ns-3 Tutorial

Release ns-3-dev

ns-3 project

Feb 17, 2026

CONTENTS

This is the *ns-3 Tutorial*. Primary documentation for the ns-3 project is organized as follows:

- Several guides that are version controlled for each release (the latest release) and development tree:

    - Tutorial *(this document)*

    - Installation Guide

    - Manual

    - Model Library

    - Contributing Guide

- ns-3 Doxygen: Documentation of the public APIs of the simulator

- 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. Source file column width is 100 columns.

ONE

QUICK START

This section is optional, for readers who want to get *ns-3* up and running as quickly as possible. Readers may skip forward to the *Introduction* chapter, followed by the *Getting Started* chapter, for a lengthier coverage of all of this material.

1.1 Brief Summary

ns-3 is a discrete-event simulator typically run from the command line. It is written directly in C++, not in a high-level modeling language; simulation events are simply C++ function calls, organized by a scheduler.

An *ns-3* user will obtain the *ns-3* source code (see below), compile it into shared (or static) libraries, and link the libraries to *main()* programs that he or she authors. The *main()* program is where the specific simulation scenario configuration is performed and where the simulator is run and stopped. Several example programs are provided, which can be modified or copied to create new simulation scenarios. Users also often edit the *ns-3* library code (and rebuild the libraries) to change its behavior.

ns-3 has optional Python bindings for authoring scenario configuration programs in Python (and using a Python-based workflow); this quick start does not cover those aspects.

1.2 Prerequisites

ns-3 has various optional extensions, but the main features just require a C++ compiler (g++ or clang++), Python (version 3.8 or above), CMake and a build-system (e.g. make, ninja, Xcode). We focus in this chapter only on getting *ns-3* up and running on a system supported by a recent C++ compiler and Python runtime support.

For Linux, use either g++ or clang++ compilers. For macOS, use clang++ (available in Xcode or Xcode Command Line Tools). For Windows, Msys2 tools with the MinGW64 toolchain can be used (since ns-3.37) for most use cases. For releases earlier than ns-3.37, or for use of emulation modes or Python bindings, we recommend to either use a Linux virtual machine, or the Windows Subsystem for Linux.

1.3 Downloading ns-3

ns-3 is distributed in source code only (some binary packages exist but they are not maintained by the open source project). There are two main ways to obtain the source code: 1) downloading the latest release as a source code archive from the main *ns-3* web site, or 2) cloning the Git repository from GitLab.com. These two options are described next; either one or the other download option (but not both) should be followed.

1.3.1 Downloading the Latest Release

1) Download the latest release from https://www.nsnam.org/releases/latest

The latest release is available in two versions, as *ns-3* by itself (as maintained by the open source project) or an `allinone` archive that contains *ns-3* and additional third-party contributed modules. Note that the main *ns-3* release can always be later extended by adding contributing modules of your choice

2) Unpack either archive version in a working directory of your choice.

```
$ tar xjf ns-3.45.tar.bz2
```

or

```
$ tar xjf ns-allinone-3.45.tar.bz2
```

3) Change into the *ns-3* directory directly; e.g.

```
$ cd ns-3.45
```

1.3.2 Cloning ns-3 from GitLab.com

You can perform a Git clone in the usual way:

```
$ git clone https://gitlab.com/nsnam/ns-3-dev.git
```

If you are content to work with the tip of the development tree; you need only to *cd* into ns-3-dev; the *master* branch is checked out by default.

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

If instead you want to try the most recent release (version 3.45 as of this writing), you can checkout a branch corresponding to that git tag:

```
$ git checkout -b ns-3.45-branch ns-3.45
```

1.4 Building and testing ns-3

Once you have obtained the source either by downloading a release or by cloning a Git repository, the next step is to configure the build using the *CMake* build system. The below commands make use of a Python wrapper around CMake, called `ns3`, that simplifies the command-line syntax, resembling *Waf* syntax. There are several options to control the build, but enabling the example programs and the tests, for a default build profile (with asserts enabled and and support for *ns-3* logging) is what is usually done at first:

```
$ ./ns3 configure --enable-examples --enable-tests
```

Then, use `ns3` to build *ns-3*:

```
$ ./ns3 build
```

Once complete, you can run the unit tests to check your build:

```
$ ./test.py
```

All tests should either PASS or be SKIPped. At this point, you have a working *ns-3* simulator. From here, you can start to run programs (look in the examples directory). To run the first tutorial program, whose source code is located at *examples/tutorial/first.cc*, use ns3 to run it (by doing so, the *ns-3* shared libraries are found automatically):

```
$ ./ns3 run first
```

To view possible command-line options, specify the *–PrintHelp* argument:

```
$ ./ns3 run 'first --PrintHelp'
```

To continue reading about the conceptual model and architecture of *ns-3*, the tutorial chapter *Conceptual Overview* would be the next natural place to skip to, or you can learn more about the project and the various build options by continuing directly with the *Introduction* and *Getting Started* chapters.

TWO

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*.

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.

We also provide a quick start guide for those who are comfortable diving right in without too much documentation.

A few key points are worth noting at the onset:

- *ns-3* is open-source, and the project strives to maintain an open environment for researchers to contribute and share their software.

- *ns-3* is not a backwards-compatible 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.

2.1 About ns-3

ns-3 has been developed to provide an open, extensible network simulation platform, for networking research and education. In brief, *ns-3* provides models of how packet data networks work and perform, and provides a simulation engine for users to conduct simulation experiments. Some of the reasons to use *ns-3* include to perform studies that are more difficult or not possible to perform with real systems, to study system behavior in a highly controlled, reproducible environment, and to learn about how networks work. Users will note that the available model set in *ns-3* focuses on modeling how Internet protocols and networks work, but *ns-3* is not limited to Internet systems; several users are using *ns-3* to model non-Internet-based systems.

Many simulation tools exist for network simulation studies. Below are a few distinguishing features of *ns-3* in contrast to other tools.

- *ns-3* is designed as a set of libraries that can be combined together and also with other external software libraries. While some simulation platforms provide users with a single, integrated graphical user interface environment in which all tasks are carried out, *ns-3* is more modular in this regard. Several external animators and data analysis and visualization tools can be used with *ns-3*. However, users should expect to work at the command line and with C++ and/or Python software development tools.

- *ns-3* is primarily used on Linux or macOS systems, although support exists for BSD systems and also for Windows frameworks that can build Linux code, such as Windows Subsystem for Linux, or Cygwin. Native Windows Visual Studio is not presently supported although a developer is working on future support. Windows users may also use a Linux virtual machine.

- *ns-3* is not an officially supported software product of any company. Support for *ns-3* is done on a best-effort basis on the ns-3-users forum (ns-3-users@googlegroups.com).

2.2 For ns-2 Users

For those familiar with *ns-2* (a popular tool that preceded *ns-3*), the most visible outward change when moving to *ns-3* is the choice of scripting language. Programs in *ns-2* are 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. New animators and visualizers are available and under current 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*?" In this author's opinion, unless the user is somehow vested in *ns-2* (either based on existing personal comfort with and knowledge of *ns-2*, or based on a specific simulation model that is only available in *ns-2*), a user will be more productive with *ns-3* for the following reasons:

- *ns-3* is actively maintained with an active, responsive users mailing list, while *ns-2* is only lightly maintained and has not seen significant development in its main code tree for over a decade.

- *ns-3* provides features not available in *ns-2*, such as a implementation code execution environment (allowing users to run real implementation code in the simulator)

- *ns-3* provides a lower base level of abstraction compared with *ns-2*, allowing it to align better with how real systems are put together. Some limitations found in *ns-2* (such as supporting multiple types of interfaces on nodes correctly) have been remedied in *ns-3*.

If in doubt, a good guideline would be to look at both simulators (as well as other simulators), and in particular the models available for your research, but keep in mind that your experience may be better in using the tool that is being actively developed and maintained (*ns-3*).

2.3 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

- An app store

- Contributed Code page, similar to *ns-2*'s popular Contributed Code page

- Documentation on how to contribute

- Use of Git hosting at GitLab.com including issue tracker

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 or comments in the code, etc. are much appreciated. The preferred way to submit patches is either to fork our project on GitLab.com and generate a Merge Request, or to open an issue on our issue tracker and append a patch.

2.4 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.

THREE

RESOURCES

3.1 The Web

There are several important resources of which any *ns-3* user must be aware. The main web site is located at https://www.nsnam.org and provides access to basic information about the *ns-3* system. Detailed documentation is available through the main web site at https://www.nsnam.org/documentation/. 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 https://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 GitLab.com: https://gitlab.com/nsnam/. 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 at the project's old Mercurial site at http://code.nsnam.org.

3.2 Git

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. Until recently, the *ns-3* project used Mercurial as its source code management system, but in December 2018, switch to using Git. Although you do not need to know much about Git in order to complete this tutorial, we recommend becoming familiar with Git and using it to access the source code. GitLab.com provides resources to get started at: https://docs.gitlab.com/ee/tutorials/.

3.3 CMake

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.

The build system CMake is used on the *ns-3* project.

For those interested in the details of CMake, the CMake documents are available at https://cmake.org/cmake/help/latest/index.html and the current code at https://gitlab.kitware.com/cmake/cmake.

3.4 Development Environment

As mentioned above, scripting in *ns-3* is done in C++ or Python. 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 a wide 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.

On Linux, 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 directly. We use CMake for these functions.

On macOS, the toolchain used is Xcode. *ns-3* users on a Mac are strongly encouraged to install Xcode and the command-line tools packages from the Apple App Store, and to look at the *ns-3* installation guide for more information (https://www.nsnam.org/docs/installation/html/).

Typically an *ns-3* author will work in Linux or a Unix-like environment. For those running under Windows, there do exist environments which simulate the Linux environment to various degrees. The *ns-3* installation guide has information about Windows support (https://www.nsnam.org/docs/installation/html/windows.html).

3.5 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.

GETTING STARTED

This section is aimed at getting a user to a working state starting with a machine that may never have had *ns-3* installed. It covers supported platforms, prerequisites, ways to obtain *ns-3*, ways to build *ns-3*, and ways to verify your build and run simple programs.

4.1 Overview

ns-3 is built as a system of software libraries that work together. User programs can be written that links with (or imports from) these libraries. User programs are written in either the C++ or Python programming languages.

ns-3 is distributed as source code, meaning that the target system needs to have a software development environment to build the libraries first, then build the user program. *ns-3* could in principle be distributed as pre-built libraries for selected systems, and in the future it may be distributed that way, but at present, many users actually do their work by editing *ns-3* itself, so having the source code around to rebuild the libraries is useful. If someone would like to undertake the job of making pre-built libraries and packages for operating systems, please contact the ns-developers mailing list.

In the following, we'll look at three ways of downloading and building *ns-3*. The first is to download and build an official release from the main web site. The second is to fetch and build development copies of a basic *ns-3* installation. The third is to use an additional build tool to download more extensions for *ns-3*. We'll walk through each since the tools involved are slightly different.

Experienced Linux users may wonder at this point why *ns-3* is not provided like most other libraries using a package management tool? Although there exist some binary packages for various Linux distributions (e.g. Debian), most users end up editing and having to rebuild the *ns-3* libraries themselves, so having the source code available is more convenient. We will therefore focus on a source installation in this tutorial.

For most uses of *ns-3*, root permissions are not needed, and the use of a non-privileged user account is recommended.

4.2 Prerequisites

The entire set of available *ns-3* libraries has a number of dependencies on third-party libraries, but most of *ns-3* can be built and used with support for a few common (often installed by default) components: a C++ compiler, an installation of Python, a source code editor (such as vim, emacs, or Eclipse) and, if using the development repositories, an installation of Git source code control system. Most beginning users need not concern themselves if their configuration reports some missing optional features of *ns-3*, but for those wishing a full installation, the project provides an installation guide for various systems, available at https://www.nsnam.org/docs/installation/html/index.html.

As of the most recent *ns-3* release (ns-3.44), the following tools are needed to get started with *ns-3*:

Prerequisite	Package/version
C++ compiler	`clang++` or `g++` (g++ version 9 or greater)
Python	`python3` version >=3.8
CMake	`cmake` version >=3.13
Build system	`make`, `ninja`, `xcodebuild` (XCode)
Git	any recent version (to access *ns-3* from GitLab.com)
tar	any recent version (to unpack an ns-3 release)
bunzip2	any recent version (to uncompress an *ns-3* release)

To check the default version of Python, type `python -V`. To check the default version of g++, type `g++ -v`. If your installation is missing or too old, please consult the *ns-3* installation guide for guidance.

From this point forward, we are going to assume that the reader is working in Linux, macOS, or a Linux emulation environment, and has at least the above prerequisites.

For example, do not use a directory path such as the below, because one of the parent directories contains a space in the directory name:

```
$ pwd
/home/user/5G simulations/ns-3-dev
```

4.2.1 Downloading a release of ns-3 as a source archive

This option is for the new user who wishes to download and experiment with the most recently released and packaged version of *ns-3*. *ns-3* publishes its releases as compressed source archives, sometimes referred to as a tarball. A tarball is a particular format of software archive where multiple files are bundled together and the archive is usually compressed. The process for downloading *ns-3* via tarball is simple; you just have to pick a release, download it and uncompress it.

We recommend downloading the most recent release (highest release number). Prior to the ns-3.45 release, we offered one downloadable source code archive, called `ns-allinone-3.nn.tar.bz2` (where nn stands for the release number), which contained the network animator NetAnim, the bake build tool and the ns-3 release. Starting with ns-3.45, we offer two release archives:

1. `ns-3.45.tar.bz2`: Contains only ns-3

2. `ns-allinone-3.45.tar.bz2`: Contains ns-3 plus some compatible extension modules available in the ns-3 App Store

We will focus on the first option above. The instructions for the second one are basically the same, except that you will need to change into the `ns-3.nn` directory after unpacking it.

Let's assume that you, as a user, wish to build *ns-3* in a local directory called `workspace`. If you adopt the `workspace` 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 workspace
$ cd workspace
$ wget https://www.nsnam.org/release/ns-3.45.tar.bz2
$ tar xjf ns-3.45.tar.bz2
```

Notice the use above of the `wget` utility, which is a command-line tool to fetch objects from the web; if you do not have this installed, you can use a browser for this step.

Following these steps, you should change into the directory `ns-3.45`. You are now ready to build the base *ns-3* distribution and may skip ahead to the section on building *ns-3*.

4.3 Downloading ns-3 using Git

The *ns-3* code is available in Git repositories on the GitLab.com service at https://gitlab.com/nsnam/. The group name `nsnam` organizes the various repositories used by the open source project.

The simplest way to get started using Git repositories is to fork or clone the `ns-3-dev` repository. If you are new to Git, the terminology of `fork` and `clone` may be foreign to you; if so, we recommend that you simply `clone` (create your own replica) of the repository found on GitLab.com, as follows:

```
$ cd
$ mkdir workspace
$ cd workspace
$ git clone https://gitlab.com/nsnam/ns-3-dev.git
$ cd ns-3-dev
```

4.4 Building ns-3

ns-3 is built with a build tool called `CMake`, described below. Most users will end up working most directly with the ns3 command-line wrapper for CMake, for the sake of convenience.

4.4.1 Building with the ns3 CMake wrapper

The build process consists of two steps: a configuration stage, and then the actual compilation. The configuration stage is used to enable and disable compiler optimizations and other options and to control the scope of the build. The project will be configured by default using the `default` build profile, which is an optimized build with debug information (CMAKE_BUILD_TYPE=relwithdebinfo) version. Let's tell the project to make an optimized build.

To maintain a similar interface for command-line users, we include a wrapper script for CMake, *ns-3*. To tell *ns-3* that it should do optimized builds that include the examples and tests, you will need to execute the following commands:

```
$ ./ns3 clean
$ ./ns3 configure --build-profile=optimized --enable-examples --enable-tests
```

This runs CMake out of the local directory (which is provided as a convenience for you). The first command to clean out the previous build is not typically strictly necessary but is good practice (but see *Build Profiles*, below); it will remove the previously built libraries and object files found in directory `build/`. When the project is reconfigured and the build system checks for various dependencies, you should see output that looks similar to the following:

```
$ ./ns3 configure --build-profile=optimized --enable-examples --enable-tests
-- CCache is enabled. Precompiled headers are disabled by default.
-- The CXX compiler identification is GNU 11.2.0
-- The C compiler identification is GNU 11.2.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
```

(continues on next page)

```
-- Detecting C compile features - done
-- Using default output directory /mnt/dev/tools/source/ns-3-dev/build
-- Found GTK3_GTK: /usr/lib/x86_64-linux-gnu/libgtk-3.so
-- GTK3 was found.
-- LibXML2 was found.
-- LibRT was found.
-- Visualizer requires Python bindings
-- Found Boost: /usr/lib/x86_64-linux-gnu/cmake/Boost-1.74.0/BoostConfig.cmake (found␣
↪version "1.74.0")
-- Found PkgConfig: /usr/bin/pkg-config (found version "0.29.2")
-- GSL was found.
-- Found Sphinx: /usr/bin/sphinx-build
-- Looking for sys/types.h
-- Looking for sys/types.h - found
-- Looking for stdint.h
-- Looking for stdint.h - found
-- Looking for stddef.h
-- Looking for stddef.h - found
-- Check size of long long
-- Check size of long long - done
-- Check size of int128_t
-- Check size of int128_t - failed
-- Check size of __int128_t
-- Check size of __int128_t - done
-- Performing Test has_hash___int128_t
-- Performing Test has_hash___int128_t - Success
-- Check size of unsigned long long
-- Check size of unsigned long long - done
-- Check size of uint128_t
-- Check size of uint128_t - failed
-- Check size of __uint128_t
-- Check size of __uint128_t - done
-- Performing Test has_hash___uint128_t
-- Performing Test has_hash___uint128_t - Success
-- Looking for C++ include inttypes.h
-- Looking for C++ include inttypes.h - found
-- Looking for C++ include stat.h
-- Looking for C++ include stat.h - not found
-- Looking for C++ include dirent.h
-- Looking for C++ include dirent.h - found
-- Looking for C++ include stdlib.h
-- Looking for C++ include stdlib.h - found
-- Looking for C++ include signal.h
-- Looking for C++ include signal.h - found
-- Looking for C++ include netpacket/packet.h
-- Looking for C++ include netpacket/packet.h - found
-- Looking for getenv
-- Looking for getenv - found
-- Processing src/antenna
-- Processing src/aodv
-- Processing src/applications
-- Processing src/bridge
```

```
-- Processing src/brite
-- Brite was not found
-- Processing src/buildings
-- Processing src/click
-- Click was not found
-- Processing src/config-store
-- Processing src/core
-- Looking for C++ include boost/units/quantity.hpp
-- Looking for C++ include boost/units/quantity.hpp - found
-- Looking for C++ include boost/units/systems/si.hpp
-- Looking for C++ include boost/units/systems/si.hpp - found
-- Boost Units have been found.
-- Processing src/csma
-- Processing src/csma-layout
-- Processing src/dsdv
-- Processing src/dsr
-- Processing src/energy
-- Processing src/fd-net-device
-- Looking for C++ include net/ethernet.h
-- Looking for C++ include net/ethernet.h - found
-- Looking for C++ include netpacket/packet.h
-- Looking for C++ include netpacket/packet.h - found
-- Looking for C++ include net/if.h
-- Looking for C++ include net/if.h - found
-- Looking for C++ include linux/if_tun.h
-- Looking for C++ include linux/if_tun.h - found
-- Looking for C++ include net/netmap_user.h
-- Looking for C++ include net/netmap_user.h - not found
-- Looking for C++ include sys/ioctl.h
-- Looking for C++ include sys/ioctl.h - found
-- Checking for module 'libdpdk'
--   No package 'libdpdk' found
-- Processing src/flow-monitor
-- Processing src/internet
-- Processing src/internet-apps
-- Processing src/lr-wpan
-- Processing src/lte
-- Processing src/mesh
-- Processing src/mobility
-- Processing src/netanim
-- Processing src/network
-- Processing src/nix-vector-routing
-- Processing src/olsr
-- Processing src/openflow
-- Openflow was not found
-- Processing src/point-to-point
-- Processing src/point-to-point-layout
-- Processing src/propagation
-- Processing src/sixlowpan
-- Processing src/spectrum
-- Processing src/stats
-- Processing src/tap-bridge
```

```
-- Processing src/test
-- Processing src/topology-read
-- Processing src/traffic-control
-- Processing src/uan
-- Processing src/virtual-net-device
-- Processing src/wifi
-- ---- Summary of optional NS-3 features:
Build profile               : optimized
Build directory             : /mnt/dev/tools/source/ns-3-dev/build
BRITE Integration           : OFF (missing dependency)
DES Metrics event collection  : OFF (not requested)
DPDK NetDevice              : OFF (missing dependency)
Emulation FdNetDevice       : ON
Examples                    : ON
File descriptor NetDevice   : ON
GNU Scientific Library (GSL)  : ON
GtkConfigStore              : ON
MPI Support                 : OFF (not requested)
NS-3 Click Integration      : OFF (missing dependency)
NS-3 OpenFlow Integration   : OFF (missing dependency)
Netmap emulation FdNetDevice  : OFF (missing dependency)
PyViz visualizer            : OFF (missing dependency)
Python Bindings             : OFF (not requested)
Real Time Simulator         : ON
SQLite stats support        : ON
Tap Bridge                  : ON
Tap FdNetDevice             : ON
Tests                       : ON


Modules configured to be built:
antenna                 aodv                    applications
bridge                  buildings               config-store
core                    csma                    csma-layout
dsdv                    dsr                     energy
fd-net-device           flow-monitor            internet
internet-apps           lr-wpan                 lte
mesh                    mobility                netanim
network                 nix-vector-routing      olsr
point-to-point          point-to-point-layout   propagation
sixlowpan               spectrum                stats
tap-bridge              test                    topology-read
traffic-control         uan                     virtual-net-device
wifi


Modules that cannot be built:
brite                   click                   mpi
openflow                visualizer


-- Configuring done
```

```
-- Generating done
-- Build files have been written to: /mnt/dev/tools/source/ns-3-dev/cmake-cache
Finished executing the following commands:
mkdir cmake-cache
cd cmake-cache; /usr/bin/cmake -DCMAKE_BUILD_TYPE=release -DNS3_NATIVE_OPTIMIZATIONS=ON -
↪DNS3_EXAMPLES=ON -DNS3_TESTS=ON -G Unix Makefiles .. ; cd ..
```

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 (`OFF (not requested)`). Other options might depend on third-party libraries, which if not found will be disabled (`OFF(missing dependency)`). 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." Finally, to reprint this summary of which optional features are enabled, use the `show config` option to ns3.

Now go ahead and switch back to the debug build that includes the examples and tests.

