");
```

Pcap Tracing Device Helper Filename Selection

Implicit in the method descriptions above is the construction of a complete filename by the implementation method. By convention, pcap traces in the *ns-3* system are of the form "<prefix>-<node id>-<device id>.pcap"

As previously mentioned, every node in the system will have a system-assigned node id; and every device will have an interface index (also called a device id) relative to its node. By default, then, a pcap trace file created as a result of enabling tracing on the first device of node 21 using the prefix "prefix" would be "prefix-21-1.pcap".

You can always use the *ns-3* object name service to make this more clear. For example, if you use the object name service to assign the name "server" to node 21, the resulting pcap trace file name will automatically become, "prefix-server-1.pcap" and if you also assign the name "eth0" to the device, your pcap file name will automatically pick this up and be called "prefix-server-eth0.pcap".

Finally, two of the methods shown above,

```
void EnablePcap (std::string prefix, Ptr<NetDevice> nd, bool promiscuous = false, bool explicitFilena
void EnablePcap (std::string prefix, std::string ndName, bool promiscuous = false, bool explicitFile
```

have a default parameter called explicitFilename. When set to true, this parameter disables the automatic filename completion mechanism and allows you to create an explicit filename. This option is only available in the methods which enable pcap tracing on a single device.

For example, in order to arrange for a device helper to create a single promiscuous pcap capture file of a specific name ("my-pcap-file.pcap") on a given device, one could:

```
Ptr<NetDevice> nd;
...
helper.EnablePcap ("my-pcap-file.pcap", nd, true, true);
```

The first true parameter enables promiscuous mode traces and the second tells the helper to interpret the prefix parameter as a complete filename.

7.4.2 Ascii Tracing Device Helpers

The behavior of the ascii trace helper `mixin` is substantially similar to the pcap version. Take a look at `src/helper/trace-helper.h` if you want to follow the discussion while looking at real code.

The class `AsciiTraceHelperForDevice` adds the high level functionality for using ascii tracing to a device helper class. As in the pcap case, every device must implement a single virtual method inherited from the ascii trace `mixin`.

```
virtual void EnableAsciiInternal (Ptr<OutputStreamWrapper> stream,
                                  std::string prefix,
                                  Ptr<NetDevice> nd,
                                  bool explicitFilename) = 0;
```

The signature of this method reflects the device-centric view of the situation at this level; and also the fact that the helper may be writing to a shared output stream. All of the public ascii-trace-related methods inherited from class `AsciiTraceHelperForDevice` reduce to calling this single device- dependent implementation method. For example, the lowest level ascii trace methods,

```
void EnableAscii (std::string prefix, Ptr<NetDevice> nd, bool explicitFilename = false);
void EnableAscii (Ptr<OutputStreamWrapper> stream, Ptr<NetDevice> nd);
```

will call the device implementation of `EnableAsciiInternal` directly, providing either a valid prefix or stream. All other public ascii tracing methods will build on these low-level functions to provide additional user-level functionality. What this means to the user is that all device helpers in the system will have all of the ascii trace methods available; and these methods will all work in the same way across devices if the devices implement `EnablAsciiInternal` correctly.

Ascii Tracing Device Helper Methods

```
void EnableAscii (std::string prefix, Ptr<NetDevice> nd, bool explicitFilename = false);
void EnableAscii (Ptr<OutputStreamWrapper> stream, Ptr<NetDevice> nd);

void EnableAscii (std::string prefix, std::string ndName, bool explicitFilename = false);
void EnableAscii (Ptr<OutputStreamWrapper> stream, std::string ndName);

void EnableAscii (std::string prefix, NetDeviceContainer d);
void EnableAscii (Ptr<OutputStreamWrapper> stream, NetDeviceContainer d);

void EnableAscii (std::string prefix, NodeContainer n);
void EnableAscii (Ptr<OutputStreamWrapper> stream, NodeContainer n);

void EnableAsciiAll (std::string prefix);
void EnableAsciiAll (Ptr<OutputStreamWrapper> stream);

