ns-3 Installation Guide

Release ns-3-dev

ns-3 project

Apr 29, 2025

CONTENTS

This is the *ns-3 Installation Guide*. 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
  - Installation Guide *(this document)*
  - 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/installation` directory of ns-3's source code. Source file column width is 100 columns.

CHAPTER

ONE

OVERVIEW

This guide documents the different ways that users can download, build, and install *ns-3* from source code. All of these actions (download, build, install) have variations or options, and can be customized or extended by users; this document attempts to inform readers about different possibilities.

Please note a few important details:

- *ns-3* is supported for Linux, macOS, and Windows systems. Linux is the primary system supported, and (almost) all *ns-3* features are supported on Linux. However, most features can also be used on macOS and Windows. Windows is probably the least used and least supported system.
- *ns-3* has minimal prerequisites for its most basic installation; namely, a **C++** compiler, **Python3** support, the **CMake** build system, and at least one of **make**, **ninja**, or **Xcode** build systems. However, some users will want to install optional packages to make use of the many optional extensions.
- *ns-3* installation requirements vary from release to release, and also as underlying operating systems evolve. This document is version controlled, so if you are using the *ns-3.X* release, try to read the *ns-3.X* version of this document. Even with this guidance, you may encounter issues if you are trying to use an old version of *ns-3* on a much newer system. For instance, *ns-3* versions until around 2020 relied on Python2, which is now end-of-life and not installed by default on many Linux distributions. Compilers also become more pedantic and issue more warnings as they evolve. Solutions to some of these forward compatibility problems exist and are discussed herein or might be found in the ns-3-users mailing list archives.

If you find any stale or inaccurate information in this document, please report it to maintainers, on our Zulip chat, in the GitLab.com Issue tracker, (or better yet, a patch to fix in the Merge Request tracker), or on our developers mailing list.

We also will accept patches to this document to provide installation guidance for the FreeBSD operating system and other operating systems being used in practice.

1.1 Software organization

ns-3 is a set of C++ libraries (usually compiled as shared libraries) that can be used by C++ or Python programs to construct simulation scenarios and execute simulations. Users can also write programs that link other C++ shared libraries (or import other Python modules). Users can choose to use a subset of the available libraries; only the `core` library is strictly required.

ns-3 uses the CMake build system (until release 3.36, the Waf build system was used). It can be built from command-line or via a code editor program.

Most users write C++ ns-3 programs; Python support is less frequently used. As of *ns-3.37*, *ns-3* uses *cppyy* to generate runtime Python bindings, and *ns-3* is available in the Pip repositories as of ns-3.39 release.

Many users may be familiar with how software is packaged and installed on Linux and other systems using package managers. For example, to install a given Linux development library such as OpenSSL, a package manager is typically

used (e.g., `apt install openssl libssl-dev`), which leads to shared libraries being installed under `/usr/lib/`, development headers being installed under `/usr/include/`, etc. Programs that wish to use these libraries must link the system libraries and include the development headers. *ns-3* is capabile of being installed in exactly the same way, and some downstream package maintainers have packaged *ns-3* for some systems such as Ubuntu. However, as of this writing, the *ns-3* project has not prioritized or standardized such package distribution, favoring instead to recommend a *source download* without a system-level install. This is mainly because most *ns-3* users prefer to slightly or extensively edit or extend the *ns-3* libraries, or to build them in specific ways (for debugging, or optimized for large-scale simulation campaign). Our build system provides an `install` command that can be used to install libraries and headers to system locations (usually requiring administrator privileges), but usually the libraries are just built and used from within the *ns-3* `build` directory.

1.2 Document organization

The rest of this document is organized as follows:

- *Quick Start* (for any operating system)
- *System Prerequisites* of requirements, recommendations, and optional prerequisites
- *Linux* installation
- *macOS* installation
- *Windows* installation
- *Installing Bake* installation

CHAPTER

TWO

QUICK START

This chapter summarizes the *ns-3* installation process for C++ users interested in trying a generic install of the main simulator. Python bindings installation is not covered.

Some of this chapter is redundant with the *ns-3* tutorial, which also covers similar steps.

The steps are:

1. Download a source archive, or make a git clone, of *ns-3* to a location on your file system (usually somewhere under your home directory).
2. Use a C++ compiler to compile the software into a set of shared libraries, executable example programs, and tests

ns-3 uses the CMake build system to manage the C++ compilation, and CMake itself calls on a lower-level build system such as `make` to perform the actual compilation.

2.1 Prerequisites

Make sure that your system has these prerequisites. Download can be via either `git` or via source archive download (via a web browser, `wget`, or `curl`).