```
$ ./ns3 clean
$ ./ns3 configure --build-profile=debug --enable-examples --enable-tests
```

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

```
$ ./ns3 build
```

Although the above steps made you build the *ns-3* part of the system twice, now you know how to change the configuration and build optimized code.

A command exists for checking which profile is currently active for an already configured project:

```
$ ./ns3 show profile
Build profile: debug
```

Here are a few more introductory tips about CMake.

Handling build errors

ns-3 releases are tested against the most recent C++ compilers available in the mainstream Linux and macOS distributions at the time of the release. However, over time, newer distributions are released, with newer compilers, and these newer compilers tend to be more pedantic about warnings. *ns-3* configures its build to treat all warnings as errors, so it is sometimes the case, if you are using an older release version on a newer system, that a compiler warning will cause the build to fail.

For instance, ns-3.28 was released prior to Fedora 28, which included a new major version of gcc (gcc-8). Building ns-3.28 or older releases on Fedora 28, when GTK+2 is installed, will result in an error such as:

```
/usr/include/gtk-2.0/gtk/gtkfilechooserbutton.h:59:8: error: unnecessary parentheses in
↪declaration of '__gtk_reserved1' [-Werror=parentheses]
 void (*__gtk_reserved1);
```

In releases starting with ns-3.28.1, an option is available in CMake to work around these issues. The option disables the inclusion of the '-Werror' flag to g++ and clang++. The option is '--disable-werror' and must be used at configure time; e.g.:

```
./ns3 configure --disable-werror --enable-examples --enable-tests
```

Configure vs. Build

Some CMake commands are only meaningful during the configure phase and some commands are valid in the build 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 that also includes the examples and tests.

```
$ ./ns3 configure --enable-sudo --enable-examples --enable-tests
```

If you do this, ns3 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 ns3. To explore these options, type:

```
$ ./ns3 --help
```

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

Build Profiles

We already saw how you can configure CMake for `debug` or `optimized` builds:

```
$ ./ns3 configure --build-profile=debug
```

There is also an intermediate build profile, `release`. `-d` is a synonym for `--build-profile`.

The build profile controls the use of logging, assertions, and compiler optimization:

Table 1: Build profiles

Fea-ture	Build Profile			
	debug	default	release	optimized
En-abled Fea-tures	NS3_BUILD_PROFILE_DEBUG NS_LOG... NS_ASSERT...	NS3_BUILD_PROFILE_DEBUG NS_LOG... NS_ASSERT...	NS3_BUILD_PROFILE_RELEASE	NS3_BUILD_PROFILE_OPTIMIZED
Code Wrap-per Macro	NS_BUILD_DEBUG(code)	NS_BUILD_DEBUG(code)	NS_BUILD_RELEASE(code)	NS_BUILD_OPTIMIZED(code)
Com-piler Flags	-Og -g	-Os -g	-O3	-O3 -march=native -mtune=native

As you can see, logging and assertions are only configured by default in debug builds, although they can be selectively enabled in other build profiles by using the `--enable-logs` and `--enable-asserts` flags during CMake configuration time. Recommended practice is to develop your scenario in debug mode, then conduct repetitive runs (for statistics or changing parameters) in optimized build profile.

If you have code that should only run in specific build profiles, use the indicated Code Wrapper macro:

```
NS_BUILD_DEBUG(std::cout << "Part of an output line..." << std::flush; timer.Start());
DoLongInvolvedComputation();
NS_BUILD_DEBUG(timer.Stop(); std::cout << "Done: " << timer << std::endl;)
```

By default ns3 puts the build artifacts in the `build` directory. You can specify a different output directory with the `--out` option, e.g.

```
$ ./ns3 configure --out=my-build-dir
```

Combining this with build profiles lets you switch between the different compile options in a clean way:

```
$ ./ns3 configure --build-profile=debug --out=build/debug
$ ./ns3 build
...
$ ./ns3 configure --build-profile=optimized --out=build/optimized
$ ./ns3 build
...
```

This allows you to work with multiple builds rather than always overwriting the last build. When you switch, ns3 will only compile what it has to, instead of recompiling everything.

When you do switch build profiles like this, you have to be careful to give the same configuration parameters each time. It may be convenient to define some environment variables to help you avoid mistakes:

```
$ export NS3CONFIG="--enable-examples --enable-tests"
$ export NS3DEBUG="--build-profile=debug --out=build/debug"
$ export NS3OPT=="--build-profile=optimized --out=build/optimized"

$ ./ns3 configure $NS3CONFIG $NS3DEBUG
$ ./ns3 build
...
$ ./ns3 configure $NS3CONFIG $NS3OPT
$ ./ns3 build
```

Compilers and Flags

In the examples above, CMake uses the GCC C++ compiler, `g++`, for building *ns-3*. However, it's possible to change the C++ compiler used by CMake by defining the `CXX` environment variable. For example, to use the Clang C++ compiler, `clang++`,

```
$ CXX="clang++" ./ns3 configure
$ ./ns3 build
```

One can also set up ns3 to do distributed compilation with `distcc` in a similar way:

```
$ CXX="distcc g++" ./ns3 configure
$ ./ns3 build
```

More info on `distcc` and distributed compilation can be found on it's project page under Documentation section.

To add compiler flags, use the `CXXFLAGS_EXTRA` environment variable when you configure *ns-3*.

Install

ns3 may be used to install libraries in various places on the system. The default location where libraries and executables are built is in the `build` directory, and because ns3 knows the location of these libraries and executables, it is not necessary to install the libraries elsewhere.

If users choose to install things outside of the build directory, users may issue the `./ns3 install` command. By default, the prefix for installation is `/usr/local`, so `./ns3 install` will install programs into `/usr/local/bin`, libraries into `/usr/local/lib`, and headers into `/usr/local/include`. Superuser privileges are typically needed to install to the default prefix, so the typical command would be `sudo ./ns3 install`. When running programs with ns3, ns3 will first prefer to use shared libraries in the build directory, then will look for libraries in the library path configured in the local environment. So when installing libraries to the system, it is good practice to check that the intended libraries are being used.

Users may choose to install to a different prefix by passing the `--prefix` option at configure time, such as:

```
./ns3 configure --prefix=/opt/local
```

If later after the build the user issues the `./ns3 install` command, the prefix `/opt/local` will be used.

The `./ns3 clean` command should be used prior to reconfiguring the project if ns3 will be used to install things at a different prefix.

In summary, it is not necessary to call `./ns3 install` to use *ns-3*. Most users will not need this command since ns3 will pick up the current libraries from the `build` directory, but some users may find it useful if their use case involves working with programs outside of the *ns-3* directory.

Clean

Cleaning refers to the removal of artifacts (e.g. files) generated or edited by the build process. There are different levels of cleaning possible:

Scope	Command	Description
clean	*./ns3 clean*	Remove artifacts generated by the CMake configuration and the build
distclean	*./ns3 distclean*	Remove artifacts from the configuration, build, documentation, test and Python
ccache	*ccache -C*	Remove all compiled artifacts from the ccache

clean can be used if the focus is on reconfiguring the way that ns-3 is presently being compiled. *distclean* can be used if the focus is on restoring the ns-3 directory to an original state.

The ccache lies outside of the ns-3 directory (typically in a hidden directory at *~/.cache/ccache*) and is shared across projects. Users should be aware that cleaning the ccache will cause cache misses on other build directories outside of the current working directory. Cleaning this cache periodically may be helpful to reclaim disk space. Cleaning the ccache is completely separate from cleaning any files within the ns-3 directory.

Because clean operations involve removing files, the option conservatively refuses to remove files if one of the deleted files or directories lies outside of the current working directory. Users may wish to precede the actual clean with a *–dry-run*, when in doubt about what the clean command will do, because a dry run will print the warning if one exists. For example:

```
./ns3 clean --dry-run
./ns3 clean
```

One ns3

There is only one ns3 script, at the top level of the *ns-3* source tree. As you work, you may find yourself spending a lot of time in `scratch/`, or deep in `src/...`, and needing to invoke ns3. You could just remember where you are, and invoke ns3 like this:

```
$ ../../../ns3 ...
```

but that gets tedious, and error prone, and there are better solutions.

One common way when using a text-based editor such as emacs or vim is to open two terminal sessions and use one to build *ns-3* and one to edit source code.

If you only have the tarball, an environment variable can help:

```
$ export NS3DIR="$PWD"
$ function ns3f { cd $NS3DIR && ./ns3 $* ; }

$ cd scratch
$ ns3f build
```

It might be tempting in a module directory to add a trivial `ns3` script along the lines of `exec ../../ns3`. Please don't. It's confusing to newcomers, and when done poorly it leads to subtle build errors. The solutions above are the way to go.

4.4.2 Building with CMake

The ns3 wrapper script calls CMake directly, mapping Waf-like options to the verbose settings used by CMake. Calling `./ns3` will execute a series of commands, that will be shown at the end of their execution. The execution of those underlying commands can be skipped to just print them using `./ns3 --dry-run`.

Here is are a few examples showing why we suggest the use of the ns3 wrapper script.

Configuration command

```
$ ./ns3 configure --enable-tests --enable-examples -d optimized
```

Corresponds to

```
$ cd /ns-3-dev/cmake-cache/
$ cmake -DCMAKE_BUILD_TYPE=release -DNS3_NATIVE_OPTIMIZATIONS=ON -DNS3_ASSERT=OFF -DNS3_
→LOG=OFF -DNS3_TESTS=ON -DNS3_EXAMPLES=ON ..
```

Build command

To build a specific target such as `test-runner` we use the following ns3 command:

```
$ ./ns3 build test-runner
```

Which corresponds to the following commands:

```
$ cd /ns-3-dev/cmake-cache/
$ cmake --build . -j 16 --target test-runner # This command builds the test-runner␣
↪target with the underlying build system
```

To build all targets such as modules, examples and tests, we use the following ns3 command:

```
$ ./ns3 build
```

Which corresponds to:

```
$ cd /ns-3-dev/cmake-cache/
$ cmake --build . -j 16  # This command builds all the targets with the underlying build␣
↪system
```

Run command

```
$ ./ns3 run test-runner
```

Corresponds to:

```
$ cd /ns-3-dev/cmake-cache/
$ cmake --build . -j 16 --target test-runner # This command builds the test-runner␣
↪target calling the underlying build system
$ export PATH=$PATH:/ns-3-dev/build/:/ns-3-dev/build/lib:/ns-3-dev/build/bindings/python␣
↪# export library paths
$ export LD_LIBRARY_PATH=/ns-3-dev/build/:/ns-3-dev/build/lib:/ns-3-dev/build/bindings/
↪python
$ export PYTHON_PATH=/ns-3-dev/build/:/ns-3-dev/build/lib:/ns-3-dev/build/bindings/python
$ /ns-3-dev/build/utils/ns3-dev-test-runner-optimized # call the executable with the␣
↪real path
```

Note: the command above would fail if `./ns3 build` was not executed first, since the examples won't be built by the test-runner target.

On Windows, the Msys2/MinGW64/bin directory path must be on the PATH environment variable, otherwise the DLL's required by the C++ runtime will not be found, resulting in crashes without any explicit reasoning.

Note: The ns-3 script adds only the ns-3 lib directory path to the PATH, ensuring the ns-3 DLLs will be found by running programs. If you are using CMake directly or an IDE, make sure to also include the path to ns-3-dev/build/lib in the PATH variable.

If you are using one of Windows's terminals (CMD, PowerShell or Terminal), you can use the setx command to change environment variables permanently or *set* to set them temporarily for that shell:

```
C:\\Windows\\system32>echo %PATH%
C:\\Windows\\system32;C:\\Windows;D:\\tools\\msys64\\mingw64\\bin;