void EnableAscii (std::string prefix, uint32_t nodeid, uint32_t deviceid, bool explicitFilename);
void EnableAscii (Ptr<OutputStreamWrapper> stream, uint32_t nodeid, uint32_t deviceid);
```

You are encouraged to peruse the Doxygen for class `TraceHelperForDevice` to find the details of these methods; but to summarize ...

There are twice as many methods available for ascii tracing as there were for pcap tracing. This is because, in addition to the pcap-style model where traces from each unique node/device pair are written to a unique file, we support a model in which trace information for many node/device pairs is written to a common file. This means that the <prefix>-<node>-<device> file name generation mechanism is replaced by a mechanism to refer to a common file; and the number of API methods is doubled to allow all combinations.

Just as in pcap tracing, you can enable ascii tracing on a particular node/net-device pair by providing a `Ptr<NetDevice>` to an `EnableAscii` method. The `Ptr<Node>` is implicit since the net device must belong to exactly one `Node`. For example,

```
Ptr<NetDevice> nd;
...
helper.EnableAscii ("prefix", nd);
```

The first four methods also include a default parameter called `explicitFilename` that operate similar to equivalent parameters in the pcap case.

In this case, no trace contexts are written to the ascii trace file since they would be redundant. The system will pick the file name to be created using the same rules as described in the pcap section, except that the file will have the suffix ".tr" instead of ".pcap".

If you want to enable ascii tracing on more than one net device and have all traces sent to a single file, you can do that as well by using an object to refer to a single file. We have already seen this in the "cwnd" example above:

```
Ptr<NetDevice> nd1;
Ptr<NetDevice> nd2;
...
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("trace-file-name.tr");
...
helper.EnableAscii (stream, nd1);
helper.EnableAscii (stream, nd2);
```

In this case, trace contexts are written to the ascii trace file since they are required to disambiguate traces from the two devices. Note that since the user is completely specifying the file name, the string should include the ",tr" for consistency.

You can enable ascii tracing on a particular node/net-device pair by providing a `std::string` representing an object name service string to an `EnablePcap` method. The `Ptr<NetDevice>` is looked up from the name string. Again, the `<Node>` is implicit since the named net device must belong to exactly one `Node`. For example,

```
Names::Add ("client" ...);
Names::Add ("client/eth0" ...);
Names::Add ("server" ...);
Names::Add ("server/eth0" ...);
...
helper.EnableAscii ("prefix", "client/eth0");
helper.EnableAscii ("prefix", "server/eth0");
```

This would result in two files named "prefix-client-eth0.tr" and "prefix-server-eth0.tr" with traces for each device in the respective trace file. Since all of the EnableAscii functions are overloaded to take a stream wrapper, you can use that form as well:

```
Names::Add ("client" ...);
Names::Add ("client/eth0" ...);
Names::Add ("server" ...);
Names::Add ("server/eth0" ...);
...
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("trace-file-name.tr");
...
helper.EnableAscii (stream, "client/eth0");
helper.EnableAscii (stream, "server/eth0");
```

This would result in a single trace file called "trace-file-name.tr" that contains all of the trace events for both devices. The events would be disambiguated by trace context strings.

You can enable ascii tracing on a collection of node/net-device pairs by providing a `NetDeviceContainer`. For

each NetDevice in the container the type is checked. For each device of the proper type (the same type as is managed by the device helper), tracing is enabled. Again, the <Node> is implicit since the found net device must belong to exactly one Node. For example,

```
NetDeviceContainer d = ...;
...
helper.EnableAscii ("prefix", d);
```

This would result in a number of ascii trace files being created, each of which follows the <prefix>-<node id>-<device id>.tr convention. Combining all of the traces into a single file is accomplished similarly to the examples above:

```
NetDeviceContainer d = ...;
...
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("trace-file-name.tr");
...
helper.EnableAscii (stream, d);
```

You can enable ascii tracing on a collection of node/net-device pairs by providing a NodeContainer. For each Node in the NodeContainer its attached NetDevices are iterated. For each NetDevice attached to each node in the container, the type of that device is checked. For each device of the proper type (the same type as is managed by the device helper), tracing is enabled.