Purpose	Tool	Minimum Version
Download	`git` (for Git download) or `tar` and `bunzip2` (for Web download)	No minimum version No minimum version
Compiler	`g++` or `clang++`	>= 10 >= 11
Configuration	`python3`	>= 3.8
Build system	`cmake`, and at least one of: `make`, `ninja`, or `Xcode`	>= 3.13 No minimum version

Note: If you are using an older version of ns-3, other tools may be needed (such as `python2` instead of `python3` and `Waf` instead of `cmake`). Check the file `RELEASE_NOTES` in the top-level directory for requirements for older releases.

From the command line, you can check the version of each of the above tools with version requirements as follows:

Tool	Version check command
g++	$ g++ --version
clang++	$ clang++ --version
python3	$ python3 -V
cmake	$ cmake --version

2.2 Download

There are two main options:

1. Download a release tarball. This will unpack to a directory such as `ns-allinone-3.44` containing *ns-3* and some other programs. Below is a command-line download using `wget`, but a browser download will also work:

```
$ wget https://www.nsnam.org/releases/ns-allinone-3.44.tar.bz2
$ tar xfj ns-allinone-3.44.tar.bz2
$ cd ns-allinone-3.44/ns-3.44
```

2. Clone *ns-3* from the Git repository. The `ns-3-allinone` can be cloned, as well as `ns-3-dev` by itself. Below, we illustrate the latter:

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

Note that if you select option 1), your directory name will contain the release number. If you clone *ns-3*, your directory will be named `ns-3-dev`. By default, Git will check out the *ns-3* `master` branch, which is a development branch. All *ns-3* releases are tagged in Git, so if you would then like to check out a past release, you can do so as follows:

```
$ git checkout -b ns-3.44-release ns-3.44
```

In this quick-start, we are omitting download and build instructions for optional *ns-3* modules, the `NetAnim` animator, Python bindings, and `NetSimulyzer`. The ns-3 Tutorial has some instructions on optional components, or else the documentation associated with the extension should be consulted.

Moreover, in this guide we will assume that you are using ns-3.36 or later. Earlier versions had different configuration, build, and run command and options.

2.3 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
```

Depending on how fast your CPU is, the configuration command can take anywhere from a few seconds to a minute.

You should see some output such as below, if successful:

```
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                     wimax

Modules that cannot be built:
brite                    click                    mpi
openflow                 visualizer
```

Do not be concerned about the list of modules that cannot be built; these modules are all optional and require some extra dependencies, but are not needed for initial exploration of *ns-3*.

Then, use the `ns3` program to build the *ns-3* module libraries and executables:

```
$ ./ns3 build
```

Build times vary based on the number of CPU cores, the speed of the CPU and memory, and the mode of the build (whether debug mode, which is faster, or the default or optimized modes, which are slower). Additional configuration (not covered here) can be used to limit the scope of the build, and the `ccache`, if installed, can speed things up. In general, plan on the build taking a few minutes on faster workstations.

At the end of the build, if successful, the output will report on the underlying `cmake` build command that was invoked by the `ns3` program. Once the build is complete, you can run the unit tests to check your build:

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

This command should run several hundred unit tests. If they pass, you have made a successful initial build of *ns-3*. Read further in this manual for instructions about building optional components, or else consult the *ns-3* Tutorial or other documentation to get started with the base *ns-3*.

If you prefer to code with an code editor, consult the documentation in the *ns-3* Manual on Working with CMake, since CMake enables *ns-3* integration with a variety of code editors, including:

- JetBrains's CLion
- Microsoft Visual Studio and Visual Studio Code
- Apple's XCode
- CodeBlocks
- Eclipse CDT4

2.4 Installing ns-3

Most users do not install *ns-3* libraries to typical system library directories; they instead just leave the libraries in the `build` directory, and the `ns3` Python program will find these libraries. However, it is possible to perform an installation step– `ns3 install`– with the following caveats.

The location of the installed libraries is set by the `--prefix` option specified at the configure step. The prefix defaults to `/usr/local`. For a given `--prefix=$PREFIX`, the installation step will install headers to a `$PREFIX/include` directory, libraries and pkgconfig files to a `$PREFIX/lib` directory, and a few binaries to a `$PREFIX/libexec` directory. For example, `./ns3 configure --prefix=/tmp`, followed by `./ns3 build` and `./ns3 install`, will lead to files being installed in `/tmp/include`, `/tmp/lib`, and `/tmp/libexec`.

Note that the `ns3` script prevents running the script as root (or as a sudo user). As a result, with the default prefix of `/usr/local`, the installation will fail unless the user has write privileges in that directory. Attempts to force this with `sudo ./ns3 install` will fail due to a check in the `ns3` program that prevents running as root. This check was installed by *ns-3* maintainers for the safety of novice users who may run `./ns3` in a root shell and later in a normal shell, and become confused about errors resulting in lack of privileges to modify files. For users who know what they are doing and who want to install to a privileged directory, users can comment out the statement `refuse_run_as_root()` in the `ns3` program (around line 1400), and then run `sudo ./ns3 install`.

CHAPTER

THREE

SYSTEM PREREQUISITES

This chapter describes the various required, recommended, and optional system prerequisites for installing and using *ns-3*. Optional prerequisites depend on whether an optional feature of *ns-3* is desired by the user. The chapter is written to be generally applicable to all operating systems, and subsequent chapters describe the specific packages for different operating systems.

The list of requirements depends on which version of ns-3 you are trying to build, and on which extensions you need.

Note: "Do I need to install all of these packages?" Some users want to install everything so that their configuration output shows that every feature is enabled. However, there is no real need to install prerequisites related to features you are not yet using; you can always come back later and install more prerequisites as needed. The build system should warn you if you are missing a prerequisite.

In the following, we have classified the prerequisites as either being required, recommended for all users, or optional depending on use cases.

Note: "Is there a maintained list of all prerequisites?" We use GitLab.com's continuous integration system for testing; the configuration YAML files for these jobs can be found in the directory `utils/tests/`. So, for instance, if you want to look at what packages the CI is installing for Alpine Linux, look at `utils/tests/gitlab-ci-alpine.yml`. The default CI (Arch Linux) `pacman` commands are in `utils/tests/gitlab-ci.yml`.

3.1 Requirements

3.1.1 Minimal requirements for release 3.36 and later

A C++ compiler (`g++` or `clang++`), Python 3, the CMake build system, and a separate C++ building tool such as `make`, `ninja-build`, or `Xcode` are the minimal requirements for compiling the software.

The `tar` and `bunzip2` utilities are needed to unpack source file archives. If you want to instead use Git to fetch code, rather than downloading a source archive, then `git` is required instead.

3.1.2 Minimal requirements for release 3.30-3.35

If you are not using Python bindings, since the Waf build system is provided as part of *ns-3*, there are only two build requirements (a C++ compiler, and Python3) for a minimal install of these older *ns-3* releases.

The `tar` and `bunzip2` utilities are needed to unpack source file archives. If you want to instead use Git to fetch code, rather than downloading a source archive, then `git` is required instead.

3.1.3 Minimal requirements for release 3.29 and earlier

Similarly, only a C++ compiler and Python2 were strictly required for building the *ns-3* releases 3.29 and earlier.

The `tar` and `bunzip2` utilities are needed to unpack source file archives. If you want to instead use Git to fetch code, rather than downloading a source archive, then `git` is required instead.

3.2 Recommended

The following are recommended for most users of *ns-3*.

3.2.1 compiler cache optimization (for ns-3.37 and later)

Ccache is a compiler cache optimization that will speed up builds across multiple *ns-3* directories, at the cost of up to an extra 5 GB of disk space used in the cache.

3.2.2 Code linting

Since ns-3.37 release, Clang-Format and Clang-Tidy are used to enforce the coding-style adopted by *ns-3*.

Users can invoke these tools directly from the command-line or through the (`utils/check-style-clang-format.py`) check program. Moreover, clang-tidy is integrated with CMake, enabling code scanning during the build phase.

Note: clang-format-14 through clang-format-16 version is required.

---

clang-format is strongly recommended to write code that follows the ns-3 code conventions, but might be skipped for simpler tasks (e.g., writing a simple simulation script for yourself).