C:\\Windows\\system32>setx PATH "%PATH%;D:\\tools\\msys64\\usr\\bin;" /m

C:\\Windows\\system32>echo %PATH%
C:\\Windows\\system32;C:\\Windows;D:\\tools\\msys64\\mingw64\\bin;
D:\\tools\\msys64\\usr\\bin;
```

Note: running on an administrator terminal will change the system PATH, while the user terminal will change the user PATH, unless the */m* flag is added.

4.4.3 Building with IDEs

With CMake, IDE integration is much easier. We list the steps on how to use ns-3 with a few IDEs.

Microsoft Visual Studio Code

Start by downloading VS Code.

Then install it and then install the CMake and C++ plugins.

This can be done accessing the extensions' menu button on the left.
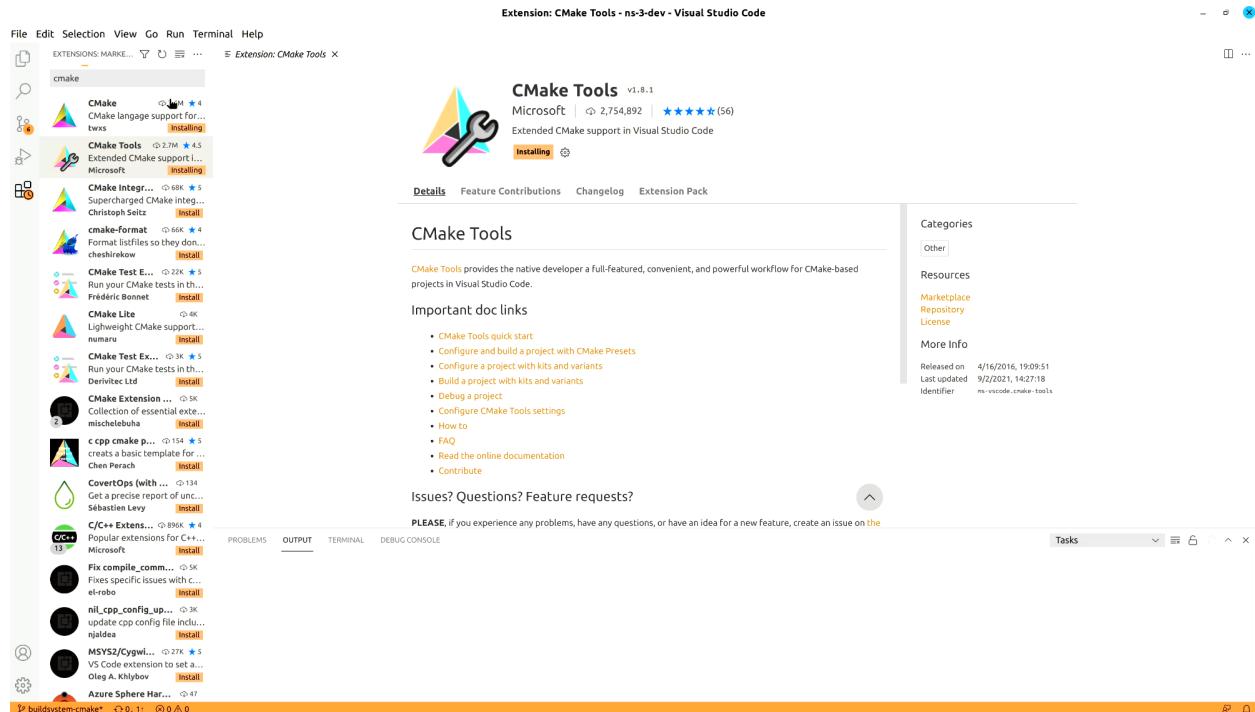
It will take a while, but it will locate the available toolchains for you to use.

After that, open the ns-3-dev folder. It should run CMake automatically and preconfigure it.

After this happens, you can choose ns-3 features by opening the CMake cache and toggling them on or off.

Just as an example, here is how to enable examples

After saving the cache, CMake will run, refreshing the cache. Then VsCode will update its list of targets on the left side of the screen in the CMake menu.

After selecting a target on the left side menu, there are options to build, run or debug it.

Any of them will automatically build the selected target. If you choose run or debug, the executable targets will be executed. You can open the source files you want, put some breakpoints and then click debug to visually debug programs.

Note: If you are running on Windows, you need to manually add your ns-3 library directory to the `PATH` environment variable. This can be accomplished in two ways.

The first, is to set VSCode's `settings.json` file to include the following:
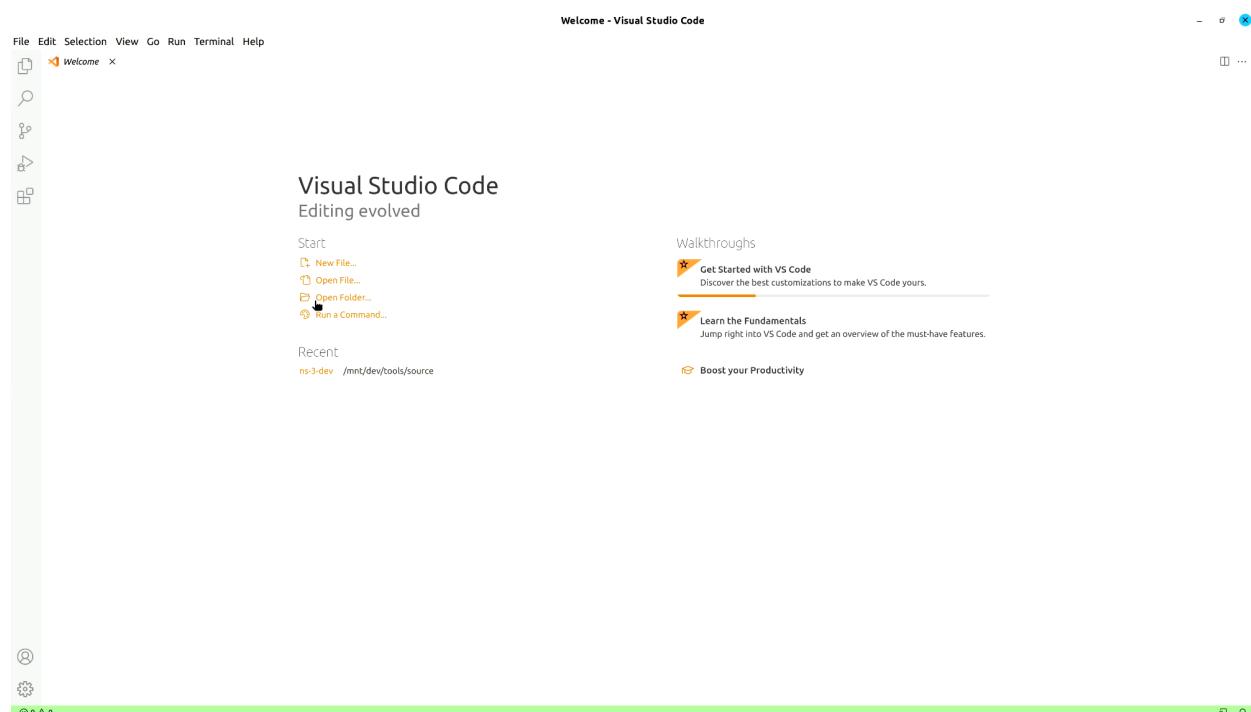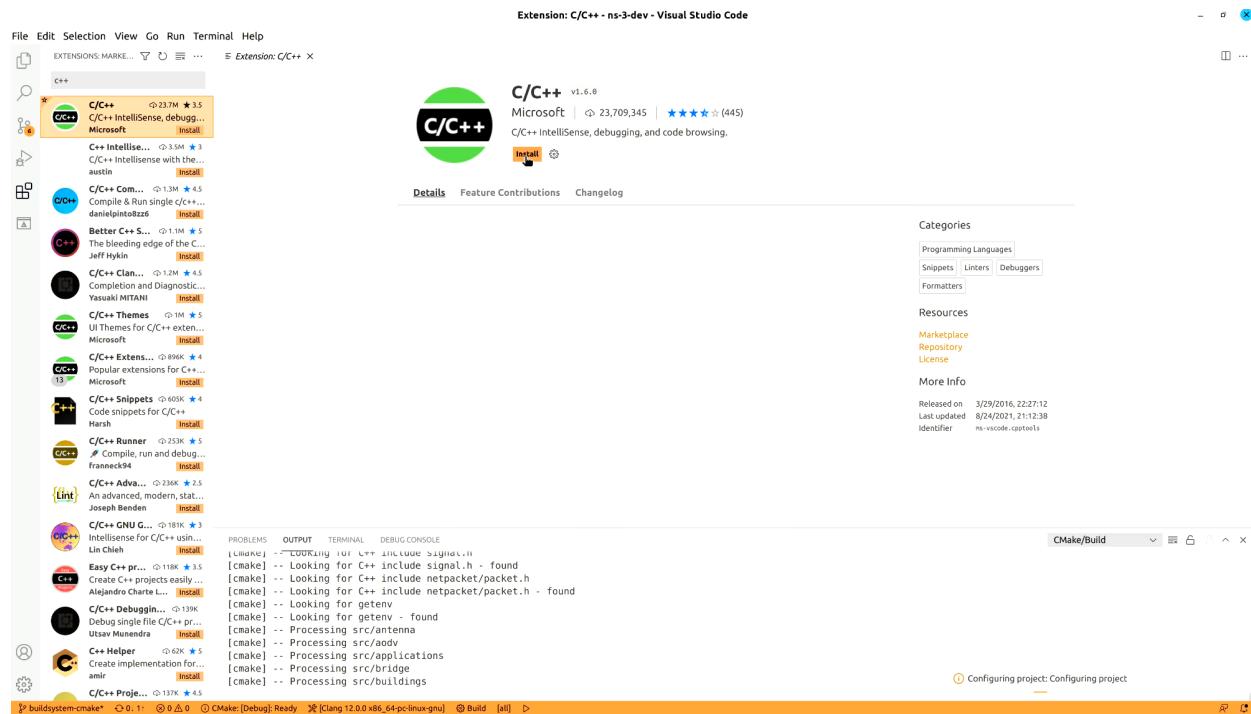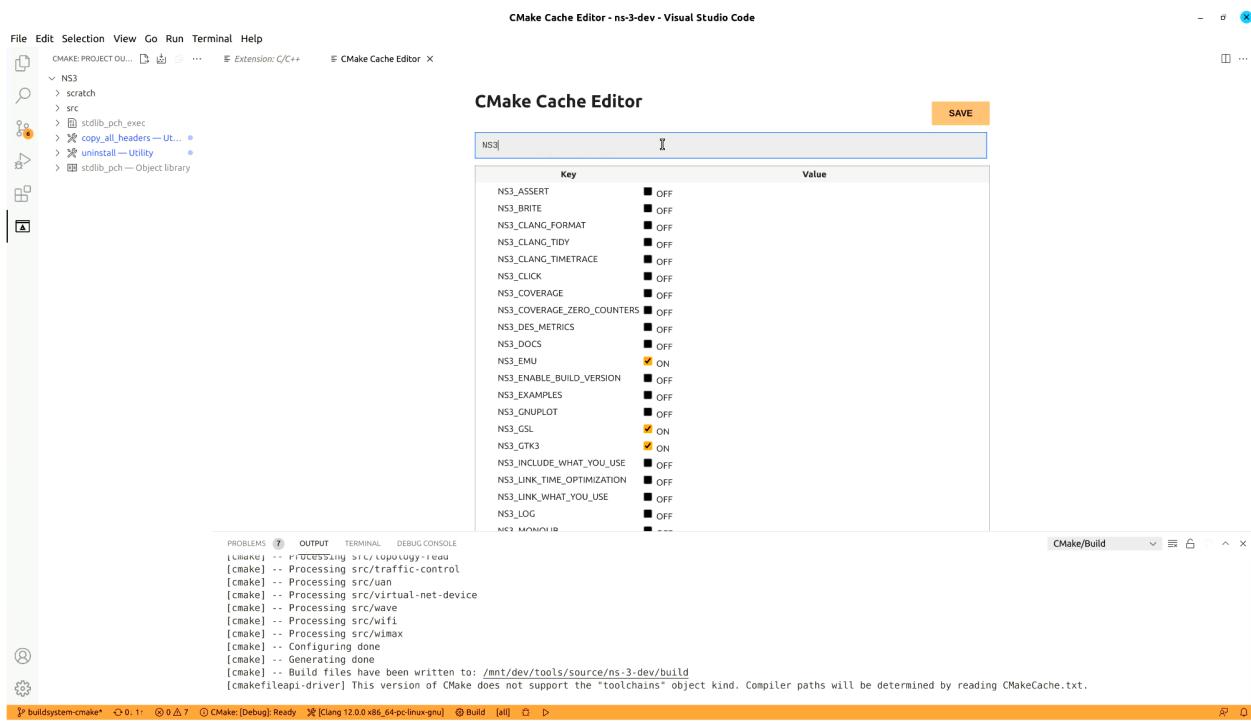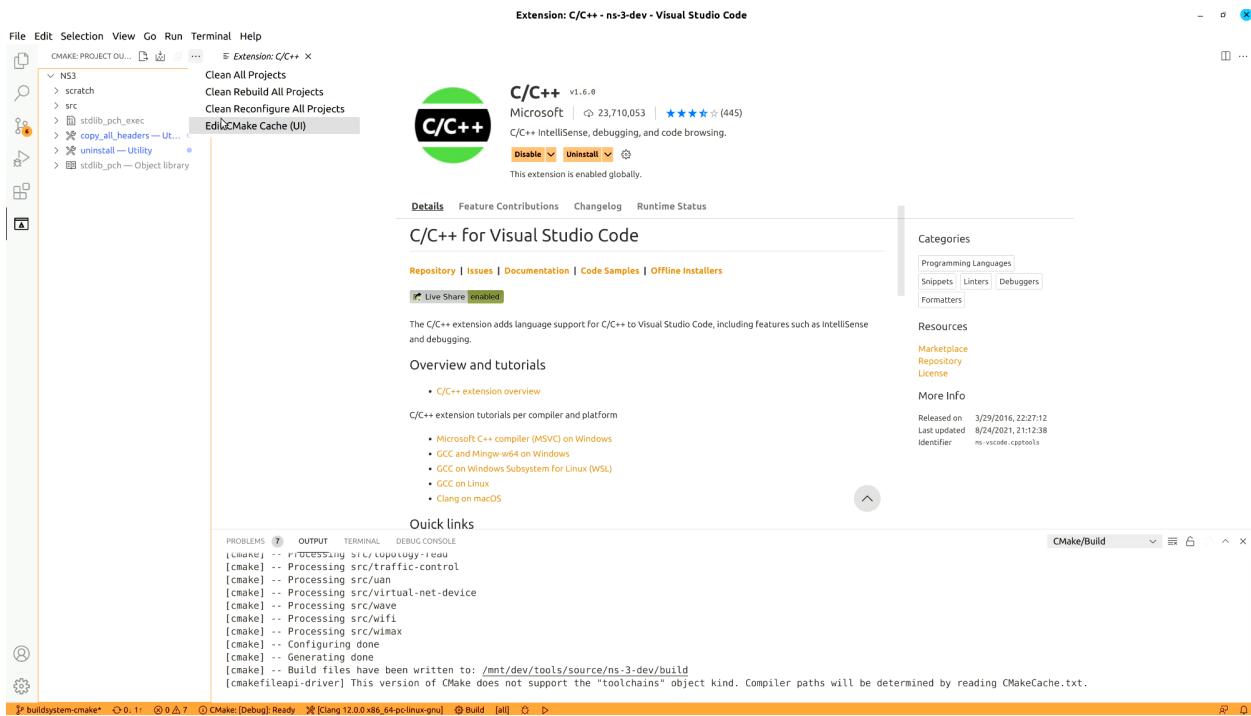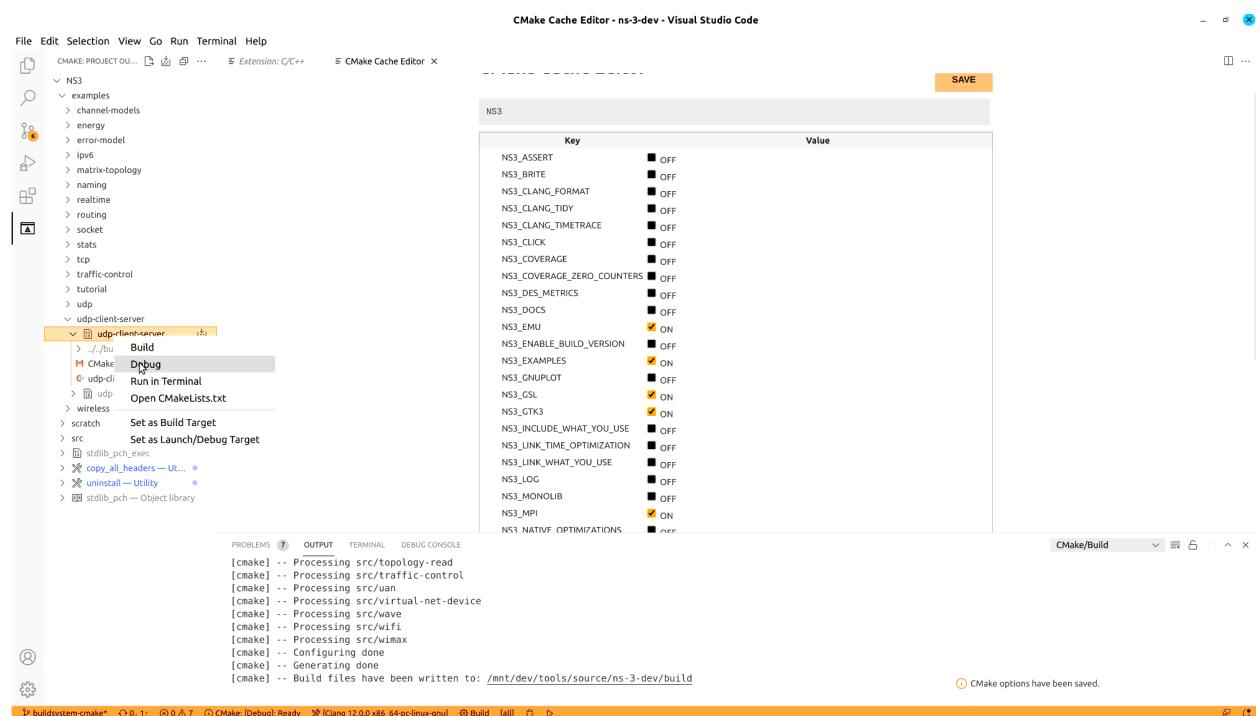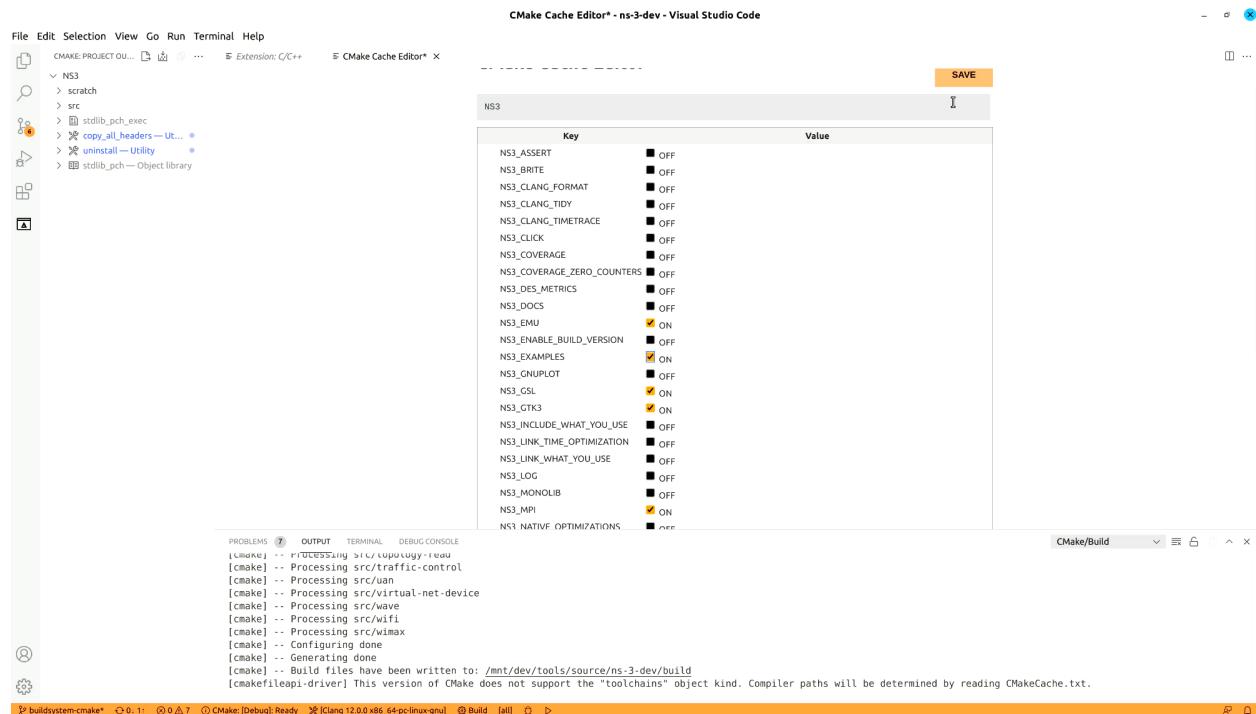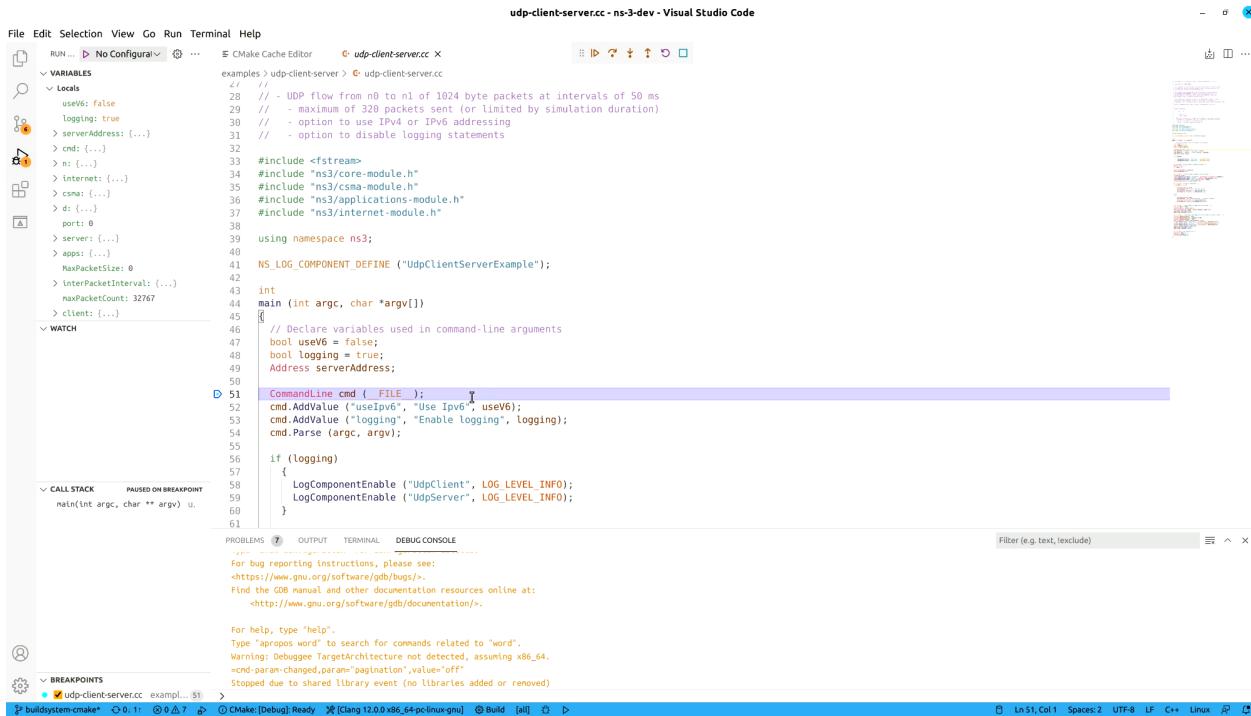
```
"cmake.environment": {
      "PATH": "${env:PATH};${workspaceFolder}/build/lib"
}
```

The second, a more permanent solution, with the following command:

```
> echo %PATH%
C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\system32;C:\Windows;C:\Windows\
↪System32\Wbem;
C:\Windows\System32\OpenSSH\;C:\Program Files\dotnet\;C:\Program Files\PuTTY\;C:\Program↵
↪Files\VSCodium\bin;
C:\Program Files\Meld\;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\system32;
↪C:\Windows;
C:\Windows\System32\Wbem;C:\Windows\System32\OpenSSH\;C:\Program Files\dotnet\;C:\
↪Program Files\PuTTY\;
C:\Program Files\VSCodium\bin;C:\Program Files\Meld\;C:\Users\username\AppData\Local\
↪Microsoft\WindowsApps;

> setx PATH "%PATH%;C:\path\to\ns-3-dev\build\lib"

SUCCESS: Specified value was saved.

> echo %PATH%
C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\system32;C:\Windows;C:\Windows\
↪System32\Wbem;
...
C:\Program Files\VSCodium\bin;C:\Program Files\Meld\;C:\Users\username\AppData\Local\
↪Microsoft\WindowsApps;
C:\tools\source\ns-3-dev\build\lib;
```

If you do not setup your `PATH` environment variable, you may end up having problems to debug that look like the following:

```
=thread-group-added,id="i1"
GNU gdb (GDB) 14.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
...
ERROR: Unable to start debugging. GDB exited unexpectedly.
The program 'C:\tools\source\ns-3-dev\build\examples\wireless\ns3-dev-wifi-he-network-
→debug.exe' has exited with code 0 (0x00000000).