```
NodeContainer n;
...
helper.EnableAscii ("prefix", n);
```

This would result in a number of ascii trace files being created, each of which follows the <prefix>-<node id>-<device id>.tr convention. Combining all of the traces into a single file is accomplished similarly to the examples above:

You can enable pcap tracing on the basis of node ID and device ID as well as with explicit Ptr. Each Node in the system has an integer node ID and each device connected to a node has an integer device ID.

```
helper.EnableAscii ("prefix", 21, 1);
```

Of course, the traces can be combined into a single file as shown above.

Finally, you can enable pcap tracing for all devices in the system, with the same type as that managed by the device helper.

```
helper.EnableAsciiAll ("prefix");
```

This would result in a number of ascii trace files being created, one for every device in the system of the type managed by the helper. All of these files will follow the <prefix>-<node id>-<device id>.tr convention. Combining all of the traces into a single file is accomplished similarly to the examples above.

Ascii Tracing Device Helper Filename Selection

Implicit in the prefix-style method descriptions above is the construction of the complete filenames by the implementation method. By convention, ascii traces in the *ns-3* system are of the form "<prefix>-<node id>-<device id>.tr"

As previously mentioned, every node in the system will have a system-assigned node id; and every device will have an interface index (also called a device id) relative to its node. By default, then, an ascii trace file created as a result of enabling tracing on the first device of node 21, using the prefix "prefix", would be "prefix-21-1.tr".

You can always use the *ns-3* object name service to make this more clear. For example, if you use the object name service to assign the name "server" to node 21, the resulting ascii trace file name will automatically become, "prefix-server-1.tr" and if you also assign the name "eth0" to the device, your ascii trace file name will automatically pick this up and be called "prefix-server-eth0.tr".

Several of the methods have a default parameter called `explicitFilename`. When set to true, this parameter disables the automatic filename completion mechanism and allows you to create an explicit filename. This option is only available in the methods which take a prefix and enable tracing on a single device.

7.4.3 Pcap Tracing Protocol Helpers

The goal of these `mixins` is to make it easy to add a consistent pcap trace facility to protocols. We want all of the various flavors of pcap tracing to work the same across all protocols, so the methods of these helpers are inherited by stack helpers. Take a look at `src/helper/trace-helper.h` if you want to follow the discussion while looking at real code.

In this section we will be illustrating the methods as applied to the protocol `Ipv4`. To specify traces in similar protocols, just substitute the appropriate type. For example, use a `Ptr<Ipv6>` instead of a `Ptr<Ipv4>` and call `EnablePcapIpv6` instead of `EnablePcapIpv4`.

The class `PcapHelperForIpv4` provides the high level functionality for using pcap tracing in the `Ipv4` protocol. Each protocol helper enabling these methods must implement a single virtual method inherited from this class. There will be a separate implementation for `Ipv6`, for example, but the only difference will be in the method names and signatures. Different method names are required to disambiguate class `Ipv4` from `Ipv6` which are both derived from class `Object`, and methods that share the same signature.

```
virtual void EnablePcapIpv4Internal (std::string prefix,
                                     Ptr<Ipv4> ipv4,
                                     uint32_t interface,
                                     bool explicitFilename) = 0;
```

The signature of this method reflects the protocol and interface-centric view of the situation at this level. All of the public methods inherited from class `PcapHelperForIpv4` reduce to calling this single device-dependent implementation method. For example, the lowest level pcap method,

```
void EnablePcapIpv4 (std::string prefix, Ptr<Ipv4> ipv4, uint32_t interface, bool explicitFilename =
```

will call the device implementation of `EnablePcapIpv4Internal` directly. All other public pcap tracing methods build on this implementation to provide additional user-level functionality. What this means to the user is that all protocol helpers in the system will have all of the pcap trace methods available; and these methods will all work in the same way across protocols if the helper implements `EnablePcapIpv4Internal` correctly.

Pcap Tracing Protocol Helper Methods

These methods are designed to be in one-to-one correspondence with the `Node`- and `NetDevice`- centric versions of the device versions. Instead of `Node` and `NetDevice` pair constraints, we use protocol and interface constraints.

Note that just like in the device version, there are six methods:

```
void EnablePcapIpv4 (std::string prefix, Ptr<Ipv4> ipv4, uint32_t interface, bool explicitFilename =
void EnablePcapIpv4 (std::string prefix, std::string ipv4Name, uint32_t interface, bool explicitFile
void EnablePcapIpv4 (std::string prefix, Ipv4InterfaceContainer c);
void EnablePcapIpv4 (std::string prefix, NodeContainer n);
void EnablePcapIpv4 (std::string prefix, uint32_t nodeid, uint32_t interface, bool explicitFilename);
void EnablePcapIpv4All (std::string prefix);
```

You are encouraged to peruse the Doxygen for class `PcapHelperForIpv4` to find the details of these methods; but to summarize ...

You can enable pcap tracing on a particular protocol/interface pair by providing a `Ptr<Ipv4>` and `interface` to an `EnablePcap` method. For example,

```
Ptr<Ipv4> ipv4 = node->GetObject<Ipv4> ();
...
helper.EnablePcapIpv4 ("prefix", ipv4, 0);
```

You can enable pcap tracing on a particular node/net-device pair by providing a `std::string` representing an object name service string to an `EnablePcap` method. The `Ptr<Ipv4>` is looked up from the name string. For example,

```
Names::Add ("serverIPv4" ...);
...
helper.EnablePcapIpv4 ("prefix", "serverIpv4", 1);
```

You can enable pcap tracing on a collection of protocol/interface pairs by providing an `Ipv4InterfaceContainer`. For each `Ipv4` / interface pair in the container the protocol type is checked. For each protocol of the proper type (the same type as is managed by the device helper), tracing is enabled for the corresponding interface. For example,

```
NodeContainer nodes;
...
NetDeviceContainer devices = deviceHelper.Install (nodes);
...
Ipv4AddressHelper ipv4;
ipv4.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces = ipv4.Assign (devices);
...
helper.EnablePcapIpv4 ("prefix", interfaces);
```

You can enable pcap tracing on a collection of protocol/interface pairs by providing a `NodeContainer`. For each `Node` in the `NodeContainer` the appropriate protocol is found. For each protocol, its interfaces are enumerated and tracing is enabled on the resulting pairs. For example,

```
NodeContainer n;
...
helper.EnablePcapIpv4 ("prefix", n);
```

You can enable pcap tracing on the basis of node ID and interface as well. In this case, the node-id is translated to a `Ptr<Node>` and the appropriate protocol is looked up in the node. The resulting protocol and interface are used to specify the resulting trace source.

```
helper.EnablePcapIpv4 ("prefix", 21, 1);
```

Finally, you can enable pcap tracing for all interfaces in the system, with associated protocol being the same type as that managed by the device helper.