ERROR: During startup program exited with code 0xc0000135.
```

Or

```
=thread-group-added,id="i1"
GNU gdb (GDB) 14.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
...
ERROR: Unable to start debugging. Unexpected GDB output from command "-exec-run". During␣
→startup program exited with code 0xc0000135.
The program 'C:\tools\source\ns-3-dev\build\examples\wireless\ns3-dev-wifi-he-network-
→debug.exe' has exited with code 0 (0x00000000).
```

JetBrains CLion

Start by downloading CLion.

The following image contains the toolchain configuration window for CLion running on Windows (only WSLv2 is currently supported).

CLion uses Makefiles for your platform as the default generator. Here you can choose a better generator like *ninja* by setting the cmake options flag to *-G Ninja*. You can also set options to enable examples (*-DNS3_EXAMPLES=ON*) and tests (*-DNS3_TESTS=ON*).

To refresh the CMake cache, triggering the discovery of new targets (libraries, executables and/or modules), you can either configure to re-run CMake automatically after editing CMake files (pretty slow and easily triggered) or reload it manually. The following image shows how to trigger the CMake cache refresh.

After configuring the project, the available targets are listed in a drop-down list on the top right corner. Select the target you want and then click the hammer symbol to build, as shown in the image below.

If you have selected and executable target, you can click either the play button to execute the program; the bug to debug the program; the play button with a chip, to run Valgrind and analyze memory usage, leaks and so on.
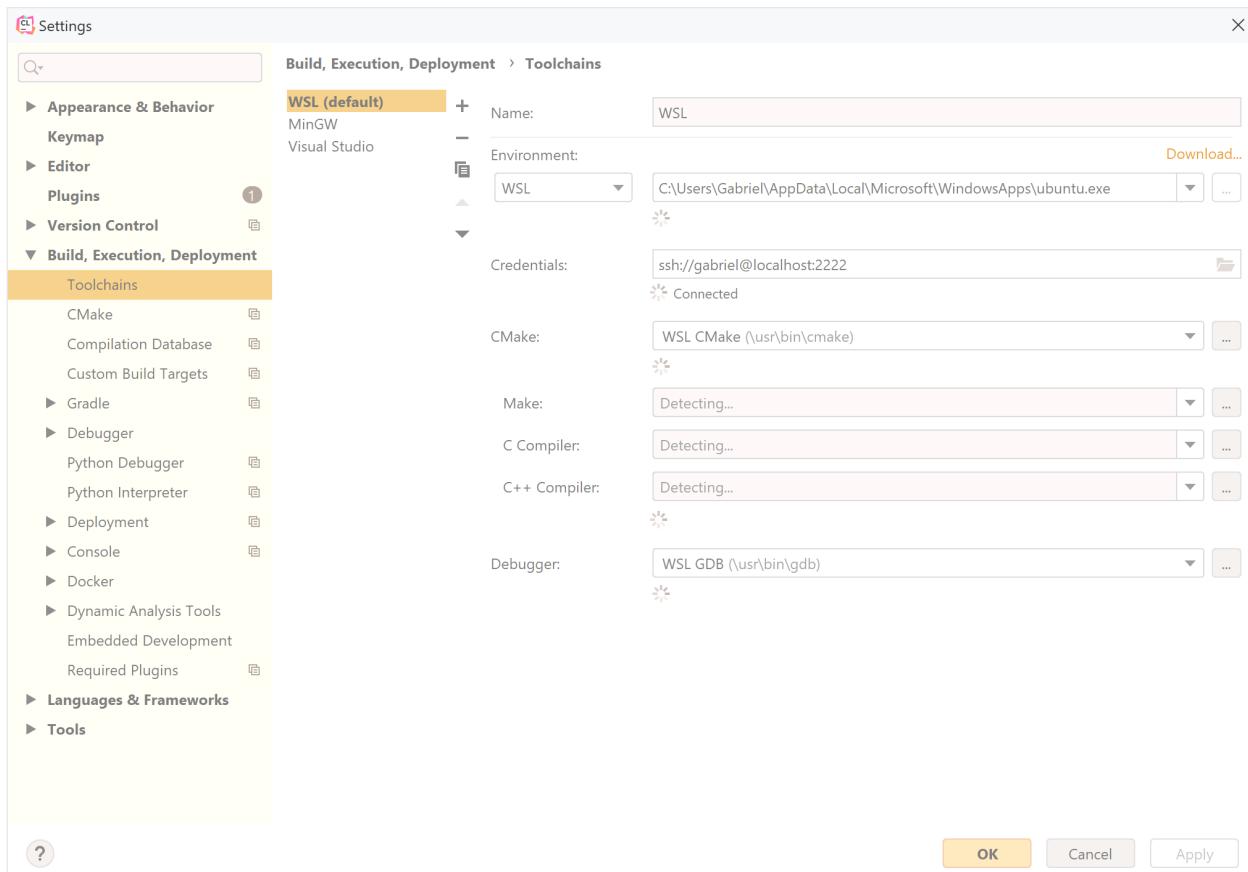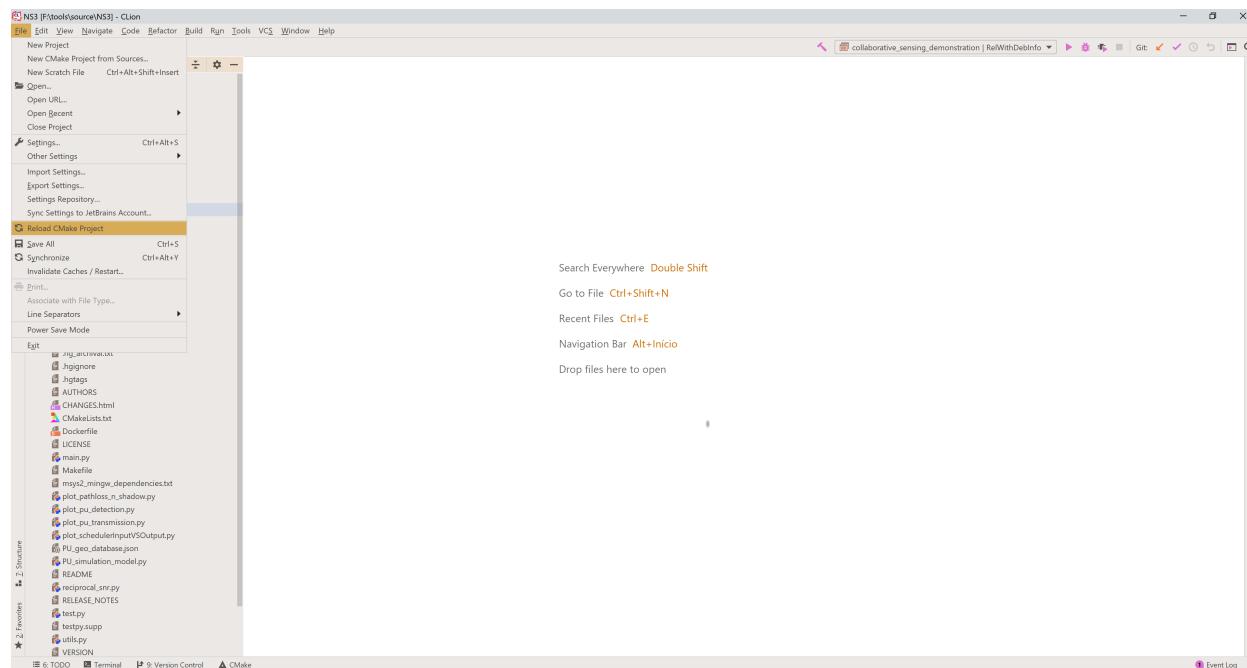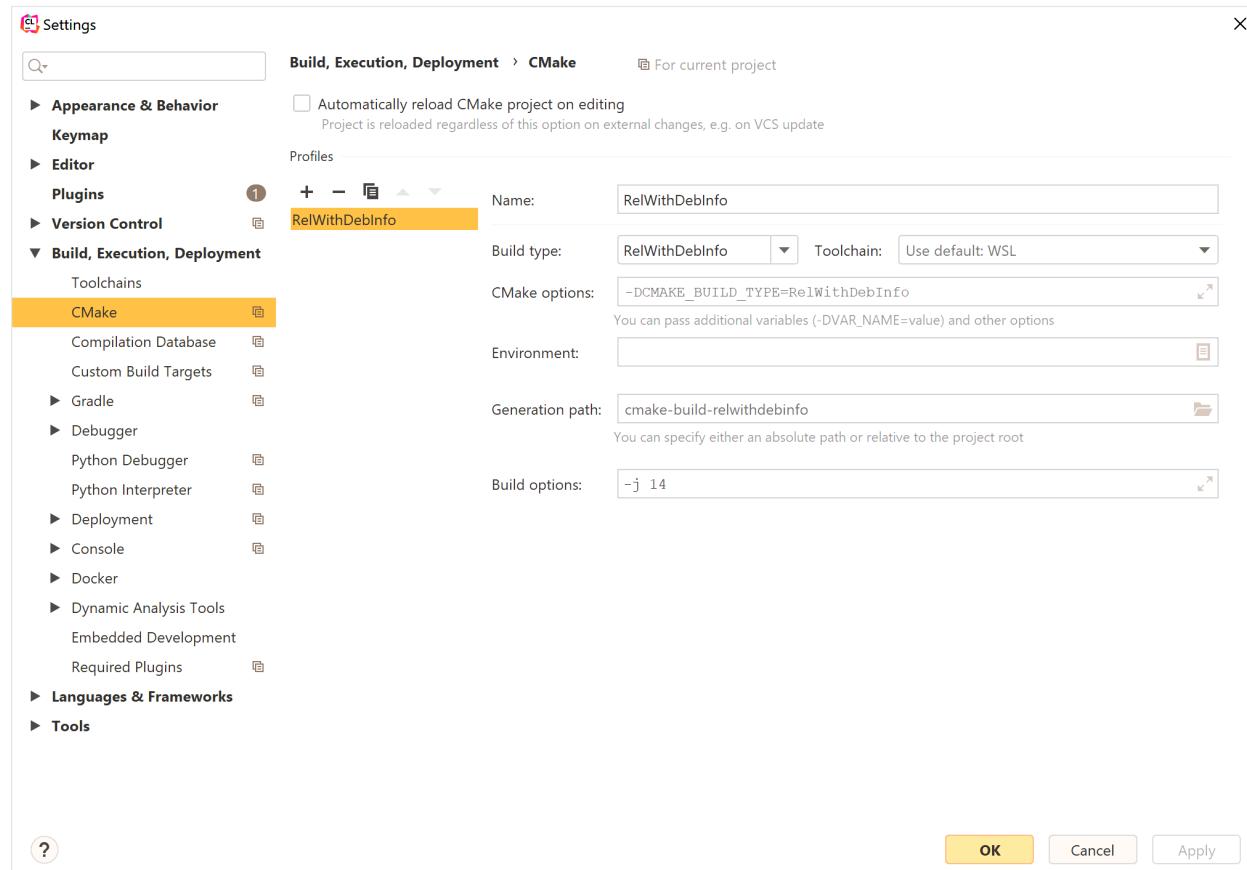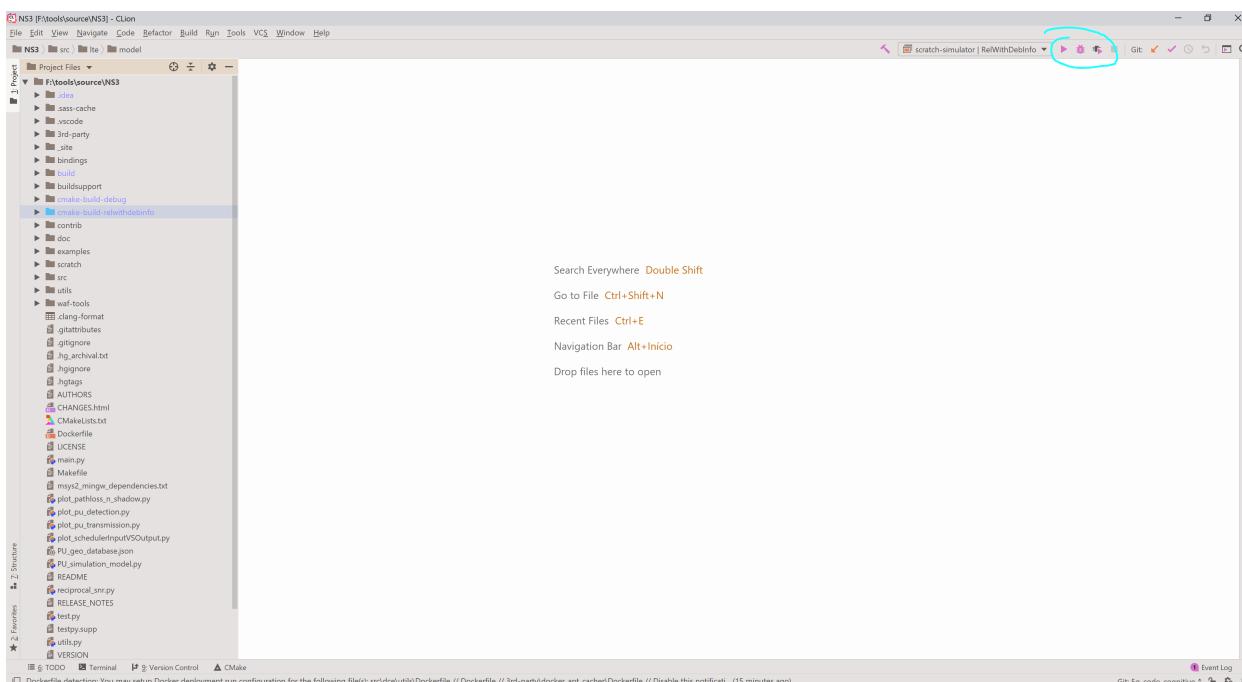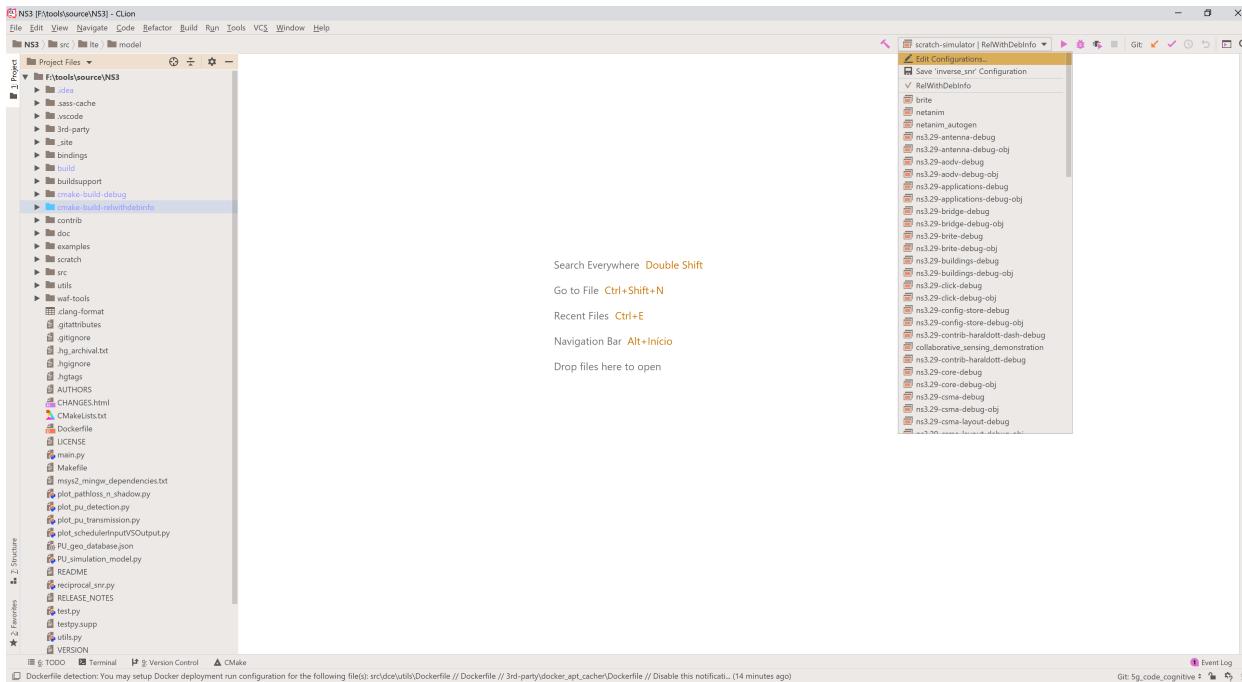
Code::Blocks

Start by installing Code::Blocks.

Code::Blocks does not support CMake project natively, but we can use the corresponding CMake generator to generate a project in order to use it. The generator name depends on the operating system and underlying build system. https://cmake.org/cmake/help/latest/generator/CodeBlocks.html

```
$ ./ns3 configure -G"CodeBlocks - Ninja" --enable-examples

...

$ -- Build files have been written to: /ns-3-dev/cmake-cache
```

There will be a NS3.cbp file inside the cache folder used during configuration (in this case cmake-cache). This is a Code::Blocks project file that can be opened by the IDE.

When you first open the IDE, you will be greeted by a window asking you to select the compiler you want.

Compilers auto-detection   ⊗

Note: After auto-detection, at least one compiler's master path is still empty and therefore invalid.
Inspect the list below and change the compiler's master path later in the compiler options.
Select you favourite default compiler here:

Compiler	Status		Set as default
GNU GCC Compiler	Detected		
Intel C/C++ Compiler	Not found		
Small Device C Compiler	Not found		
Tiny C Compiler	Not found		
LLVM Clang Compiler	Detected		
GNU GCC Compiler for ARM	Not found		
GNU GCC Compiler for AVR	Not found		
GNU GCC Compiler for LM8	Not found		
GNU GCC Compiler for LM32	Not found		

Current default compiler: GNU GCC Compiler

                                                                                    OK

After that you will get into the landing page where you can open the project.

Loading it will take a while.

After that we can select a target in the top menu (where it says "all") and click to build, run or debug. We can also set breakpoints on the source code.

After clicking to build, the build commands of the underlying build system will be printed in the tab at the bottom. If you clicked to debug, the program will start automatically and stop at the first breakpoint.

You can inspect memory and the current stack enabling those views in Debug->Debugging Windows->Watches and Call Stack. Using the debugging buttons, you can advance line by line, continue until the next breakpoint.

Note: as Code::Blocks doesn't natively support CMake projects, it doesn't refresh the CMake cache, which means you will need to close the project, run the `./ns3` command to refresh the CMake caches after adding/removing source files to/from the CMakeLists.txt files, adding a new module or dependencies between modules.

Apple XCode

Start by installing XCode. Then open it for the first time and accept the license. Then open Xcode->Preferences->Locations and select the command-line tools location.
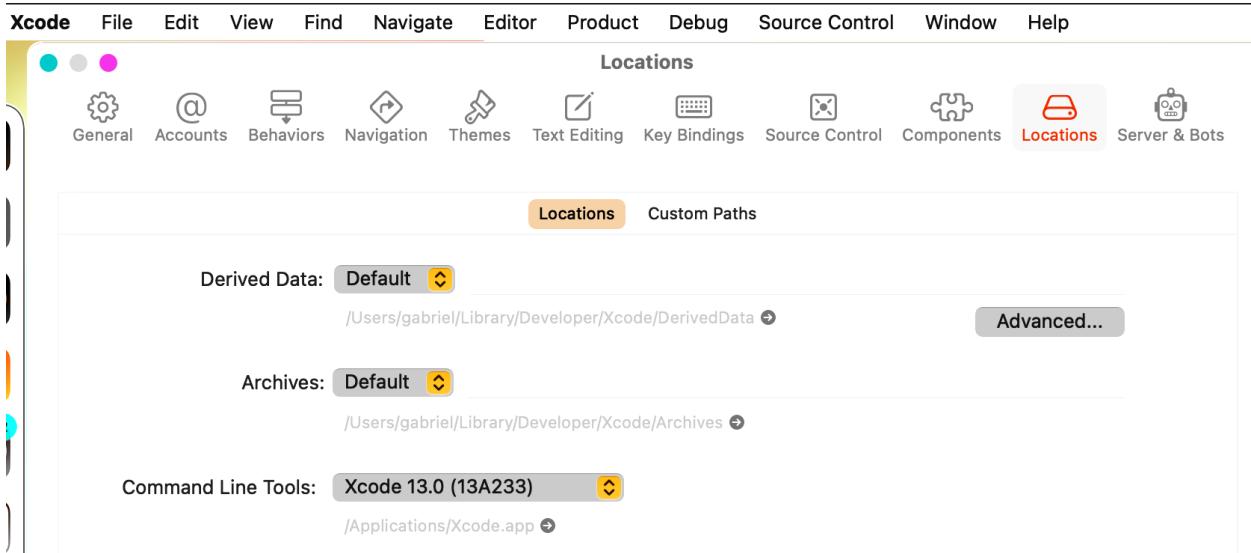
XCode does not support CMake project natively, but we can use the corresponding CMake generator to generate a project in order to use it. The generator name depends on the operating system and underlying build system. https://cmake.org/cmake/help/latest/generator/Xcode.html

```
$ ./ns3 configure -GXcode --enable-examples

...

$ -- Build files have been written to: /ns-3-dev/cmake-cache
```

There will be a NS3.xcodeproj file inside the cache folder used during configuration (in this case cmake-cache). This is a XCode project file that can be opened by the IDE.

Loading the project will take a while, and you will be greeted with the following prompt. Select to automatically create the schemes.
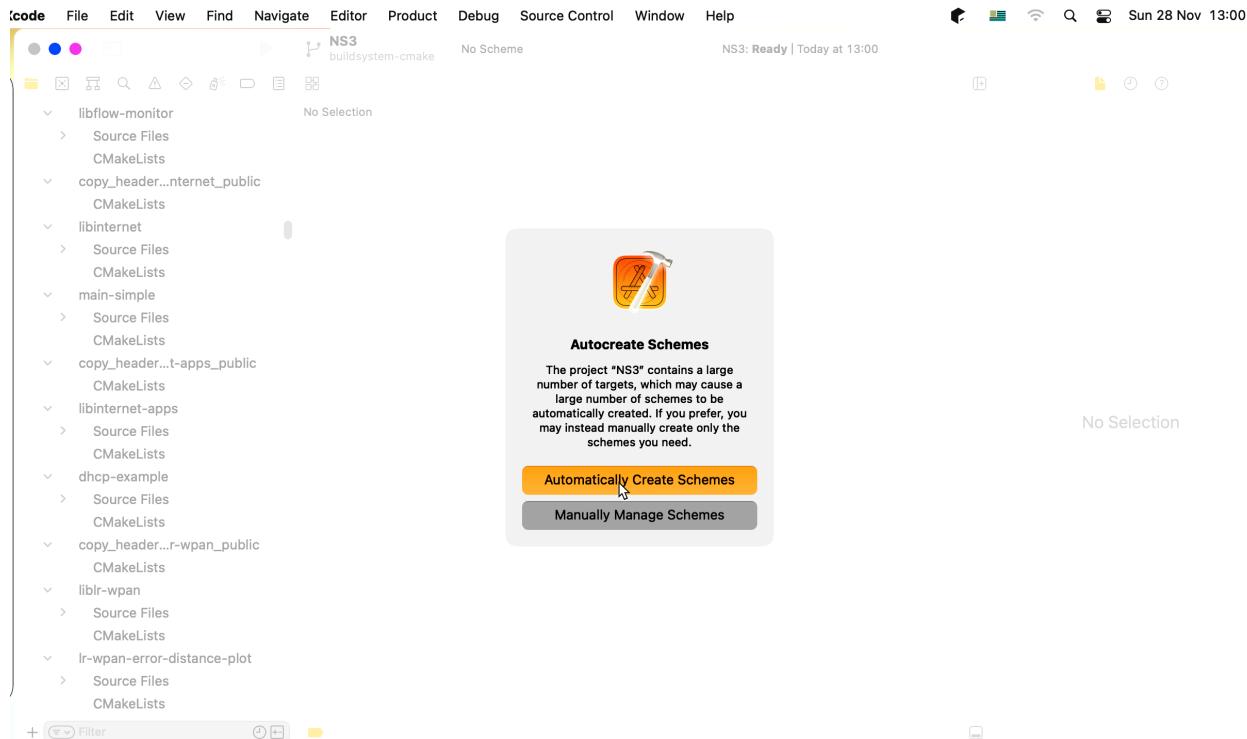
After that we can select a target in the top menu and click to run, which will build and run (if executable, or debug if build with debugging symbols).

After clicking to build, the build will start and progress is shown in the top bar.

Before debugging starts, Xcode will request for permissions to attach to the process (as an attacker could pretend to be a debugging tool and steal data from other processes).

After attaching, we are greeted with profiling information and call stack on the left panel, source code, breakpoint and warnings on the central panel. At the bottom there are the memory watches panel in the left and the output panel on the right, which is also used to read the command line.

Note: as XCode doesn't natively support CMake projects, it doesn't refresh the CMake cache, which means you will need to close the project, run the `./ns3` command to refresh the CMake caches after adding/removing source files to/from the CMakeLists.txt files, adding a new module or dependencies between modules.

4.5 Testing ns-3

You can run the unit tests of the *ns-3* distribution by running the `./test.py` script:

```
$ ./test.py --no-build
```

These tests are run in parallel by ns3. You should eventually see a report saying that

```
92 of 92 tests passed (92 passed, 0 failed, 0 crashed, 0 valgrind errors)
```

This is the important message to check for; failures, crashes, or valgrind errors indicate problems with the code or incompatibilities between the tools and the code.

You will also see the summary output from ns3 and the test runner executing each test, which will actually look something like:

```
-- CCache is enabled
-- The CXX compiler identification is GNU 11.2.0
-- The C compiler identification is GNU 11.2.0

...

-- Configuring done
-- Generating done
-- Build files have been written to: /ns-3-dev/cmake-cache

...
Scanning dependencies of target tap-creator
[  1%] Building CXX object src/fd-net-device/CMakeFiles/tap-device-creator.dir/helper/
→tap-device-creator.cc.o
[  1%] Building CXX object src/tap-bridge/CMakeFiles/tap-creator.dir/model/tap-creator.
→cc.o
[  1%] Building CXX object src/fd-net-device/CMakeFiles/raw-sock-creator.dir/helper/
→creator-utils.cc.o
[  1%] Building CXX object src/tap-bridge/CMakeFiles/tap-creator.dir/model/tap-encode-
→decode.cc.o
[  1%] Linking CXX executable ../../../build/src/fd-net-device/ns3-dev-tap-device-creator

 ...

[100%] Linking CXX executable ../../../build/examples/matrix-topology/ns3-dev-matrix-
→topology
[100%] Built target manet-routing-compare
[100%] Built target matrix-topology
[1/742] PASS: TestSuite aodv-routing-id-cache
[2/742] PASS: TestSuite routing-aodv
[3/742] PASS: TestSuite uniform-planar-array-test
[4/742] PASS: TestSuite angles

...

[740/742] PASS: Example src/wifi/examples/wifi-manager-example --wifiManager=MinstrelHt -
→-standard=802.11ax-6GHz --serverChannelWidth=160 --clientChannelWidth=160 --
→serverShortGuardInterval=3200 --clientShortGuardInterval=3200 --serverNss=4 --
```

<div align="right">(continues on next page)</div>

```
↪clientNss=4 --stepTime=0.1
[741/742] PASS: Example src/lte/examples/lena-radio-link-failure --numberOfEnbs=2 --
↪useIdealRrc=0 --interSiteDistance=700 --simTime=17
[742/742] PASS: Example src/lte/examples/lena-radio-link-failure --numberOfEnbs=2 --
↪interSiteDistance=700 --simTime=17
739 of 742 tests passed (739 passed, 3 skipped, 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. (Note the order of the PASS: ... lines can vary, which is okay. What's important is that the summary line at the end report that all tests passed; none failed or crashed.)

Both ns3 and test.py will split up the job on the available CPU cores of the machine, in parallel.

4.6 Running a Script

We typically run scripts under the control of ns3. 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 ns3. Let's run the *ns-3* equivalent of the ubiquitous hello world program by typing the following:

```
$ ./ns3 run hello-simulator
```

ns3 first checks to make sure that the program is built correctly and executes a build if required. ns3 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 see ns3 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 the ns3 CMake wrapper* 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:

```
$ ./ns3 configure --build-profile=debug --enable-examples --enable-tests
```

to tell ns3 to build the debug versions of the *ns-3* programs that includes the examples and tests. You must still build the actual debug version of the code by typing

```
$ ./ns3
```

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

4.6.1 Program Arguments

To feed command line arguments to an *ns-3* program use this pattern:

```
$ ./ns3 run <ns3-program> --command-template="%s <args>"
```

Substitute your program name for `<ns3-program>`, and the arguments for `<args>`. The `--command-template` argument to ns3 is basically a recipe for constructing the actual command line ns3 should use to execute the program. ns3 checks that the build is complete, sets the shared library paths, then invokes the executable using the provided command line template, inserting the program name for the `%s` placeholder.

If you find the above to be syntactically complicated, a simpler variant exists, which is to include the *ns-3* program and its arguments enclosed by single quotes, such as:

```
$ ./ns3 run '<ns3-program> --arg1=value1 --arg2=value2 ...'
```

Another particularly useful example is to run a test suite by itself. Let's assume that a `mytest` test suite exists (it doesn't). Above, we used the `./test.py` script to run a whole slew of tests in parallel, by repeatedly invoking the real testing program, `test-runner`. To invoke `test-runner` directly for a single test:

```
$ ./ns3 run test-runner --command-template="%s --suite=mytest --verbose"
```

This passes the arguments to the `test-runner` program. Since `mytest` does not exist, an error message will be generated. To print the available `test-runner` options:

```
$ ./ns3 run test-runner --command-template="%s --help"
```

4.6.2 Debugging

To run *ns-3* programs under the control of another utility, such as a debugger (*e.g.* gdb) or memory checker (*e.g.* valgrind), you use a similar `--command-template="..."` form.

For example, to run your *ns-3* program `hello-simulator` with the arguments `<args>` under the gdb debugger:

```
$ ./ns3 run hello-simulator --command-template="gdb %s --args <args>"
```

Notice that the *ns-3* program name goes with the `--run` argument, and the control utility (here gdb) is the first token in the `--command-template` argument. The `--args` tells gdb that the remainder of the command line belongs to the "inferior" program. (Some gdb's don't understand the `--args` feature. In this case, omit the program arguments from the `--command-template`, and use the gdb command `set args`.)

We can combine this recipe and the previous one to run a test under the debugger:

```
$ ./ns3 run test-runner --command-template="gdb %s --args --suite=mytest --verbose"
```

4.6.3 Working Directory

ns3 needs to run from its location at the top of the *ns-3* tree. This becomes the working directory where output files will be written. But what if you want to keep those files out of the *ns-3* source tree? Use the `--cwd` argument:

```
$ ./ns3 run program-name --cwd=...
```

We mention this `--cwd` command for completeness; most users will simply run ns3 from the top-level directory and generate the output data files there.

4.6.4 Running without Building

As of the ns-3.30 release, a new ns3 option was introduced to allow the running of programs while skipping the build step. This can reduce the time to run programs when, for example, running the same program repeatedly through a shell script, or when demonstrating program execution. The option `--no-build` modifies the `run` option, skipping the build steps of the program and required ns-3 libraries.

```
$ ./ns3 run '<ns3-program> --arg1=value1 --arg2=value2 ...' --no-build
```

4.6.5 Build version

As of the ns-3.32 release, a new ns3 configure option `--enable-build-version` was introduced which inspects the local ns3 git repository during builds and adds version metadata to the core module.

This configuration option has the following prerequisites:

- The ns-3 directory must be part of a local git repository
- The local git repository must have at least one ns-3 release tag

or

- A file named version.cache, containing version information, is located in the src/core directory

If these prerequisites are not met, the configuration will fail.

When these prerequisites are met and ns-3 is configured with the `--enable-build-version` option, the ns3 command `show version` can be used to query the local git repository and display the current version metadata.

```
$ ./ns3 show version
```

ns3 will collect information about the build and print out something similar to the output below.

```
ns-3.33+249@g80e0dd0-dirty-debug
```

If `show version` is run when `--enable-build-version` was not configured, an error message indicating that the option is disabled will be displayed instead.

```
Build version support is not enabled, reconfigure with --enable-build-version flag
```

The build information is generated by examining the current state of the git repository. The output of `show version` will change whenever the state of the active branch changes.

The output of `show version` has the following format:

```
<version_tag>[+closest_tag][+distance_from_tag]@<commit_hash>[-tree_state]-<profile>
```

version_tag

      version_tag contains the version of the ns-3 code. The version tag is defined as a git tag with the format ns-3*. If multiple git tags match the format, the tag on the active branch which is closest to the current commit is chosen.

closest_tag

      closest_tag is similar to version_tag except it is the first tag found, regardless of format. The closest tag is not included in the output when closest_tag and version_tag have the same value.

distance_from_tag

      distance_from_tag contains the number of commits between the current commit and closest_tag. distance_from_tag is not included in the output when the value is 0 (i.e. when closest_tag points to the current commit)

commit_hash

      commit_hash is the hash of the commit at the tip of the active branch. The value is 'g' followed by the first 7 characters of the commit hash. The 'g' prefix is used to indicate that this is a git hash.

tree_state

      tree_state indicates the state of the working tree. When the working tree has uncommitted changes this field has the value 'dirty'. The tree state is not included in the version output when the working tree is clean (e.g. when there are no uncommitted changes).

profile

      The build profile specified in the `--build-profile` option passed to `ns3 configure`

A new class, named Version, has been added to the core module. The Version class contains functions to retrieve individual fields of the build version as well as functions to print the full build version like `show version`. The `build-version-example` application provides an example of how to use the Version class to retrieve the various build version fields. See the documentation for the Version class for specifics on the output of the Version class functions.

The version information stored in the Version class is updated every time the git repository changes. This may lead to frequent recompilations/linking of the core module when the `--enable-build-version` option is configured.

```
build-version-example:
Program Version (according to CommandLine): ns-3.33+249@g80e0dd0-dirty-debug

Version fields:
LongVersion:        ns-3.33+249@g80e0dd0-dirty-debug
ShortVersion:       ns-3.33+*
BuildSummary:       ns-3.33+*
VersionTag:         ns-3.33
Major:              3
Minor:              33
Patch:              0
ReleaseCandidate:
ClosestAncestorTag: ns-3.33
TagDistance:        249
CommitHash:         g80e0dd0
BuildProfile:       debug
WorkingTree:        dirty
```

The CommandLine class has also been updated to support the `--version` option which will print the full build version and exit.

```
./ns3 run "command-line-example --version" --no-build
ns-3.33+249@g80e0dd0-dirty-debug
```

If the `--enable-build-version` option was not configured, `--version` will print out a message similar to `show version` indicating that the build version option is not enabled.

4.6.6 Source version

An alternative to storing build version information in the *ns-3* libraries is to track the source code version used to build the code. When using Git, the following recipe can be added to Bash shell scripts to create a `version.txt` file with Git revision information, appended with a patch of any changes to that revision if the repository is dirty. The resulting text file can then be saved with any corresponding *ns-3* simulation results.

```
echo `git describe` > version.txt
gitDiff=`git diff`
if [[ $gitDiff ]]
then
    echo "$gitDiff" >> version.txt
fi
```

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.

5.1 Key Abstractions

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

5.1.1 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*.

5.1.2 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

5.1.3 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.

5.1.4 Net Device

It used to be the case that if you wanted to connect a computer 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 `WifiNetDevice` is designed to work with a `WifiChannel`.

5.1.5 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.

5.2 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 `workspace` under your home directory. Change into that release directory, and you should find a directory structure something like the following:

```
AUTHORS        CMakeLists.txt    examples    RELEASE_NOTES.md  testpy.supp
bindings       contrib           LICENSE     scratch           utils
build-support  CONTRIBUTING.md   ns3         src               utils.py
CHANGES.md     doc               README.md   test.py           VERSION
```

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.

5.2.1 Copyright

The *ns-3* simulator is licensed using the GNU General Public License version 2. 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.

```
/*
 * SPDX-License-Identifier: GPL-2.0-only
 */
```

5.2.2 Module Includes

The code proper starts with a number of include statements.

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-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, ns3 will place public header files in an `ns3` directory under the appropriate `build/debug` or `build/optimized` directory depending on your configuration. CMake 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 run the following command from the top-level directory:

```
$ ./ns3 configure -d debug --enable-examples --enable-tests
```

in order to configure the project to perform debug builds that include examples and tests. You will also have called

```
$ ./ns3 build
```

to build the project. So now if you look in the directory `../../build/include/ns3` you will find the four module include files shown above (among many other header files). 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.

5.2.3 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.

5.2.4 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 "Documentation" in the navigation bar. If you select this link, you will be taken to our documentation page. There is a link to "Latest Release" that will take you to the documentation for the latest stable release of *ns-3*. If you select the "API Documentation" link, you will be taken to the *ns-3* API documentation page.

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. The *ns-3* logging subsystem is discussed in the *Using the Logging Module* section, so we'll get to it later in this tutorial, but you can find out about the above statement by looking at the `Core` module, then expanding the `Debugging tools` book, and then selecting the `Logging` page. Click on `Logging`.

You should now be looking at the Doxygen documentation for the Logging module. In the list of `Macros`'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.

5.2.5 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 line sets the time resolution to one nanosecond, which happens to be the default value:

```
Time::SetResolution(Time::NS);
```

The resolution is the smallest time value that can be represented (as well as the smallest representable difference between two time values). You can change the resolution exactly once. The mechanism enabling this flexibility is somewhat memory hungry, so once the resolution has been set explicitly we release the memory, preventing further updates. (If you don't set the resolution explicitly, it will default to one nanosecond, and the memory will be released when the simulation starts.)

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.

5.2.6 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 propagation 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.

5.2.7 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));
serverApps.Stop(Seconds(10));
```

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 `NodeContainer` 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));
serverApps.Stop(Seconds(10));
```

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));
clientApps.Stop(Seconds(10));
```

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).

5.2.8 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));
serverApps.Stop(Seconds(10));
...
clientApps.Start(Seconds(2));
clientApps.Stop(Seconds(10));
```

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;
}
```

When the simulator will stop?

ns-3 is a Discrete Event (DE) simulator. In such a simulator, each event is associated with its execution time, and the simulation proceeds by executing events in the temporal order of simulation time. Events may cause future events to be scheduled (for example, a timer may reschedule itself to expire at the next interval).

The initial events are usually triggered by each object, e.g., IPv6 will schedule Router Advertisements, Neighbor Solicitations, etc., an Application schedule the first packet sending event, etc.

When an event is processed, it may generate zero, one or more events. As a simulation executes, events are consumed, but more events may (or may not) be generated. The simulation will stop automatically when no further events are in the event queue, or when a special Stop event is found. The Stop event is created through the `Simulator::Stop(stopTime);` function.

There is a typical case where `Simulator::Stop` is absolutely necessary to stop the simulation: when there is a self-sustaining event. Self-sustaining (or recurring) events are events that always reschedule themselves. As a consequence, they always keep the event queue non-empty.

There are many protocols and modules containing recurring events, e.g.:

- FlowMonitor - periodic check for lost packets

- RIPng - periodic broadcast of routing tables update

- etc.

In these cases, `Simulator::Stop` is necessary to gracefully stop the simulation. In addition, when *ns-3* is in emulation mode, the `RealtimeSimulator` is used to keep the simulation clock aligned with the machine clock, and `Simulator::Stop` is necessary to stop the process.

Many of the simulation programs in the tutorial do not explicitly call `Simulator::Stop`, since the event queue will automatically run out of events. However, these programs will also accept a call to `Simulator::Stop`. For example, the following additional statement in the first example program will schedule an explicit stop at 11 seconds:

```
+  Simulator::Stop(Seconds(11));
   Simulator::Run();
   Simulator::Destroy();
   return 0;
}
```

The above will not actually change the behavior of this program, since this particular simulation naturally ends after 10 seconds. But if you were to change the stop time in the above statement from 11 seconds to 1 second, you would notice that the simulation stops before any output is printed to the screen (since the output occurs around time 2 seconds of simulation time).

It is important to call `Simulator::Stop` *before* calling `Simulator::Run`; otherwise, `Simulator::Run` may never return control to the main program to execute the stop!

5.2.9 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 ns3. 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 ns3:

```
$ ./ns3 build
```

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

```
Scanning dependencies of target scratch_myfirst
[  0%] Building CXX object scratch/CMakeFiles/scratch_myfirst.dir/myfirst.cc.o
[  0%] Linking CXX executable ../../build/scratch/ns3.36.1-myfirst-debug
Finished executing the following commands:
cd cmake-cache; cmake --build . -j 7 ; cd ..
```

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):

```
$ ./ns3 run scratch/myfirst
```

You should see some output:

```
At time +2s client sent 1024 bytes to 10.1.1.2 port 9
At time +2.00369s server received 1024 bytes from 10.1.1.1 port 49153
At time +2.00369s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.00737s client received 1024 bytes from 10.1.1.2 port 9
```

Here 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.

5.3 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.

Our example scripts are in the `examples` directory. If you change to `examples` directory, you will see a list of subdirectories. One of the files in `tutorial` subdirectory 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. The core of the simulator is in the `src/core/model` subdirectory. The first file you will find there (as of this writing) is `abort.h`. If you open that file, you can view 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/applications/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.

TWEAKING

6.1 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.

6.1.1 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.

- LOG_ERROR — Log error messages (associated macro: NS_LOG_ERROR);

- LOG_WARN — Log warning messages (associated macro: NS_LOG_WARN);

- LOG_INFO — Log informational messages about program progress (associated macro: NS_LOG_INFO);

- LOG_FUNCTION — Log a message describing each function called (two associated macros: NS_LOG_FUNCTION, used for member functions, and NS_LOG_FUNCTION_NOARGS, used for static functions);

- LOG_LOGIC – Log messages describing logical flow within a function (associated macro: NS_LOG_LOGIC);

- LOG_DEBUG — Log relatively rare, ad-hoc debugging messages (associated macro: NS_LOG_DEBUG);

- LOG_ALL — Log everything mentioned above (no associated macro).

For each LOG_TYPE there is also LOG_LEVEL_TYPE that, if used, enables logging of all the levels above it in addition to its level (see list above). (As a consequence of this, LOG_ERROR and LOG_LEVEL_ERROR and also LOG_ALL and LOG_LEVEL_ALL are functionally equivalent.) For example, enabling LOG_INFO will only enable messages provided by NS_LOG_INFO macro, while enabling LOG_LEVEL_INFO will also enable messages provided by NS_LOG_DEBUG, NS_LOG_WARN and NS_LOG_ERROR macros.

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

- NS_LOG_UNCOND – Log the associated message unconditionally (no associated log level).

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.

6.1.2 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,

```
$ ./ns3 run scratch/myfirst
```

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

```
At time +2s client sent 1024 bytes to 10.1.1.2 port 9
At time +2.00369s server received 1024 bytes from 10.1.1.1 port 49153
At time +2.00369s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.00737s client received 1024 bytes from 10.1.1.2 port 9
```

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 something similar to the following output:

```
UdpEchoClientApplication:UdpEchoClient(0xef90d0)
UdpEchoClientApplication:SetDataSize(0xef90d0, 1024)
```

(continues on next page)

```
UdpEchoClientApplication:StartApplication(0xef90d0)
UdpEchoClientApplication:ScheduleTransmit(0xef90d0, +0ns)
UdpEchoClientApplication:Send(0xef90d0)
At time +2s client sent 1024 bytes to 10.1.1.2 port 9
At time +2.00369s server received 1024 bytes from 10.1.1.1 port 49153
At time +2.00369s server sent 1024 bytes to 10.1.1.1 port 49153
UdpEchoClientApplication:HandleRead(0xef90d0, 0xee7b20)
At time +2.00737s client received 1024 bytes from 10.1.1.2 port 9
UdpEchoClientApplication:StopApplication(0xef90d0)
UdpEchoClientApplication:DoDispose(0xef90d0)
UdpEchoClientApplication:~UdpEchoClient(0xef90d0)
```

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. Generally, use of (at least) NS_LOG_FUNCTION(this) in member functions is preferred. Use NS_LOG_FUNCTION_NOARGS() only in static functions. Note, however, 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.

```
UdpEchoClientApplication:UdpEchoClient(0xea8e50)
UdpEchoClientApplication:SetDataSize(0xea8e50, 1024)
UdpEchoClientApplication:StartApplication(0xea8e50)
UdpEchoClientApplication:ScheduleTransmit(0xea8e50, +0ns)
UdpEchoClientApplication:Send(0xea8e50)
UdpEchoClientApplication:Send(): At time +2s client sent 1024 bytes to 10.1.1.2 port 9
At time +2.00369s server received 1024 bytes from 10.1.1.1 port 49153
At time +2.00369s server sent 1024 bytes to 10.1.1.1 port 49153
UdpEchoClientApplication:HandleRead(0xea8e50, 0xea5b20)
UdpEchoClientApplication:HandleRead(): At time +2.00737s client received 1024 bytes from␣
→10.1.1.2 port 9
UdpEchoClientApplication:StopApplication(0xea8e50)
UdpEchoClientApplication:DoDispose(0xea8e50)
UdpEchoClientApplication:~UdpEchoClient(0xea8e50)
```

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. Also, in most log statements, you will see a hexadecimal value printed such as `0xea8e50`; this is because most statements print out the value of the C++ `this` pointer, so that objects can be distinguished from one another.

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.

```
UdpEchoServerApplication:UdpEchoServer(0x2101590)
UdpEchoClientApplication:UdpEchoClient(0x2101820)
UdpEchoClientApplication:SetDataSize(0x2101820, 1024)
UdpEchoServerApplication:StartApplication(0x2101590)
UdpEchoClientApplication:StartApplication(0x2101820)
UdpEchoClientApplication:ScheduleTransmit(0x2101820, +0ns)
UdpEchoClientApplication:Send(0x2101820)
UdpEchoClientApplication:Send(): At time +2s client sent 1024 bytes to 10.1.1.2 port 9
UdpEchoServerApplication:HandleRead(0x2101590, 0x2106240)
UdpEchoServerApplication:HandleRead(): At time +2.00369s server received 1024 bytes from␣
↪10.1.1.1 port 49153
UdpEchoServerApplication:HandleRead(): Echoing packet
UdpEchoServerApplication:HandleRead(): At time +2.00369s server sent 1024 bytes to 10.1.
↪1.1 port 49153
UdpEchoClientApplication:HandleRead(0x2101820, 0x21134b0)
UdpEchoClientApplication:HandleRead(): At time +2.00737s client received 1024 bytes from␣
↪10.1.1.2 port 9
UdpEchoClientApplication:StopApplication(0x2101820)
UdpEchoServerApplication:StopApplication(0x2101590)
UdpEchoClientApplication:DoDispose(0x2101820)
UdpEchoServerApplication:DoDispose(0x2101590)
UdpEchoClientApplication:~UdpEchoClient(0x2101820)
UdpEchoServerApplication:~UdpEchoServer(0x2101590)
```

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:

```
+0.000000000s UdpEchoServerApplication:UdpEchoServer(0x8edfc0)
+0.000000000s UdpEchoClientApplication:UdpEchoClient(0x8ee210)
+0.000000000s UdpEchoClientApplication:SetDataSize(0x8ee210, 1024)
+1.000000000s UdpEchoServerApplication:StartApplication(0x8edfc0)
+2.000000000s UdpEchoClientApplication:StartApplication(0x8ee210)
+2.000000000s UdpEchoClientApplication:ScheduleTransmit(0x8ee210, +0ns)
+2.000000000s UdpEchoClientApplication:Send(0x8ee210)
```

(continues on next page)

```
+2.000000000s UdpEchoClientApplication:Send(): At time +2s client sent 1024 bytes to 10.
→1.1.2 port 9
+2.003686400s UdpEchoServerApplication:HandleRead(0x8edfc0, 0x936770)
+2.003686400s UdpEchoServerApplication:HandleRead(): At time +2.00369s server received
→1024 bytes from 10.1.1.1 port 49153
+2.003686400s UdpEchoServerApplication:HandleRead(): Echoing packet
+2.003686400s UdpEchoServerApplication:HandleRead(): At time +2.00369s server sent 1024
→bytes to 10.1.1.1 port 49153
+2.007372800s UdpEchoClientApplication:HandleRead(0x8ee210, 0x8f3140)
+2.007372800s UdpEchoClientApplication:HandleRead(): At time +2.00737s client received
→1024 bytes from 10.1.1.2 port 9
+10.000000000s UdpEchoClientApplication:StopApplication(0x8ee210)
+10.000000000s UdpEchoServerApplication:StopApplication(0x8edfc0)
UdpEchoClientApplication:DoDispose(0x8ee210)
UdpEchoServerApplication:DoDispose(0x8edfc0)
UdpEchoClientApplication:~UdpEchoClient(0x8ee210)
UdpEchoServerApplication:~UdpEchoServer(0x8edfc0)
```

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/myfirst.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 thousands of 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,

```
$ ./ns3 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.