```
helper.EnablePcapIpv4All ("prefix");
```

Pcap Tracing Protocol Helper Filename Selection

Implicit in all of the method descriptions above is the construction of the complete filenames by the implementation method. By convention, pcap traces taken for devices in the *ns-3* system are of the form "<prefix>-<node id>-<device id>.pcap". In the case of protocol traces, there is a one-to-one correspondence between protocols and `Nodes`. This is because protocol `Objects` are aggregated to `Node Objects`. Since there is no global protocol id in the system, we use the corresponding node id in file naming. Therefore there is a possibility for file name collisions in automatically chosen trace file names. For this reason, the file name convention is changed for protocol traces.

As previously mentioned, every node in the system will have a system-assigned node id. Since there is a one-to-one correspondence between protocol instances and node instances we use the node id. Each interface has an interface

id relative to its protocol. We use the convention "<prefix>-n<node id>-i<interface id>.pcap" for trace file naming in protocol helpers.

Therefore, by default, a pcap trace file created as a result of enabling tracing on interface 1 of the Ipv4 protocol of node 21 using the prefix "prefix" would be "prefix-n21-i1.pcap".

You can always use the *ns-3* object name service to make this more clear. For example, if you use the object name service to assign the name "serverIpv4" to the Ptr<Ipv4> on node 21, the resulting pcap trace file name will automatically become, "prefix-nserverIpv4-i1.pcap".

Several of the methods have a default parameter called `explicitFilename`. When set to true, this parameter disables the automatic filename completion mechanism and allows you to create an explicit filename. This option is only available in the methods which take a prefix and enable tracing on a single device.

7.4.4 Ascii Tracing Protocol Helpers

The behavior of the ascii trace helpers is substantially similar to the pcap case. Take a look at `src/helper/trace-helper.h` if you want to follow the discussion while looking at real code.

In this section we will be illustrating the methods as applied to the protocol `Ipv4`. To specify traces in similar protocols, just substitute the appropriate type. For example, use a `Ptr<Ipv6>` instead of a `Ptr<Ipv4>` and call `EnableAsciiIpv6` instead of `EnableAsciiIpv4`.

The class `AsciiTraceHelperForIpv4` adds the high level functionality for using ascii tracing to a protocol helper. Each protocol that enables these methods must implement a single virtual method inherited from this class.

```
virtual void EnableAsciiIpv4Internal (Ptr<OutputStreamWrapper> stream,
                                      std::string prefix,
                                      Ptr<Ipv4> ipv4,
                                      uint32_t interface,
                                      bool explicitFilename) = 0;
```

The signature of this method reflects the protocol- and interface-centric view of the situation at this level; and also the fact that the helper may be writing to a shared output stream. All of the public methods inherited from class `PcapAndAsciiTraceHelperForIpv4` reduce to calling this single device- dependent implementation method. For example, the lowest level ascii trace methods,

```
void EnableAsciiIpv4 (std::string prefix, Ptr<Ipv4> ipv4, uint32_t interface, bool explicitFilename =
void EnableAsciiIpv4 (Ptr<OutputStreamWrapper> stream, Ptr<Ipv4> ipv4, uint32_t interface);
```

will call the device implementation of `EnableAsciiIpv4Internal` directly, providing either the prefix or the stream. All other public ascii tracing methods will build on these low-level functions to provide additional user-level functionality. What this means to the user is that all device helpers in the system will have all of the ascii trace methods available; and these methods will all work in the same way across protocols if the protocols implement `EnablAsciiIpv4Internal` correctly.

Ascii Tracing Protocol Helper Methods

```
void EnableAsciiIpv4 (std::string prefix, Ptr<Ipv4> ipv4, uint32_t interface, bool explicitFilename =
void EnableAsciiIpv4 (Ptr<OutputStreamWrapper> stream, Ptr<Ipv4> ipv4, uint32_t interface);

void EnableAsciiIpv4 (std::string prefix, std::string ipv4Name, uint32_t interface, bool explicitFile
void EnableAsciiIpv4 (Ptr<OutputStreamWrapper> stream, std::string ipv4Name, uint32_t interface);

void EnableAsciiIpv4 (std::string prefix, Ipv4InterfaceContainer c);
void EnableAsciiIpv4 (Ptr<OutputStreamWrapper> stream, Ipv4InterfaceContainer c);
```

```
void EnableAsciiIpv4 (std::string prefix, NodeContainer n);
void EnableAsciiIpv4 (Ptr<OutputStreamWrapper> stream, NodeContainer n);

void EnableAsciiIpv4All (std::string prefix);
void EnableAsciiIpv4All (Ptr<OutputStreamWrapper> stream);

void EnableAsciiIpv4 (std::string prefix, uint32_t nodeid, uint32_t deviceid, bool explicitFilename);
void EnableAsciiIpv4 (Ptr<OutputStreamWrapper> stream, uint32_t nodeid, uint32_t interface);
```

You are encouraged to peruse the Doxygen for class `PcapAndAsciiHelperForIpv4` to find the details of these methods; but to summarize ...

There are twice as many methods available for ascii tracing as there were for pcap tracing. This is because, in addition to the pcap-style model where traces from each unique protocol/interface pair are written to a unique file, we support a model in which trace information for many protocol/interface pairs is written to a common file. This means that the <prefix>-n<node id>-<interface> file name generation mechanism is replaced by a mechanism to refer to a common file; and the number of API methods is doubled to allow all combinations.

Just as in pcap tracing, you can enable ascii tracing on a particular protocol/interface pair by providing a `Ptr<Ipv4>` and an `interface` to an `EnableAscii` method. For example,