6.1.3 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 ns3 and clear the `NS_LOG` variable to turn off the torrent of logging we previously enabled:

```
$ ./ns3
$ export NS_LOG=""
```

Now, if you run the script,

```
$ ./ns3 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,

```
Creating Topology
At time +2s client sent 1024 bytes to 10.1.1.2 port 9
At time +2.00369s server received 1024 bytes from 10.1.1.1 port 49153
At time +2.00369s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.00737s client received 1024 bytes from 10.1.1.2 port 9
```

6.2 Using Command Line Arguments

6.2.1 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,

```
$ ./ns3 run "scratch/myfirst --PrintHelp"
```

This will ask ns3 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,

```
myfirst [General Arguments]
General Arguments:
  --PrintGlobals:              Print the list of globals.
  --PrintGroups:               Print the list of groups.
  --PrintGroup=[group]:        Print all TypeIds of group.
  --PrintTypeIds:              Print all TypeIds.
  --PrintAttributes=[typeid]:  Print all attributes of typeid.
  --PrintVersion:              Print the ns-3 version.
  --PrintHelp:                 Print this help message.
```

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,

```
$ ./ns3 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 ns3 (`./ns3`) 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,

```
+0.000000000s UdpEchoServerApplication:UdpEchoServer(0x20d0d10)
+1.000000000s UdpEchoServerApplication:StartApplication(0x20d0d10)
At time +2s client sent 1024 bytes to 10.1.1.2 port 9
+2.257324218s UdpEchoServerApplication:HandleRead(0x20d0d10, 0x20900b0)
+2.257324218s At time +2.25732s server received 1024 bytes from 10.1.1.1 port 49153
+2.257324218s Echoing packet
+2.257324218s At time +2.25732s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.51465s client received 1024 bytes from 10.1.1.2 port 9
+10.000000000s UdpEchoServerApplication:StopApplication(0x20d0d10)
UdpEchoServerApplication:DoDispose(0x20d0d10)
UdpEchoServerApplication:~UdpEchoServer(0x20d0d10)
```

Recall that the last time we looked at the simulation time at which the packet was received by the echo server, it was at 2.0073728 seconds.

```
+2.007372800s 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:

```
$ ./ns3 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:

```
$ ./ns3 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,

```
$ ./ns3 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:

```
+0.000000000s UdpEchoServerApplication:UdpEchoServer(0x1df20f0)
+1.000000000s UdpEchoServerApplication:StartApplication(0x1df20f0)
At time +2s client sent 1024 bytes to 10.1.1.2 port 9
+2.003686400s UdpEchoServerApplication:HandleRead(0x1df20f0, 0x1de0250)
+2.003686400s At time +2.00369s server received 1024 bytes from 10.1.1.1 port 49153
+2.003686400s Echoing packet
+2.003686400s At time +2.00369s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.00737s client received 1024 bytes from 10.1.1.2 port 9
+10.000000000s UdpEchoServerApplication:StopApplication(0x1df20f0)
UdpEchoServerApplication:DoDispose(0x1df20f0)
UdpEchoServerApplication:~UdpEchoServer(0x1df20f0)
```

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,

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

A natural question to arise at this point is how to learn about the existence of all of these attributes. Again, the command line help facility has a feature for this. If we ask for command line help we should see:

```
$ ./ns3 run "scratch/myfirst --PrintHelp"
myfirst [General Arguments]
```

(continues on next page)

```
General Arguments:
  --PrintGlobals:              Print the list of globals.
  --PrintGroups:               Print the list of groups.
  --PrintGroup=[group]:        Print all TypeIds of group.
  --PrintTypeIds:              Print all TypeIds.
  --PrintAttributes=[typeid]:  Print all attributes of typeid.
  --PrintVersion:              Print the ns-3 version.
  --PrintHelp:                 Print this help message.
```

If you select the "PrintGroups" argument, you should see a list of all registered `TypeId` groups. The group names are aligned with the module names in the source directory (although with a leading capital letter). Printing out all of the information at once would be too much, so a further filter is available to print information on a per-group basis. So, focusing again on the point-to-point module:

```
./ns3 run "scratch/myfirst --PrintGroup=PointToPoint"
TypeIds in group PointToPoint:
  ns3::PointToPointChannel
  ns3::PointToPointNetDevice
  ns3::PppHeader
```

and from here, one can find the possible `TypeId` names to search for attributes, such as in the `--PrintAttributes=ns3::PointToPointChannel` example shown above.

Another way to find out about attributes is through the ns-3 Doxygen; there is a page that lists out all of the registered attributes in the simulator.

6.2.2 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,

```
$ ./ns3 build
$ ./ns3 run "scratch/myfirst --PrintHelp"
```

```
[Program Options] [General Arguments]

Program Options:
  --nPackets:  Number of packets to echo [1]

General Arguments:
  --PrintGlobals:              Print the list of globals.
  --PrintGroups:               Print the list of groups.
  --PrintGroup=[group]:        Print all TypeIds of group.
  --PrintTypeIds:              Print all TypeIds.
  --PrintAttributes=[typeid]:  Print all attributes of typeid.
  --PrintVersion:              Print the ns-3 version.
  --PrintHelp:                 Print this help message.
```

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,

```
$ ./ns3 run "scratch/myfirst --nPackets=2"
```

You should now see

```
+0.000000000s UdpEchoServerApplication:UdpEchoServer(0x836e50)
+1.000000000s UdpEchoServerApplication:StartApplication(0x836e50)
At time +2s client sent 1024 bytes to 10.1.1.2 port 9
+2.003686400s UdpEchoServerApplication:HandleRead(0x836e50, 0x8450c0)
+2.003686400s At time +2.00369s server received 1024 bytes from 10.1.1.1 port 49153
+2.003686400s Echoing packet
+2.003686400s At time +2.00369s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.00737s client received 1024 bytes from 10.1.1.2 port 9
At time +3s client sent 1024 bytes to 10.1.1.2 port 9
+3.003686400s UdpEchoServerApplication:HandleRead(0x836e50, 0x8450c0)
+3.003686400s At time +3.00369s server received 1024 bytes from 10.1.1.1 port 49153
+3.003686400s Echoing packet
+3.003686400s At time +3.00369s server sent 1024 bytes to 10.1.1.1 port 49153
At time +3.00737s client received 1024 bytes from 10.1.1.2 port 9
+10.000000000s UdpEchoServerApplication:StopApplication(0x836e50)
UdpEchoServerApplication:DoDispose(0x836e50)
UdpEchoServerApplication:~UdpEchoServer(0x836e50)
```

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.

6.3 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:

```cpp
#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.

6.3.1 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:

```
$ ./ns3 run scratch/myfirst
```

Just as you have seen many times before, you will see some messages from ns3 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 ns3 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 ns3 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 reference number on the left side:

```
1  +
2  2
3  /NodeList/0/DeviceList/0/$ns3::PointToPointNetDevice/TxQueue/Enqueue
4  ns3::PppHeader (
5    Point-to-Point Protocol: IP (0x0021))
6    ns3::Ipv4Header (
7      tos 0x0 ttl 64 id 0 protocol 17 offset 0 flags [none]
8      length: 1052 10.1.1.1 > 10.1.1.2)
9      ns3::UdpHeader (
10       length: 1032 49153 > 9)
11       Payload (size=1024)
```

The first section of this expanded trace event (reference number 0) is the operation. We have a + character, so this corresponds to an *enqueue* operation on the transmit queue. The second section (reference 1) 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 section of the example trace (reference 2) 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 sections in the trace should be fairly intuitive. References 3-4 indicate that the packet is encapsulated in the point-to-point protocol. References 5-7 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 8-9 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.

```
1  r
2  2.25732
3  /NodeList/1/DeviceList/0/$ns3::PointToPointNetDevice/MacRx
4    ns3::Ipv4Header (
5      tos 0x0 ttl 64 id 0 protocol 17 offset 0 flags [none]
6      length: 1052 10.1.1.1 > 10.1.1.2)
7      ns3::UdpHeader (
8        length: 1032 49153 > 9)
9        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.

6.3.2 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:

```
$ ./ns3 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*.

BUILDING TOPOLOGIES

7.1 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/network-module.h"
#include "ns3/csma-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/ipv4-global-routing-helper.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 2000 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));
serverApps.Stop(Seconds(10));
```

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));
clientApps.Stop(Seconds(10));
```

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 the ns3 build script 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
$ ./ns3 build
```

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 closely, you will still have the NS_LOG variable set, so go ahead and clear that variable and run the program.

```
$ export NS_LOG=""
$ ./ns3 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.

```
At time +2s client sent 1024 bytes to 10.1.2.4 port 9
At time +2.0078s server received 1024 bytes from 10.1.1.1 port 49153
At time +2.0078s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.01761s client received 1024 bytes from 10.1.2.4 port 9
```

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 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
2.017607 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
2.013921 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)
2.007698 ARP, Request who-has 10.1.2.4 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1, length 50
2.007710 ARP, Reply 10.1.2.4 is-at 00:00:00:00:00:06, length 50
2.007803 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
```

```
2.013815 ARP, Request who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.4, length 50
2.013828 ARP, Reply 10.1.2.1 is-at 00:00:00:00:00:03, length 50
2.013921 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,

```
2.007698 ARP, Request who-has 10.1.2.4 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1, length 50
2.007710 ARP, Reply 10.1.2.4 is-at 00:00:00:00:00:06, length 50
```

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

```
2.007803 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.

```
2.013815 ARP, Request who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.4, length 50
2.013828 ARP, Reply 10.1.2.1 is-at 00:00:00:00:00:03, length 50
```

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

```
2.013921 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
2.013921 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 2.017607 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
2.017607 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, instead of the default value of three (extra nodes):

```
$ ./ns3 run "scratch/mysecond --nCsma=4"
```

You should now see,

```
At time +2s client sent 1024 bytes to 10.1.2.5 port 9
At time +2.0118s server received 1024 bytes from 10.1.1.1 port 49153
At time +2.0118s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.02461s client received 1024 bytes from 10.1.2.5 port 9
```

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
```

On line 110, notice the following command to enable tracing on one node (the index 1 corresponds to the second CSMA node in the container):

```
csma.EnablePcap("second", csmaDevices.Get(1), true);
```

Change the index to the quantity `nCsma`, corresponding to the last node in the topology– the node that contains the echo server:

```
csma.EnablePcap("second", csmaDevices.Get(nCsma), true);
```

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

```
$ ./ns3 build
$ ./ns3 run "scratch/mysecond --nCsma=100"
```

you will see the following output:

```
At time +2s client sent 1024 bytes to 10.1.2.101 port 9
At time +2.0068s server received 1024 bytes from 10.1.1.1 port 49153
At time +2.0068s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.01761s client received 1024 bytes from 10.1.2.101 port 9
```

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-1-0.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 true. This means that the trace gathered on that node was in promiscuous mode.

To illustrate the difference between promiscuous and non-promiscuous traces, let's add a non-promiscuous trace for the next-to-last node. Add the following line before or after the existing PCAP trace line; the last argument of `false` indicates that you would like a non-promiscuous trace:

```
csma.EnablePcap("second", csmaDevices.Get(nCsma - 1), false);
```

Now build and run as before:

```
$ rm *.pcap
$ ./ns3 build
$ ./ns3 run "scratch/mysecond --nCsma=100"
```

This will produce a new PCAP file, `second-100-0.pcap`. 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)
2.006698 ARP, Request who-has 10.1.2.101 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1, length 50
2.013815 ARP, Request who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.101, length 50
```

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

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

Node 101 is really the participant in the echo exchange; the following trace will exist regardless of whether promiscuous mode is set on that PCAP statement.

```
reading from file second-101-0.pcap, link-type EN10MB (Ethernet)
2.006698 ARP, Request who-has 10.1.2.101 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1, length 50
2.006698 ARP, Reply 10.1.2.101 is-at 00:00:00:00:00:67, length 50
2.006803 IP 10.1.1.1.49153 > 10.1.2.101.9: UDP, length 1024
2.013803 ARP, Request who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.101, length 50
2.013828 ARP, Reply 10.1.2.1 is-at 00:00:00:00:00:03, length 50
2.013828 IP 10.1.2.101.9 > 10.1.1.1.49153: UDP, length 1024
```

7.2 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.

7.3 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 Wi-Fi 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/point-to-point-module.h"
#include "ns3/network-module.h"
#include "ns3/applications-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/csma-module.h"
#include "ns3/internet-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 `Install` 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 Wi-Fi 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 communicate and interfere:

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

Once the PHY helper is configured, we can focus on the MAC layer. The WifiMacHelper object is used to set MAC parameters. The second statement below 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.

```
WifiMacHelper mac;
Ssid ssid = Ssid("ns-3-ssid");
```

WifiHelper will, by default, configure the standard in use to be 802.11ax (known commercially as Wi-Fi 6) and configure a compatible rate control algorithm (IdealWifiManager).

```
WifiHelper wifi;
```

We are now ready to install Wi-Fi models on the nodes, using these four helper objects (YansWifiChannelHelper, YansWifiPhyHelper, WifiMacHelper, WifiHelper) and the Ssid object created above. These helpers have encapsulated a lot of default configuration, and can be further tailored using additional attribute configuration if desired. We also will create NetDevice containers to store pointers to the WifiNetDevice objects that the helper create.

```
NetDeviceContainer staDevices
mac.SetType("ns3::StaWifiMac",
            "Ssid", SsidValue(ssid),
            "ActiveProbing", BooleanValue(false));
```

In the above code, the specific kind of MAC layer that will be created by the helper is specified by the `TypeId` value of `ns3::StaWifiMac` type. The "QosSupported" attribute is set to true by default for `WifiMacHelper` objects when the standard is at least 802.11n or newer. The combination of these two configurations means that the MAC instance next created will be a QoS-aware, 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, and stations will listen for AP beacons.

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 Wi-Fi devices of these stations:

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

We have configured Wi-Fi 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 `WifiMacHelper` to reflect the requirements of the AP.

```
mac.SetType("ns3::ApWifiMac",
            "Ssid", SsidValue(ssid));
```

In this case, the `WifiMacHelper` is going to create MAC layers of the "ns3::ApWifiMac", the latter specifying that a MAC instance configured as an AP should be created.

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 Wi-Fi 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));
serverApps.Stop(Seconds(10));
```

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));
clientApps.Stop(Seconds(10));
```

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));
```

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 Wi-Fi 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 CMake to build just as you did with the `second.cc` example. If you are in the top-level directory of the repository, type the following:

```
$ cp examples/tutorial/third.cc scratch/mythird.cc
$ ./ns3 run 'scratch/mythird --tracing=1'
```

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

```
At time +2s client sent 1024 bytes to 10.1.2.4 port 9
At time +2.01624s server received 1024 bytes from 10.1.3.3 port 49153
At time +2.01624s server sent 1024 bytes to 10.1.3.3 port 49153
At time +2.02849s client received 1024 bytes from 10.1.2.4 port 9
```

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 look in the top level directory, and you enabled tracing at the command-line as suggested above, 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 Wi-Fi 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 Wi-Fi 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_RADIO (802.11 plus radiotap␣
↪header)

0.033119 33119us tsft 6.0 Mb/s 5210 MHz 11a Beacon (ns-3-ssid) [6.0* 9.0 12.0* 18.0 24.
↪0* 36.0 48.0 54.0 Mbit] ESS
0.120504 120504us tsft 6.0 Mb/s 5210 MHz 11a -62dBm signal -94dBm noise Assoc Request␣
↪(ns-3-ssid) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.120520 120520us tsft 6.0 Mb/s 5210 MHz 11a Acknowledgment RA:00:00:00:00:00:08
0.120632 120632us tsft 6.0 Mb/s 5210 MHz 11a -62dBm signal -94dBm noise CF-End␣
↪RA:ff:ff:ff:ff:ff:ff
0.120666 120666us tsft 6.0 Mb/s 5210 MHz 11a Assoc Response AID(1) :: Successful
...
```

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 left 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)
2.006440 IP 10.1.3.3.49153 > 10.1.2.4.9: UDP, length 1024
2.025048 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 Wi-Fi 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)
2.010126 IP 10.1.3.3.49153 > 10.1.2.4.9: UDP, length 1024
2.021361 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 Wi-Fi 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)
2.016126 ARP, Request who-has 10.1.2.4 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1, length 50
2.016151 ARP, Reply 10.1.2.4 is-at 00:00:00:00:00:06, length 50
2.016151 IP 10.1.3.3.49153 > 10.1.2.4.9: UDP, length 1024
2.021255 ARP, Request who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.4, length 50
2.021255 ARP, Reply 10.1.2.1 is-at 00:00:00:00:00:03, length 50
2.021361 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 (i.e., just after the `NS_LOG_COMPONENT_DEFINE` statement), 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.

```
$ ./ns3 build
$ ./ns3 run scratch/mythird
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10, y = 0
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.36083, y = -0.769065
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.62346, y = 0.195831
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.42533, y = 1.17601
/NodeList/7/$ns3::MobilityModel/CourseChange x = 8.4854, y = 0.834616
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.79244, y = 1.55559
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.85546, y = 2.55361
At time +2s client sent 1024 bytes to 10.1.2.4 port 9
At time +2.01624s server received 1024 bytes from 10.1.3.3 port 49153
At time +2.01624s server sent 1024 bytes to 10.1.3.3 port 49153
At time +2.02849s client received 1024 bytes from 10.1.2.4 port 9
/NodeList/7/$ns3::MobilityModel/CourseChange x = 8.72774, y = 2.06461
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.52954, y = 2.6622
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10.523, y = 2.77665
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10.7054, y = 3.75987
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10.143, y = 2.93301
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10.2355, y = 1.9373
/NodeList/7/$ns3::MobilityModel/CourseChange x = 11.2152, y = 1.73647
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10.2379, y = 1.94864
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10.4491, y = 0.971199
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.56013, y = 1.42913
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.11607, y = 2.32513
/NodeList/7/$ns3::MobilityModel/CourseChange x = 8.22047, y = 1.88027
/NodeList/7/$ns3::MobilityModel/CourseChange x = 8.79149, y = 1.05934
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.41195, y = 0.275103
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.83369, y = -0.631617
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.15219, y = 0.100206
/NodeList/7/$ns3::MobilityModel/CourseChange x = 8.32714, y = 0.665266
```

(continues on next page)

```
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.46368, y = 0.160847
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.40394, y = -0.837367
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.96716, y = -1.73693
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.62062, y = -2.49388
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.99793, y = -1.56779
```

7.4 Queues in ns-3

The selection of queueing disciplines in *ns-3* can have a large impact on performance, and it is important for users to understand what is installed by default and how to change the defaults and observe the performance.

Architecturally, *ns-3* separates the device layer from the IP layers or traffic control layers of an Internet host. Since recent releases of *ns-3*, outgoing packets traverse two queueing layers before reaching the channel object. The first queueing layer encountered is what is called the 'traffic control layer' in *ns-3*; here, active queue management (RFC7567) and prioritization due to quality-of-service (QoS) takes place in a device-independent manner through the use of queueing disciplines. The second queueing layer is typically found in the NetDevice objects. Different devices (e.g. LTE, Wi-Fi) have different implementations of these queues. This two-layer approach mirrors what is found in practice, (software queues providing prioritization, and hardware queues specific to a link type). In practice, it may be even more complex than this. For instance, address resolution protocols have a small queue. Wi-Fi in Linux has four layers of queueing (https://lwn.net/Articles/705884/).

The traffic control layer is effective only if it is notified by the NetDevice when the device queue is full, so that the traffic control layer can stop sending packets to the NetDevice. Otherwise, the backlog of the queueing disciplines is always null and they are ineffective. Currently, flow control, i.e., the ability of notifying the traffic control layer, is supported by the following NetDevices, which use Queue objects (or objects of Queue subclasses) to store their packets:

- Point-To-Point
- Csma
- Wi-Fi
- SimpleNetDevice

The performance of queueing disciplines is highly impacted by the size of the queues used by the NetDevices. Currently, queues by default in *ns-3* are not autotuned for the configured link properties (bandwidth, delay), and are typically the simplest variants (e.g. FIFO scheduling with drop-tail behavior). However, the size of the queues can be dynamically adjusted by enabling BQL (Byte Queue Limits), the algorithm implemented in the Linux kernel to adjust the size of the device queues to fight bufferbloat while avoiding starvation. Currently, BQL is supported by the NetDevices that support flow control. An analysis of the impact of the size of the device queues on the effectiveness of the queueing disciplines conducted by means of *ns-3* simulations and real experiments is reported in:

P. Imputato and S. Avallone. An analysis of the impact of network device buffers on packet schedulers through experiments and simulations. Simulation Modelling Practice and Theory, 80(Supplement C):1–18, January 2018. DOI: 10.1016/j.simpat.2017.09.008

7.4.1 Available queueing models in *ns-3*

At the traffic-control layer, these are the options:

- PFifoFastQueueDisc: The default maximum size is 1000 packets

- FifoQueueDisc: The default maximum size is 1000 packets

- RedQueueDisc: The default maximum size is 25 packets

- CoDelQueueDisc: The default maximum size is 1500 kilobytes

- FqCoDelQueueDisc: The default maximum size is 10240 packets

- PieQueueDisc: The default maximum size is 25 packets

- MqQueueDisc: This queue disc has no limits on its capacity

- TbfQueueDisc: The default maximum size is 1000 packets

By default, a pfifo_fast queueing discipline is installed on a NetDevice when an IPv4 or IPv6 address is assigned to an interface associated with the NetDevice, unless a queueing discipline has been already installed on the NetDevice.

At the device layer, there are device specific queues:

- PointToPointNetDevice: The default configuration (as set by the helper) is to install a DropTail queue of default size (100 packets)

- CsmaNetDevice: The default configuration (as set by the helper) is to install a DropTail queue of default size (100 packets)

- WiFiNetDevice: The default configuration is to install a DropTail queue of default size (100 packets) for non-QoS stations and four DropTail queues of default size (100 packets) for QoS stations

- SimpleNetDevice: The default configuration is to install a DropTail queue of default size (100 packets)

- LteNetDevice: Queueing occurs at the RLC layer (RLC UM default buffer is 10 * 1024 bytes, RLC AM does not have a buffer limit).

- UanNetDevice: There is a default 10 packet queue at the MAC layer

7.4.2 Changing from the defaults

- The type of queue used by a NetDevice can be usually modified through the device helper:

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

PointToPointHelper p2p;
p2p.SetQueue("ns3::DropTailQueue", "MaxSize", StringValue("50p"));

NetDeviceContainer devices = p2p.Install(nodes);
```

- The type of queue disc installed on a NetDevice can be modified through the traffic control helper:

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

TrafficControlHelper tch;
tch.SetRootQueueDisc("ns3::CoDelQueueDisc", "MaxSize", StringValue("1000p"));
tch.Install(devices);
```

- BQL can be enabled on a device that supports it through the traffic control helper:

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

TrafficControlHelper tch;
tch.SetRootQueueDisc("ns3::CoDelQueueDisc", "MaxSize", StringValue("1000p"));
tch.SetQueueLimits("ns3::DynamicQueueLimits", "HoldTime", StringValue("4ms"));
tch.Install(devices);
```

EIGHT

TRACING

8.1 Background

As mentioned in *Using the Tracing System*, the whole point of running an *ns-3* simulation is to generate output for study. You have two basic strategies to obtain output from *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 may require writing scripts to parse and filter for data of interest. 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. `NS_LOG` output is not considered part of the *ns-3* API, and can change without warning between releases. In addition, `NS_LOG` output is only available in debug builds, so relying on it imposes a performance penalty. Of course, if the information of interest 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`, `awk`, `perl` or `python` scripts. If you desire, your output can be formatted directly into a form acceptable by gnuplot, for example (see also *GnuplotHelper*). 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*.

8.1.1 Blunt Instruments

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

```
#include <iostream>
...
void
SomeFunction()
{
  uint32_t x = SOME_INTERESTING_VALUE;
  ...
```

(continues on next page)

```
    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 (see *Using the Logging Module*). 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::ProcessEstablished()` there is no log message for the reception of a SYN+ACK in ESTABLISHED state. You could simply add one, changing the code. Here is the original:

```
/* Received a packet upon ESTABLISHED state. This function is mimicking the
    role of tcp_rcv_established() in tcp_input.c in Linux kernel. */
void
TcpSocketBase::ProcessEstablished(Ptr<Packet> packet, const TcpHeader& tcpHeader)
{
  NS_LOG_FUNCTION(this << tcpHeader);
  ...

  else if (tcpflags == (TcpHeader::SYN | TcpHeader::ACK))
  {
    // No action for received SYN+ACK, it is probably a duplicated packet
  }
  ...
```

To log the SYN+ACK case, you can add a new NS_LOG_LOGIC in the `if` statement body:

```
/* Received a packet upon ESTABLISHED state. This function is mimicking the
   role of tcp_rcv_established() in tcp_input.c in Linux kernel. */
void
TcpSocketBase::ProcessEstablished(Ptr<Packet> packet, const TcpHeader& tcpHeader)
{
  NS_LOG_FUNCTION(this << tcpHeader);
  ...
  else if (tcpflags == (TcpHeader::SYN | TcpHeader::ACK))
  {
    // No action for received SYN+ACK, it is probably a duplicated packet
    NS_LOG_LOGIC("TcpSocketBase " << this << " ignoring SYN+ACK");
  }
```

```
...
```

This may seem fairly simple and satisfying at first glance, but something to consider is that you will be writing code to add `NS_LOG` statements 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, which is typically an entire source code file.

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 which you depend on 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.

Finally, `NS_LOG` output is only available in debug builds, you can't get log output from optimized builds, which run about twice as fast. Relying on `NS_LOG` imposes a performance penalty.

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*, but not suitable for serious work.

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 user code 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.

8.2 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. Every time the congestion window changes connected trace sinks are notified with the old and new value.

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 data 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. Inserting trace sources introduces a very small execution overhead.

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 hooked to 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.

8.2.1 Simple 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 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;
```

(But read the C++-FAQ Section 33 before writing code like this!) 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 need 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 works. Basically, a trace sink *is* a callback. When a trace sink expresses interest in receiving trace events, it adds itself as 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 arguments. The `operator(...)` eventually wanders down into the system and does something remarkably like the indirect call you just saw, providing zero or more parameters, just as 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 multiple 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 *ns-3* Manual.

Walkthrough: `fourth.cc`

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:

```
/*
 * SPDX-License-Identifier: GPL-2.0-only
 */

#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 to get everything at once, but we do the includes explicitly here to illustrate 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 itself around, rather than passing the address of the object. What this all really means is that you will be able to trace all changes made to a TracedValue in a really simple way.

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()
  {
    static TypeId tid = TypeId("MyObject")
      .SetParent(Object::GetTypeId())
      .SetGroupName("MyGroup")
      .AddConstructor<MyObject>()
      .AddTraceSource("MyInteger",
                      "An integer value to trace.",
                      MakeTraceSourceAccessor(&MyObject::m_myInt),
                      "ns3::TracedValueCallback::Int32")
    ;
    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 first argument is a name for this trace source, which makes it visible in the Config system. The

second argument is a help string. Now look at the third argument, in fact focus on the *argument* of the third argument: `&MyObject::m_myInt`. This is the TracedValue which is being added to the class; it is always a class data member. (The final argument is the name of a `typedef` for the TracedValue type, as a string. This is used to generate documentation for the correct Callback function signature, which is useful especially for more general types of Callbacks.)

The `TracedValue<>` declaration provides the infrastructure that drives the callback process. Any time the underlying value is changed the TracedValue mechanism will provide both the old and the new value of that variable, in this case an `int32_t` value. The trace sink function `traceSink` for this TracedValue will need the signature

```
void (*traceSink)(int32_t oldValue, int32_t newValue);
```

All trace sinks hooking this trace source must have this signature. We'll discuss below how you can determine the required callback signature in other cases.

Sure enough, continuing through `fourth.cc` we see:

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

This is the definition of a matching trace sink. It corresponds directly to the callback function signature. Once it is connected, this function will be called whenever the `TracedValue` changes.

We have now seen the trace source and the trace sink. What remains is code to connect the source to the sink, which happens in `main`:

```
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 MyObject instance in which the trace source lives.

The next step, the `TraceConnectWithoutContext`, forms the connection between the trace source and the trace sink. The first argument is just the trace source name "MyInteger" we saw above. 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 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 assignment operators 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 assigning a value to `m_myInt`:

```
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.

Since `m_myInt` is a `TracedValue`, this operator is defined 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 who registering interest in the source being called with the parameters provided by the source.

If you now build and run this example,

```
$ ./ns3 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.

8.2.2 Connect with Config

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 select 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 experimenting 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 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()`. We could have used this `Ptr<Node>` to call a Connect method directly:

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

In the `third.cc` example, we actually wanted 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->GetObject<MobilityModel>()->TraceConnect("CourseChange", MakeCallback(&
↪CourseChange));
```

It turns out that the internal code for `Config::ConnectWithoutContext` and `Config::Connect` actually 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 (recall indices start at '0'). 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 make widespread use of object aggregation. This allows us to form an association between different Objects without building a complicated inheritance tree or pre-deciding what objects will be part of a Node. Each Object in an Aggregation can be reached from the other Objects.

In our example the next path segment being walked begins with the "$" character. This indicates to the config system that the segment is the name of an object type, so a `GetObject` call should be made looking for that type. It turns out that the `MobilityHelper` used in `third.cc` arranges to Aggregate, or associate, a mobility model to each of the wireless `Nodes`. When you add the "$" 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/model/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),
                "ns3::MobilityModel::TracedCallback");
```

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

```
ns3::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 `typedef` for the callback function signature is also defined in the header file:

```
typedef void (*TracedCallback)(Ptr<const MobilityModel> model);
```

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() 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. The easy way is to connect with `Config::Connect`, instead of `Config::ConnectWithoutContext`.

8.2.3 Finding Sources

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 and the Config path, 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?"*

We'll address each of these in turn.

8.2.4 Available Sources

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 answer to the first question is found in the *ns-3* API documentation. If you go to the project web site, ns-3 project, you will find a link to "Documentation" in the navigation bar. If you select this link, you will be taken to our documentation page. There is a link to "Latest Release" that will take you to the documentation for the latest stable release of *ns-3*. If you select the "API Documentation" link, you will be taken to the *ns-3* API documentation page.

In the sidebar you should see a hierarchy that begins

- ns-3
  - ns-3 Documentation
  - All TraceSources
  - All Attributes
  - All GlobalValues

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

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

<div style="border:1px solid">

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

</div>

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

8.2.5 Config Paths

Okay, I found a trace source, how do I figure out the Config path to use when I connect to it?

If you know which object you are interested in, the "Detailed Description" section for the class will list all available trace sources. For example, starting from the list of "All TraceSources," click on the `ns3::MobilityModel` link, which will take you to the documentation for the `MobilityModel` class. Almost at the top of the page is a one line brief description of the class, ending in a link "More…". Click on this link to skip the API summary and go to the "Detailed Description" of the class. At the end of the description will be (up to) three lists:

- **Config Paths**: a list of typical Config paths for this class.
- **Attributes**: a list of all attributes supplied by this class.
- **TraceSources**: a list of all TraceSources available from this class.

First we'll discuss the Config paths.

Let's assume that you have just found the "CourseChange" trace source in the "All TraceSources" list 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 either click on the class name in the "All TraceSources" list, or find `ns3::RandomWalk2dMobilityModel` in the "Class List". Either way you should now be looking at the "ns3::RandomWalk2dMobilityModel Class Reference" page.

If you now scroll down to the "Detailed Description" section, after the summary list of class methods and attributes (or just click on the "More…" link at the end of the class brief description at the top of the page) you will see the overall documentation for the class. Continuing to scroll down, find the "Config Paths" list:

Config Paths

ns3::RandomWalk2dMobilityModel 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 trace source you are looking for is found in the base class.

Look further down in the "Detailed Description" section for the list of trace sources. You will find

No TraceSources are defined for this type.

TraceSources defined in parent class ``ns3::MobilityModel``

- **CourseChange**: The value of the position and/or velocity vector changed.

  Callback signature: `ns3::MobilityModel::TracedCallback`

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 API Documentation 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"
```

As an aside, another way to find the Config path 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, `src/mobility/examples/main-random-topology.cc` has something just waiting for you to use:

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

We'll return to this example in a moment.

8.2.6 Callback Signatures

Okay, I found a trace source and the Config path, how do I figure out what the return type and formal arguments of my callback function need to be?

The easiest way is to examine the callback signature typedef, which is given in the "Callback signature" of the trace source in the "Detailed Description" for the class, as shown above.

Repeating the "CourseChange" trace source entry from ns3::RandomWalk2dMobilityModel we have:

- **CourseChange**: The value of the position and/or velocity vector changed.

  Callback signature: ns3::MobilityModel::TracedCallback

The callback signature is given as a link to the relevant typedef, where we find

  typedef void (*TracedCallback)(Ptr<const MobilityModel> model);

  TracedCallback signature for course change notifications.

  If the callback is connected using ConnectWithoutContext omit the context argument from the signature.

  Parameters:

    [in] context The context string supplied by the Trace source.
    [in] model The MobilityModel which is changing course.

As above, to see this in use grep around in the *ns-3* codebase for an example. The example above, from src/mobility/examples/main-random-topology.cc, connects the "CourseChange" trace source to the CourseChange function in the same file:

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

Notice that this function:

- Takes a "context" string argument, which we'll describe in a minute. (If the callback is connected using the ConnectWithoutContext function the context argument will be omitted.)

- Has the MobilityModel supplied as the last argument (or only argument if ConnectWithoutContext is used).

- Returns void.

If, by chance, the callback signature hasn't been documented, and there are no examples to work from, determining the right callback function signature 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 Ptr<const MobilityModel>. For example:

```
void
CourseChange(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 template arguments:

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

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

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

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

Implementation

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" trace source. 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 303 lines fly by (I piped this through `wc` to see how bad it was). Although that may seem like a lot, 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()() 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/model/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 API Documentation for the Callback template class, though. Fortunately, there is some English:

> **Callback** template class.
>
> 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 remaining (optional) template arguments represent the type of the subsequent arguments to the callback.
>
> - up to nine arguments are supported.

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) template argument, void, represents the return type of the Callback. The second (optional) template argument, Ptr<const MobilityModel> represents the type of 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 context, 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.

8.2.7 TracedValues

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 what a TracedValue really is and how to find the return type and formal arguments for the callback.

As we mentioned, 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 itself around, rather than passing the address of the object. We extend that requirement to include the full set of assignment-style operators that are pre-defined for plain-old-data (POD) types:

operator= (assignment)	
operator*=	operator/=
operator+=	operator-=
operator++ (both prefix and postfix)	
operator-- (both prefix and postfix)	
operator<<=	operator>>=
operator&=	operator|=
operator%=	operator^=

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

The `TracedValue<>` declaration we saw above provides the infrastructure that overloads the operators mentioned above and drives the callback process. On use of any of the operators above with a `TracedValue` it will provide both the old and the new value of that variable, in this case an `int32_t` value. By inspection of the `TracedValue` declaration, we know the trace sink function will have arguments (`int32_t oldValue`, `int32_t newValue`). The return type for a `TracedValue` callback function is always `void`, so the expected callback signature for the sink function `traceSink` will be:

```
void (* traceSink)(int32_t oldValue, int32_t newValue);
```

The `.AddTraceSource` in the `GetTypeId` method provides the "hooks" used for connecting the trace source to the outside world through the Config system. We already discussed the first three arguments to `AddTraceSource`: the Attribute name for the Config system, a help string, and the address of the TracedValue class data member.

The final string argument, "ns3::TracedValueCallback::Int32" in the example, is the name of a `typedef` for the callback function signature. We require these signatures to be defined, and give the fully qualified type name to `AddTraceSource`, so the API documentation can link a trace source to the function signature. For TracedValue the signature is straightforward; for TracedCallbacks we've already seen the API docs really help.

8.3 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`.

8.3.1 Available Sources

The first thing to think about is how we want to get the data out. What is it that we need to trace? So let's consult "All Trace Sources" list to see what we have to work with. Recall that this is found in the *ns-3* API Documentation. If you scroll through the list, you will eventually find:

ns3::TcpSocketBase

- **CongestionWindow**: The TCP connection's congestion window
- **SlowStartThreshold**: TCP slow start threshold (bytes)

It turns out that the *ns-3* TCP implementation lives (mostly) in the file `src/internet/model/tcp-socket-base.cc` while congestion control variants are in files such as `src/internet/model/tcp-bic.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.

Bringing up the class documentation for `TcpSocketBase` and skipping to the list of TraceSources you will find

TraceSources

- **CongestionWindow**: The TCP connection's congestion window

  Callback signature: **ns3::TracedValueCallback::Uint32**

Clicking on the callback `typedef` link we see the signature you now know to expect:

```
typedef void(* ns3::TracedValueCallback::Int32)(int32_t oldValue, int32_t newValue)
```

You should now understand this code completely. If we have a pointer to the `TcpSocketBase` object, 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`. And we know that we have to provide a callback function with that signature.

8.3.2 Finding Examples

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 `TcpSocketBase`, 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()`). 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`.

8.3.3 Dynamic Trace Sources

The `fifth.cc` example demonstrates an extremely important rule that you must understand before using any kind of trace source: you must ensure that the target of a `Config::Connect` 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 execution phases that exist in any *ns-3* script. The first phase is sometimes called "Configuration Time" or "Setup Time," and exists during the period when the `main` function of your script is running, but before `Simulator::Run` is called. The second phase is sometimes called "Simulation Time" and exists 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 the "Teardown Phase," which is when the structures and objects created during setup are 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?). As a result, during the configuration phase you can't connect a trace source to a trace sink if one of them is created dynamically during the simulation.

The two solutions to this conundrum are

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 `TutorialApp` `Application`, the entire purpose of which is to take a `Socket` as a parameter.

8.3.4 Walkthrough: `fifth.cc`

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:

```
/*
 * SPDX-License-Identifier: GPL-2.0-only
 */

#include "tutorial-app.h"

#include "ns3/applications-module.h"
#include "ns3/core-module.h"
#include "ns3/internet-module.h"
#include "ns3/network-module.h"
#include "ns3/point-to-point-module.h"

#include <fstream>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("FifthScriptExample");
```

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.

Previous versions of *ns-3* declared a custom application called `MyApp` for use in this program. Current versions of *ns-3* have moved this to a separate header file (`tutorial-app.h`) and implementation file (`tutorial-app.cc`). This simple application allows the `Socket` to be created at configuration time.

```
/**
 * Tutorial - a simple Application sending packets.
 */
class TutorialApp : public Application
{
  public:
    TutorialApp();
    ~TutorialApp() override;

    /**
     * Register this type.
     * \return The TypeId.
     */
    static TypeId GetTypeId();

    /**
     * Setup the socket.
     * \param socket The socket.
     * \param address The destination address.
     * \param packetSize The packet size to transmit.
     * \param nPackets The number of packets to transmit.
     * \param dataRate the data rate to use.
     */
    void Setup(Ptr<Socket> socket,
               Address address,
               uint32_t packetSize,
               uint32_t nPackets,
```

```
                DataRate dataRate);

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

    /// Schedule a new transmission.
    void ScheduleTx();
    /// Send a packet.
    void SendPacket();

    Ptr<Socket> m_socket;    //!< The transmission socket.
    Address m_peer;          //!< The destination address.
    uint32_t m_packetSize;   //!< The packet size.
    uint32_t m_nPackets;     //!< The number of packets to send.
    DataRate m_dataRate;     //!< The data rate to use.
    EventId m_sendEvent;     //!< Send event.
    bool m_running;          //!< True if the application is running.
    uint32_t m_packetsSent;  //!< The number of packets sent.
};
```

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

Starting/Stopping Applications

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) familiar lines of an *ns-3* script:

```
ApplicationContainer apps = ...
apps.Start(Seconds(1));
apps.Stop(Seconds(10));
```

The application container code (see `src/network/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 `TutorialApp`, 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 `TutorialApp` you will find that `TutorialApp::StartApplication` does the initial `Bind`, and `Connect` on the socket, and then starts data flowing by calling `TutorialApp::SendPacket`. `TutorialApp::StopApplication` stops generating packets by cancelling any pending send events then closes 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/network/model/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/network/model/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 idiom in *ns-3*. So, take a look at `NodeListPriv::Add`. There you will find,

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

This tells you that whenever a Node is created in a simulation, as a side-effect, a call to that node's `Initialize` 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::Initialize` at time zero to advise a new Node that the simulation has started. If you look in `src/network/model/node.h` you will, however, not find a method called `Node::Initialize`. It turns out that the `Initialize` 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/model/object.cc` next and search for `Object::Initialize`. This code is not as straightforward as you might have expected since *ns-3* `Objects` support aggregation. The code in `Object::Initialize` then loops through all of the objects that have been aggregated together and calls their `DoInitialize` method. This is another idiom that is very common in *ns-3*, sometimes called the "template design pattern.": a public non-virtual API method, which stays constant across implementations, and that calls a private virtual 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::DoInitialize` method in `src/network/model/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->Initialize` and `application->Initialize` 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::DoInitialize` is called. Take a look at `src/network/model/application.cc` and you will find:

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

(continues on next page)

```
    Object::DoInitialize();
}
```

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::Initialize` method of the newly added Node to be called when the simulation starts. Since a Node inherits from `Object`, this calls the `Object::Initialize` method on the Node which, in turn, calls the `DoInitialize` methods on all of the `Objects` aggregated to the Node (think mobility models). Since the Node `Object` has overridden `DoInitialize`, that method is called when the simulation starts. The `Node::DoInitialize` method calls the `Initialize` methods of all of the `Applications` on the node. Since `Applications` are also `Objects`, this causes `Application::DoInitialize` to be called. When `Application::DoInitialize` 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 TutorialApp Application

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

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

TutorialApp::~TutorialApp()
{
    m_socket = nullptr;
}
```

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

```
void
TutorialApp::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 `TcpSocketBase`) and hook its "CongestionWindow" trace source before passing it to the `Setup` method.

```
void
TutorialApp::StartApplication()
{
    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
TutorialApp::StopApplication()
{
    m_running = false;

    if (m_sendEvent.IsPending())
    {
        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 `IsPending` will return `true`. In this code, if `IsPending()` 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
TutorialApp::SendPacket()
{
    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
TutorialApp::ScheduleTx()
{
    if (m_running)
    {
        Time tNext(Seconds(m_packetSize * 8 / static_cast<double>(m_dataRate.
→GetBitRate())));
        m_sendEvent = Simulator::Schedule(tNext, &TutorialApp::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.

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/point-to-point/model/point-to-point-net-device.cc`) you will see that this trace source refers to `PointToPointNetDevice::m_phyRxDropTrace`. If you then look in `src/point-to-point/model/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>` (assuming we use `ConnectWithoutContext`) – just what we have above.

Main Program

The main function starts off by configuring the TCP type to use a legacy `NewReno` congestion control variant, with what is called the `classic` TCP loss recovery mechanism. When this tutorial program was originally written, these were the default TCP configurations, but over time, *ns-3* TCP has evolved to use the current Linux TCP defaults of `Cubic` and `Prr` loss recovery. The first statements also configure the command-line argument processing.

```
int
main(int argc, char* argv[])
{
    CommandLine cmd(__FILE__);
    cmd.Parse(argc, argv);

    // In the following three lines, TCP NewReno is used as the congestion
    // control algorithm, the initial congestion window of a TCP connection is
    // set to 1 packet, and the classic fast recovery algorithm is used. Note
    // that this configuration is used only to demonstrate how TCP parameters
    // can be configured in ns-3. Otherwise, it is recommended to use the default
    // settings of TCP in ns-3.
    Config::SetDefault("ns3::TcpL4Protocol::SocketType", StringValue("ns3::TcpNewReno"));
    Config::SetDefault("ns3::TcpSocket::InitialCwnd", UintegerValue(1));
    Config::SetDefault("ns3::TcpL4Protocol::RecoveryType",
                       TypeIdValue(TypeId::LookupByName("ns3::TcpClassicRecovery")));
```

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

```
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 = CreateObject<RateErrorModel>();
em->SetAttribute("ErrorRate", DoubleValue(0.00001));
devices.Get(1)->SetAttribute("ReceiveErrorModel", PointerValue(em));
```

The above code instantiates a `RateErrorModel` Object, and we set the "ErrorRate" `Attribute` to the desired value. 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<TutorialApp> app = CreateObject<TutorialApp>();
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 `TutorialApp` – 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 `TutorialApp 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` drop event to our `RxDrop` 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.