```
Ptr<Ipv4> ipv4;
...
helper.EnableAsciiIpv4 ("prefix", ipv4, 1);
```

In this case, no trace contexts are written to the ascii trace file since they would be redundant. The system will pick the file name to be created using the same rules as described in the pcap section, except that the file will have the suffix ".tr" instead of ".pcap".

If you want to enable ascii tracing on more than one interface and have all traces sent to a single file, you can do that as well by using an object to refer to a single file. We have already something similar to this in the "cwnd" example above:

```
Ptr<Ipv4> protocol1 = node1->GetObject<Ipv4> ();
Ptr<Ipv4> protocol2 = node2->GetObject<Ipv4> ();
...
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("trace-file-name.tr");
...
helper.EnableAsciiIpv4 (stream, protocol1, 1);
helper.EnableAsciiIpv4 (stream, protocol2, 1);
```

In this case, trace contexts are written to the ascii trace file since they are required to disambiguate traces from the two interfaces. Note that since the user is completely specifying the file name, the string should include the ",tr" for consistency.

You can enable ascii tracing on a particular protocol by providing a `std::string` representing an object name service string to an `EnablePcap` method. The `Ptr<Ipv4>` is looked up from the name string. The `<Node>` in the resulting filenames is implicit since there is a one-to-one correspondence between protocol instances and nodes, For example,

```
Names::Add ("node1Ipv4" ...);
Names::Add ("node2Ipv4" ...);
...
helper.EnableAsciiIpv4 ("prefix", "node1Ipv4", 1);
helper.EnableAsciiIpv4 ("prefix", "node2Ipv4", 1);
```

This would result in two files named "prefix-nnode1Ipv4-i1.tr" and "prefix-nnode2Ipv4-i1.tr" with traces for each interface in the respective trace file. Since all of the EnableAscii functions are overloaded to take a stream wrapper, you can use that form as well:

```
Names::Add ("node1Ipv4" ...);
Names::Add ("node2Ipv4" ...);
...
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("trace-file-name.tr");
...
helper.EnableAsciiIpv4 (stream, "node1Ipv4", 1);
helper.EnableAsciiIpv4 (stream, "node2Ipv4", 1);
```

This would result in a single trace file called "trace-file-name.tr" that contains all of the trace events for both interfaces. The events would be disambiguated by trace context strings.

You can enable ascii tracing on a collection of protocol/interface pairs by providing an `Ipv4InterfaceContainer`. For each protocol of the proper type (the same type as is managed by the device helper), tracing is enabled for the corresponding interface. Again, the `<Node>` is implicit since there is a one-to-one correspondence between each protocol and its node. For example,

```
NodeContainer nodes;
...
NetDeviceContainer devices = deviceHelper.Install (nodes);
...
Ipv4AddressHelper ipv4;
ipv4.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces = ipv4.Assign (devices);
...
...
helper.EnableAsciiIpv4 ("prefix", interfaces);
```

This would result in a number of ascii trace files being created, each of which follows the <prefix>-n<node id>-i<interface>.tr convention. Combining all of the traces into a single file is accomplished similarly to the examples above:

```
NodeContainer nodes;
...
NetDeviceContainer devices = deviceHelper.Install (nodes);
...
Ipv4AddressHelper ipv4;
ipv4.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces = ipv4.Assign (devices);
...
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("trace-file-name.tr");
...
helper.EnableAsciiIpv4 (stream, interfaces);
```

You can enable ascii tracing on a collection of protocol/interface pairs by providing a `NodeContainer`. For each `Node` in the `NodeContainer` the appropriate protocol is found. For each protocol, its interfaces are enumerated and tracing is enabled on the resulting pairs. For example,