8.3.5 Running `fifth.cc`

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

```
$ ./ns3 run fifth
1.00419 536
1.0093  1072
1.01528 1608
1.02167 2144
```

(continues on next page)

```
...
1.11319 8040
1.12151 8576
1.12983 9112
RxDrop at 1.13696
...
```

You can probably see immediately a downside of using prints of any kind in your traces. We get those extraneous ns3 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`:

```
$ ./ns3 run fifth > cwnd.dat 2>&1
```

Now edit up "cwnd.dat" in your favorite editor and remove the ns3 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
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:

8.3.6 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 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.
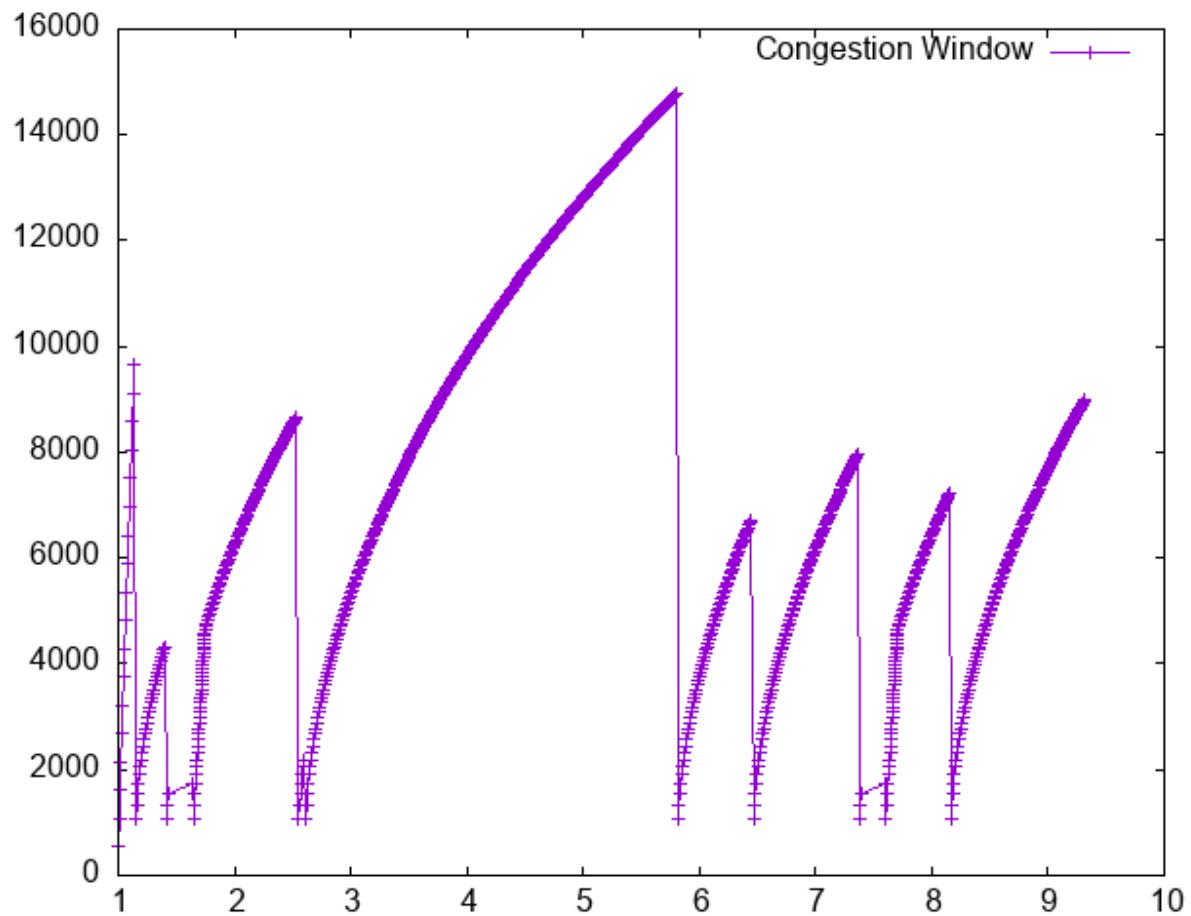
Walkthrough: `sixth.cc`

Let's take a look at the changes required to go from `fifth.cc` to `sixth.cc`. Open `examples/tutorial/sixth.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 << std::endl;
}

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 `std::ostream` is marked private. This means that `std::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::endl;
```

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 `std::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 as the first argument 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 truncated (contents deleted) 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 familiar 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/network/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 heavyweight 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 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/network/utils/pcap-file-wrapper.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/network/model/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 build and run this example,

```
$ ./ns3 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       0       536
1.0093  536     1072
1.01528 1072    1608
1.02167 1608    2144
...
9.69256 5149    5204
9.89311 5204    5259
```

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 view it with `tcpdump`.

```
reading from file sixth.pcap, link-type PPP (PPP)
1.136956 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 17177:17681, ack 1, win 32768,
→ options [TS val 1133 ecr 1127,eol], length 504
1.403196 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 33280:33784, ack 1, win 32768,
→ options [TS val 1399 ecr 1394,eol], length 504
...
7.426220 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 785704:786240, ack 1, win␣
→32768, options [TS val 7423 ecr 7421,eol], length 536
9.630693 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 882688:883224, ack 1, win␣
→32768, options [TS val 9620 ecr 9618,eol], length 536
```

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 << std::endl;
}

...

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));
```

8.4 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 when it 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`.

8.4.1 Device Helpers

PCAP

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/network/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 explicitFilename) = 0;
```

The signature of this method reflects the device-centric view of the situation at this level. All of the public methods inherited from class `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
↪explicitFilename = false);
```

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.

Methods

```
void EnablePcap(std::string prefix, Ptr<NetDevice> nd, bool promiscuous = false, bool
↪explicitFilename = false);
void EnablePcap(std::string prefix, std::string ndName, bool promiscuous = false, bool
↪explicitFilename = false);
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 API Documentation 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");
```

Filenames

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␣
↪explicitFilename = false);
void EnablePcap(std::string prefix, std::string ndName, bool promiscuous = false, bool␣
↪explicitFilename = false);
```

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.

ASCII

The behavior of the ASCII trace helper `mixin` is substantially similar to the PCAP version. Take a look at `src/network/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 `EnableAsciiInternal` correctly.

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 API Documentation for class `AsciiTraceHelperForDevice` 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` suffix 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 ASCII 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 ASCII 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.

Filenames

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.

8.4.2 Protocol Helpers

PCAP

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/network/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 = false);
```

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.

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 = false);
void EnablePcapIpv4(std::string prefix, std::string ipv4Name, uint32_t interface, bool␣
→explicitFilename = false);
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 API Documentation 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");
```

Filenames

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.

ASCII

The behavior of the ASCII trace helpers is substantially similar to the PCAP case. Take a look at `src/network/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 = false);
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 `EnableAsciiIpv4Internal` correctly.

Methods

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

void EnableAsciiIpv4(std::string prefix, std::string ipv4Name, uint32_t interface, bool␣
→explicitFilename = false);
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 API Documentation 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 ASCII 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.

Filenames

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.

8.5 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

DATA COLLECTION

Our final tutorial chapter introduces some components that were added to *ns-3* in version 3.18, and that are still under development. This tutorial section is also a work-in-progress.

9.1 Motivation

One of the main points of running simulations is to generate output data, either for research purposes or simply to learn about the system. In the previous chapter, we introduced the tracing subsystem and the example `sixth.cc.` from which PCAP or ASCII trace files are generated. These traces are valuable for data analysis using a variety of external tools, and for many users, such output data is a preferred means of gathering data (for analysis by external tools).

However, there are also use cases for more than trace file generation, including the following:

- generation of data that does not map well to PCAP or ASCII traces, such as non-packet data (e.g. protocol state machine transitions),

- large simulations for which the disk I/O requirements for generating trace files is prohibitive or cumbersome, and

- the need for *online* data reduction or computation, during the course of the simulation. A good example of this is to define a termination condition for the simulation, to tell it when to stop when it has received enough data to form a narrow-enough confidence interval around the estimate of some parameter.

The *ns-3* data collection framework is designed to provide these additional capabilities beyond trace-based output. We recommend that the reader interested in this topic consult the *ns-3* Manual for a more detailed treatment of this framework; here, we summarize with an example program some of the developing capabilities.

9.2 Example Code

The tutorial example `examples/tutorial/seventh.cc` resembles the `sixth.cc` example we previously reviewed, except for a few changes. First, it has been enabled for IPv6 support with a command-line option:

```
CommandLine cmd;
cmd.AddValue("useIpv6", "Use Ipv6", useV6);
cmd.Parse(argc, argv);
```

If the user specifies `useIpv6`, option, the program will be run using IPv6 instead of IPv4. The `help` option, available on all *ns-3* programs that support the CommandLine object as shown above, can be invoked as follows (please note the use of double quotes):

```
./ns3 run "seventh --help"
```

which produces:

```
ns3-dev-seventh-debug [Program Arguments] [General Arguments]

Program Arguments:
    --useIpv6:  Use Ipv6 [false]

General Arguments:
    --PrintGlobals:              Print the list of globals.
    --PrintGroups:               Print the list of groups.
    --PrintGroup=[group]:        Print all TypeIds of group.
    --PrintTypeIds:              Print all TypeIds.
    --PrintAttributes=[typeid]:  Print all attributes of typeid.
    --PrintHelp:                 Print this help message.
```

This default (use of IPv4, since useIpv6 is false) can be changed by toggling the boolean value as follows:

```
./ns3 run "seventh --useIpv6=1"
```

and have a look at the pcap generated, such as with tcpdump:

```
tcpdump -r seventh.pcap -nn -tt
```

This has been a short digression into IPv6 support and the command line, which was also introduced earlier in this tutorial. For a dedicated example of command line usage, please see src/core/examples/command-line-example.cc.

Now back to data collection. In the examples/tutorial/ directory, type the following command: diff -u sixth.cc seventh.cc, and examine some of the new lines of this diff:

```
+  std::string probeType;
+  std::string tracePath;
+  if (useV6 == false)
+    {
  ...
+      probeType = "ns3::Ipv4PacketProbe";
+      tracePath = "/NodeList/*/$ns3::Ipv4L3Protocol/Tx";
+    }
+  else
+    {
  ...
+      probeType = "ns3::Ipv6PacketProbe";
+      tracePath = "/NodeList/*/$ns3::Ipv6L3Protocol/Tx";
+    }
 ...
+    // Use GnuplotHelper to plot the packet byte count over time
+    GnuplotHelper plotHelper;
+
+    // Configure the plot.  The first argument is the file name prefix
+    // for the output files generated.  The second, third, and fourth
+    // arguments are, respectively, the plot title, x-axis, and y-axis labels
+    plotHelper.ConfigurePlot("seventh-packet-byte-count",
+                             "Packet Byte Count vs. Time",
+                             "Time(Seconds)",
+                             "Packet Byte Count");
+
+    // Specify the probe type, trace source path (in configuration namespace), and
```

Chapter 9. Data Collection

```
+    // probe output trace source ("OutputBytes") to plot.  The fourth argument
+    // specifies the name of the data series label on the plot.  The last
+    // argument formats the plot by specifying where the key should be placed.
+    plotHelper.PlotProbe(probeType,
+                         tracePath,
+                         "OutputBytes",
+                         "Packet Byte Count",
+                         GnuplotAggregator::KEY_BELOW);
+
+    // Use FileHelper to write out the packet byte count over time
+    FileHelper fileHelper;
+
+    // Configure the file to be written, and the formatting of output data.
+    fileHelper.ConfigureFile("seventh-packet-byte-count",
+                             FileAggregator::FORMATTED);
+
+    // Set the labels for this formatted output file.
+    fileHelper.Set2dFormat("Time (Seconds) = %.3e\tPacket Byte Count = %.0f");
+
+    // Specify the probe type, probe path (in configuration namespace), and
+    // probe output trace source ("OutputBytes") to write.
+    fileHelper.WriteProbe(probeType,
+                          tracePath,
+                          "OutputBytes");
+
     Simulator::Stop(Seconds(20));
     Simulator::Run();
     Simulator::Destroy();
```

The careful reader will have noticed, when testing the IPv6 command line attribute above, that `seventh.cc` had created a number of new output files:

```
seventh-packet-byte-count-0.txt
seventh-packet-byte-count-1.txt
seventh-packet-byte-count.dat
seventh-packet-byte-count.plt
seventh-packet-byte-count.png
seventh-packet-byte-count.sh
```

These were created by the additional statements introduced above; in particular, by a GnuplotHelper and a FileHelper. This data was produced by hooking the data collection components to *ns-3* trace sources, and marshaling the data into a formatted `gnuplot` and into a formatted text file. In the next sections, we'll review each of these.

9.3 GnuplotHelper

The GnuplotHelper is an *ns-3* helper object aimed at the production of `gnuplot` plots with as few statements as possible, for common cases. It hooks *ns-3* trace sources with data types supported by the data collection system. Not all *ns-3* trace sources data types are supported, but many of the common trace types are, including TracedValues with plain old data (POD) types.

Let's look at the output produced by this helper:

```
seventh-packet-byte-count.dat
seventh-packet-byte-count.plt
seventh-packet-byte-count.sh
```

The first is a gnuplot data file with a series of space-delimited timestamps and packet byte counts. We'll cover how this particular data output was configured below, but let's continue with the output files. The file `seventh-packet-byte-count.plt` is a gnuplot plot file, that can be opened from within gnuplot. Readers who understand gnuplot syntax can see that this will produce a formatted output PNG file named `seventh-packet-byte-count.png`. Finally, a small shell script `seventh-packet-byte-count.sh` runs this plot file through gnuplot to produce the desired PNG (which can be viewed in an image editor); that is, the command:

```
sh seventh-packet-byte-count.sh
```

will yield `seventh-packet-byte-count.png`. Why wasn't this PNG produced in the first place? The answer is that by providing the plt file, the user can hand-configure the result if desired, before producing the PNG.

The PNG image title states that this plot is a plot of "Packet Byte Count vs. Time", and that it is plotting the probed data corresponding to the trace source path:

```
/NodeList/*/$ns3::Ipv6L3Protocol/Tx
```

Note the wild-card in the trace path. In summary, what this plot is capturing is the plot of packet bytes observed at the transmit trace source of the Ipv6L3Protocol object; largely 596-byte TCP segments in one direction, and 60-byte TCP acks in the other (two node trace sources were matched by this trace source).

How was this configured? A few statements need to be provided. First, the GnuplotHelper object must be declared and configured:

```
+   // Use GnuplotHelper to plot the packet byte count over time
+   GnuplotHelper plotHelper;
+
+   // Configure the plot.  The first argument is the file name prefix
+   // for the output files generated.  The second, third, and fourth
+   // arguments are, respectively, the plot title, x-axis, and y-axis labels
+   plotHelper.ConfigurePlot("seventh-packet-byte-count",
+                            "Packet Byte Count vs. Time",
+                            "Time (Seconds)",
+                            "Packet Byte Count");
```

To this point, an empty plot has been configured. The filename prefix is the first argument, the plot title is the second, the x-axis label the third, and the y-axis label the fourth argument.

The next step is to configure the data, and here is where the trace source is hooked. First, note above in the program we declared a few variables for later use:

```
+   std::string probeType;
+   std::string tracePath;
```

```
+  probeType = "ns3::Ipv6PacketProbe";
+  tracePath = "/NodeList/*/$ns3::Ipv6L3Protocol/Tx";
```

We use them here:

```
+  // Specify the probe type, trace source path (in configuration namespace), and
+  // probe output trace source ("OutputBytes") to plot.  The fourth argument
+  // specifies the name of the data series label on the plot.  The last
+  // argument formats the plot by specifying where the key should be placed.
+  plotHelper.PlotProbe(probeType,
+                       tracePath,
+                       "OutputBytes",
+                       "Packet Byte Count",
+                       GnuplotAggregator::KEY_BELOW);
```

The first two arguments are the name of the probe type and the trace source path. These two are probably the hardest to determine when you try to use this framework to plot other traces. The probe trace here is the `Tx` trace source of class `Ipv6L3Protocol`. When we examine this class implementation (`src/internet/model/ipv6-l3-protocol.cc`) we can observe:

```
.AddTraceSource("Tx", "Send IPv6 packet to outgoing interface.",
                MakeTraceSourceAccessor(&Ipv6L3Protocol::m_txTrace))
```

This says that `Tx` is a name for variable `m_txTrace`, which has a declaration of:

```
/**
 * \brief Callback to trace TX (transmission) packets.
 */
TracedCallback<Ptr<const Packet>, Ptr<Ipv6>, uint32_t> m_txTrace;
```

It turns out that this specific trace source signature is supported by a Probe class (what we need here) of class Ipv6PacketProbe. See the files `src/internet/model/ipv6-packet-probe.{h,cc}`.

So, in the PlotProbe statement above, we see that the statement is hooking the trace source (identified by path string) with a matching *ns-3* Probe type of `Ipv6PacketProbe`. If we did not support this probe type (matching trace source signature), we could have not used this statement (although some more complicated lower-level statements could have been used, as described in the manual).

The Ipv6PacketProbe exports, itself, some trace sources that extract the data out of the probed Packet object:

```
TypeId
Ipv6PacketProbe::GetTypeId()
{
  static TypeId tid = TypeId("ns3::Ipv6PacketProbe")
    .SetParent<Probe>()
    .SetGroupName("Stats")
    .AddConstructor<Ipv6PacketProbe>()
    .AddTraceSource("Output",
                    "The packet plus its IPv6 object and interface that serve as the␣
→output for this probe",
                    MakeTraceSourceAccessor(&Ipv6PacketProbe::m_output))
    .AddTraceSource("OutputBytes",
                    "The number of bytes in the packet",
                    MakeTraceSourceAccessor(&Ipv6PacketProbe::m_outputBytes))
```

```
  ;
  return tid;
}
```

The third argument of our PlotProbe statement specifies that we are interested in the number of bytes in this packet; specifically, the "OutputBytes" trace source of Ipv6PacketProbe. Finally, the last two arguments of the statement provide the plot legend for this data series ("Packet Byte Count"), and an optional gnuplot formatting statement (GnuplotAggregator::KEY_BELOW) that we want the plot key to be inserted below the plot. Other options include NO_KEY, KEY_INSIDE, and KEY_ABOVE.

9.4 Supported Trace Types

The following traced values are supported with Probes as of this writing:

TracedValue type	Probe type	File
double	DoubleProbe	stats/model/double-probe.h
uint8_t	Uinteger8Probe	stats/model/uinteger-8-probe.h
uint16_t	Uinteger16Probe	stats/model/uinteger-16-probe.h
uint32_t	Uinteger32Probe	stats/model/uinteger-32-probe.h
bool	BooleanProbe	stats/model/uinteger-16-probe.h
ns3::Time	TimeProbe	stats/model/time-probe.h

The following TraceSource types are supported by Probes as of this writing:

TracedSource type	Probe type	Probe outputs	File
Ptr<const Packet>	PacketProbe	OutputBytes	network/utils/packet-probe.h
Ptr<const Packet>, Ptr<Ipv4>, uint32_t	Ipv4PacketProbe	OutputBytes	internet/model/ipv4-packet-probe.h
Ptr<const Packet>, Ptr<Ipv6>, uint32_t	Ipv6PacketProbe	OutputBytes	internet/model/ipv6-packet-probe.h
Ptr<const Packet>, Ptr<Ipv6>, uint32_t	Ipv6PacketProbe	OutputBytes	internet/model/ipv6-packet-probe.h
Ptr<const Packet>, const Address&	ApplicationPacketProbe	OutputBytes	applications/model/application-packet-probe.h

As can be seen, only a few trace sources are supported, and they are all oriented towards outputting the Packet size (in bytes). However, most of the fundamental data types available as TracedValues can be supported with these helpers.

9.5 FileHelper

The FileHelper class is just a variation of the previous GnuplotHelper example. The example program provides formatted output of the same timestamped data, such as follows:

```
Time (Seconds) = 9.312e+00    Packet Byte Count = 596
Time (Seconds) = 9.312e+00    Packet Byte Count = 564
```

Two files are provided, one for node "0" and one for node "1" as can be seen in the filenames. Let's look at the code piece-by-piece:

```
+    // Use FileHelper to write out the packet byte count over time
+    FileHelper fileHelper;
+
+    // Configure the file to be written, and the formatting of output data.
+    fileHelper.ConfigureFile("seventh-packet-byte-count",
+                             FileAggregator::FORMATTED);
```

The file helper file prefix is the first argument, and a format specifier is next. Some other options for formatting include SPACE_SEPARATED, COMMA_SEPARATED, and TAB_SEPARATED. Users are able to change the formatting (if FORMATTED is specified) with a format string such as follows:

```
+
+    // Set the labels for this formatted output file.
+    fileHelper.Set2dFormat("Time (Seconds) = %.3e\tPacket Byte Count = %.0f");
```

Finally, the trace source of interest must be hooked. Again, the probeType and tracePath variables in this example are used, and the probe's output trace source "OutputBytes" is hooked:

```
+
+    // Specify the probe type, trace source path (in configuration namespace), and
+    // probe output trace source ("OutputBytes") to write.
+    fileHelper.WriteProbe(probeType,
+                          tracePath,
+                          "OutputBytes");
+
```

The wildcard fields in this trace source specifier match two trace sources. Unlike the GnuplotHelper example, in which two data series were overlaid on the same plot, here, two separate files are written to disk.

9.6 Summary

Data collection support is new as of ns-3.18, and basic support for providing time series output has been added. The basic pattern described above may be replicated within the scope of support of the existing probes and trace sources. More capabilities including statistics processing will be added in future releases.

CONCLUSION

10.1 Futures

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

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.

10.2 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. Readers who want to learn more are encouraged to read the following additional documentation:

- The *ns-3* manual
- The *ns-3* model library documentation
- The *ns-3* Doxygen (API documentation)
- The *ns-3* wiki

– The *ns-3* development team.