```
NodeContainer n;
...
helper.EnableAsciiIpv4 ("prefix", n);
```

This would result in a number of ascii trace files being created, each of which follows the <prefix>-<node id>-<device id>.tr convention. Combining all of the traces into a single file is accomplished similarly to the examples above:

You can enable pcap tracing on the basis of node ID and device ID as well. In this case, the node-id is translated to a `Ptr<Node>` and the appropriate protocol is looked up in the node. The resulting protocol and interface are used to specify the resulting trace source.

```
helper.EnableAsciiIpv4 ("prefix", 21, 1);
```

Of course, the traces can be combined into a single file as shown above.

Finally, you can enable ascii tracing for all interfaces in the system, with associated protocol being the same type as that managed by the device helper.

```
helper.EnableAsciiIpv4All ("prefix");
```

This would result in a number of ascii trace files being created, one for every interface in the system related to a protocol of the type managed by the helper. All of these files will follow the <prefix>-n<node id>-i<interface.tr convention. Combining all of the traces into a single file is accomplished similarly to the examples above.

Ascii Tracing Protocol Helper Filename Selection

Implicit in the prefix-style method descriptions above is the construction of the complete filenames by the implementation method. By convention, ascii traces in the *ns-3* system are of the form "<prefix>-<node id>-<device id>.tr"

As previously mentioned, every node in the system will have a system-assigned node id. Since there is a one-to-one correspondence between protocols and nodes we use to node-id to identify the protocol identity. Every interface on a given protocol will have an interface index (also called simply an interface) relative to its protocol. By default, then, an ascii trace file created as a result of enabling tracing on the first device of node 21, using the prefix "prefix", would be "prefix-n21-i1.tr". Use the prefix to disambiguate multiple protocols per node.

You can always use the *ns-3* object name service to make this more clear. For example, if you use the object name service to assign the name "serverIpv4" to the protocol on node 21, and also specify interface one, the resulting ascii trace file name will automatically become, "prefix-nserverIpv4-1.tr".

Several of the methods have a default parameter called `explicitFilename`. When set to true, this parameter disables the automatic filename completion mechanism and allows you to create an explicit filename. This option is only available in the methods which take a prefix and enable tracing on a single device.

7.5 Summary

ns-3 includes an extremely rich environment allowing users at several levels to customize the kinds of information that can be extracted from simulations.

There are high-level helper functions that allow users to simply control the collection of pre-defined outputs to a fine granularity. There are mid-level helper functions to allow more sophisticated users to customize how information is extracted and saved; and there are low-level core functions to allow expert users to alter the system to present new and previously unexported information in a way that will be immediately accessible to users at higher levels.

This is a very comprehensive system, and we realize that it is a lot to digest, especially for new users or those not intimately familiar with C++ and its idioms. We do consider the tracing system a very important part of *ns-3* and so recommend becoming as familiar as possible with it. It is probably the case that understanding the rest of the *ns-3* system will be quite simple once you have mastered the tracing system

CHAPTER

EIGHT

CONCLUSION

8.1 Futures

This document is a work in process. We hope and expect it to grow over time to cover more and more of the nuts and bolts of *ns-3*.

We hope to add the following chapters over the next few releases:

- The Callback System
- The Object System and Memory Management
- The Routing System
- Adding a New NetDevice and Channel
- Adding a New Protocol
- Working with Real Networks and Hosts

Writing manual and tutorial chapters is not something we all get excited about, but it is very important to the project. If you are an expert in one of these areas, please consider contributing to *ns-3* by providing one of these chapters; or any other chapter you may think is important.

8.2 Closing

ns-3 is a large and complicated system. It is impossible to cover all of the things you will need to know in one small tutorial.

We have really just scratched the surface of *ns-3* in this tutorial, but we hope to have covered enough to get you started doing useful networking research using our favorite simulator.

– The *ns-3* development team.