clang-tidy is recommended when writing a module, to both follow code conventions and to provide hints on possible bugs in code.

Both are used in the CI system, and a merge request will likely fail if you did not use them.

3.2.3 Debugging

GDB and Valgrind are useful for debugging and recommended if you are doing C++ development of new models or scenarios. Both of the above tools are available for Linux and BSD systems; for macOS, LLDB is similar to GDB, but Valgrind doesn't appear to be available for M1 machines.

3.3 Optional

The remaining prerequisites listed below are only needed for optional ns-3 components.

Note: As of ns-3.30 release (August 2019), ns-3 uses Python 3 by default, but earlier releases depend on Python 2 packages, and at least a Python 2 interpreter is recommended. If installing the below prerequisites for an earlier release, one may in general substitute 'python' where 'python3' is found in the below (e.g., install 'python-dev' instead of 'python3-dev').

3.3.1 To read pcap packet traces

Many *ns-3* examples generate pcap files that can be viewed by pcap analyzers such as Tcpdump and Wireshark.

3.3.2 Database support

SQLite is recommended if you are using the statistics framework or if you are running LTE or NR simulations (which make use of SQLite databases):

3.3.3 Python bindings (ns-3.37 and newer)

ns-3 Python support now uses cppyy. Version 3.1.2 is the most recent supported cppyy release since ns-3.42.

Cppyy version 2.4.2 should be used from ns-3.37 up to 3.41.

Due to an upstream limitation with cppyy, Python bindings do not work on macOS machines with Apple silicon (M1 and M2 processors).

3.3.4 Using Python bindings (release 3.30 to ns-3.36)

This pertains to the use of existing Python bindings shipped with ns-3; for updating or generating new bindings, see further below.

Python bindings used pybindgen in the past, which can usually be found in the `ns-3-allinone` directory. Python3 development packages, and setup tools, are typically needed.

3.3.5 NetAnim animator

The Qt5 or Qt6 development tools are needed for NetAnim animator.

3.3.6 PyViz visualizer

The PyViz visualizer uses a variety of Python packages supporting GraphViz. In general, to enable Python support in ns-3, cppyy is required.

3.3.7 MPI-based distributed simulation

Open MPI support is needed if you intend to run large, parallel *ns-3* simulations.

3.3.8 Doxygen

Doxygen is only needed if you intend to write new Doxygen documentation for header files.

3.3.9 Sphinx documentation

The ns-3 manual (`doc/manual`), tutorial (`doc/tutorial`) and others are written in reStructuredText for Sphinx, and figures are typically in dia. To build PDF versions, texlive packages are needed.

3.3.10 Eigen3 support

Eigen3 is used to support more efficient calculations when using the 3GPP propagation loss models in LTE and NR simulations.

3.3.11 GNU Scientific Library (GSL)

GNU Scientific Library (GSL) is is only used for more accurate 802.11b (legacy) WiFi error models (not needed for more modern OFDM-based Wi-Fi).

3.3.12 XML-based version of the config store

Libxml2 is needed for the XML-based Config Store feature.

3.3.13 A GTK-based configuration system

GTK development libraries are also related to the (optional) config store, for graphical desktop support.

3.3.14 Generating modified python bindings (ns-3.36 and earlier)

To modify the Python bindings found in release 3.36 and earlier (not needed for modern releases, or if you do not use Python, the LLVM toolchain and cxxfilt are needed.

You will also need to install castxml and pygccxml as per the instructions for Python bindings (or through the bake build tool as described in the *ns-3* tutorial). If you plan to work with bindings or rescan them for any ns-3 C++ changes you might make, please read the chapter in the manual (corresponding to the release) on this topic.

3.3.15 To experiment with virtual machines and ns-3

Linux systems can use LXC and TUN/TAP device drivers for emulation support.

3.3.16 Support for openflow module

OpenFlow switch support requires XML and Boost development libraries.

CHAPTER

FOUR

LINUX

This chapter describes Linux-specific installation commands to install the options described in the previous chapter. The chapter is written initially with Ubuntu (Debian-based) Linux examples (Ubuntu is the most frequently used Linux distribution by *ns-3* users) but should translate fairly well to derivatives (e.g. Linux Mint).

The list of packages depends on which version of ns-3 you are trying to build, and on which extensions you need; please review the previous chapter if you need more information.

4.1 Requirements

The minimum supported version of Ubuntu is Ubuntu 18.04 LTS (as long as a modern compiler version such as g++ version 9 or later is added).

ns-3 Version	apt Packages
3.36 and later	`g++ python3 cmake ninja-build git`
3.30-3.35	`g++ python3 git`
3.29 and earlier	`g++ python2`

Note: As of July 2023 (ns-3.39 release and later), the minimum g++ version is g++-9. Older ns-3 releases may work with older versions of g++; check the RELEASE_NOTES.

4.2 Recommended

Feature	apt Packages
Compiler cache optimization	`ccache`
Code linting	`clang-format clang-tidy`
Debugging	`gdb valgrind`

Note: For Ubuntu 20.04 release and earlier, the version of ccache provided by apt (3.7.7 or earlier) may not provide performance benefits, and users are recommended to install version 4 or later, possibly as a source install. For Ubuntu 22.04 and later, ccache can be installed using apt.

Note: clang-format-14 through clang-format-16 version is required.

4.3 Optional

Please see below subsections for Python-related package requirements.

Feature	apt Packages
Reading pcap traces	`tcpdump wireshark`
Database support	`sqlite sqlite3 libsqlite3-dev`
NetAnim animator	`qtbase5-dev qtchooser qt5-qmake qtbase5-dev-tools`
MPI-based distributed simulation	`openmpi-bin openmpi-common openmpi-doc libopenmpi-dev`
Building Doxygen	`doxygen graphviz imagemagick`
Sphinx documentation	`python3-sphinx dia imagemagick texlive dvipng latexmk texlive-extra-utils texlive-latex-extra texlive-font-utils`
Eigen3	`libeigen3-dev`
GNU Scientific Library	`gsl-bin libgsl-dev libgslcblas0`
XML config store	`libxml2 libxml2-dev`
GTK-based config store	`libgtk-3-dev`
Emulation with virtual machines and tap bridge	`lxc-utils lxc-templates iproute2 iptables`
Support for openflow	`libxml2 libxml2-dev libboost-all-dev`

Note: For Ubuntu 20.10 and earlier, the single 'qt5-default' package suffices for NetAnim (`apt install qt5-default`)

4.3.1 Python bindings

Python requires *Cppyy, <https://cppyy.readthedocs.io/en/latest/installation.html>* and specifically, version 3.1.2 is the latest version known to work with ns-3 at this time.

ns-3.42 and newer:

```
python3 -m pip install --user cppyy==3.1.2
```

ns-3.37-3.41:

```
python3 -m pip install --user cppyy==2.4.2
```

ns-3.30-3.36 (also requires pybindgen, found in the `allinone` directory):

```
apt install python3-dev pkg-config python3-setuptools
```

4.3.2 PyViz visualizer

The PyViz visualizer uses a variety of Python packages supporting GraphViz.:

```
apt install gir1.2-goocanvas-2.0 python3-gi python3-gi-cairo python3-pygraphviz gir1.
↪2-gtk-3.0 ipython3
```

For Ubuntu 18.04 and later, python-pygoocanvas is no longer provided. The ns-3.29 release and later upgrades the support to GTK+ version 3, and requires these packages:

For ns-3.28 and earlier releases, PyViz is based on Python2, GTK+ 2, GooCanvas, and GraphViz:

```
apt install python-pygraphviz python-kiwi python-pygoocanvas libgoocanvas-dev ipython
```

4.3.3 Generating modified python bindings (ns-3.36 and earlier)

To modify the Python bindings found in release 3.36 and earlier (not needed for modern releases, or if you do not use Python).:

```
apt install cmake libc6-dev libc6-dev-i386 libclang-dev llvm-dev automake python3-pip
python3 -m pip install --user cxxfilt
```

and you will want to install castxml and pygccxml as per the instructions for python bindings (or through the bake build tool as described in the tutorial). The 'castxml' and 'pygccxml' packages provided by Ubuntu 18.04 and earlier are not recommended; a source build (coordinated via bake) is recommended. If you plan to work with bindings or rescan them for any ns-3 C++ changes you might make, please read the chapter in the manual on this topic.

4.4 Caveats and troubleshooting

When building documentation, if you get an error such as `convert ... not authorized source-temp/figures/lena-dual-stripe.eps`, see this post about editing ImageMagick's security policy configuration. In brief, you will want to make this kind of change to ImageMagick security policy:

```
--- ImageMagick-6/policy.xml.bak 2020-04-28 21:10:08.564613444 -0700
+++ ImageMagick-6/policy.xml 2020-04-28 21:10:29.413438798 -0700
@@ -87,10 +87,10 @@
   <policy domain="path" rights="none" pattern="@*"/>
-  <policy domain="coder" rights="none" pattern="PS" />
+  <policy domain="coder" rights="read|write" pattern="PS" />
   <policy domain="coder" rights="none" pattern="PS2" />
   <policy domain="coder" rights="none" pattern="PS3" />
   <policy domain="coder" rights="none" pattern="EPS" />
-  <policy domain="coder" rights="none" pattern="PDF" />
+  <policy domain="coder" rights="read|write" pattern="PDF" />
   <policy domain="coder" rights="none" pattern="XPS" />
 </policymap>
```

CHAPTER

FIVE

MACOS

This chapter describes installation steps specific to Apple macOS. macOS installation of *ns-3* requires either the installation of the full Xcode IDE or a more minimal install of Xcode Command Line Tools).

The full Xcode IDE requires 40 GB of disk space. If you are just interested in getting *ns-3* to run, the full Xcode is not necessary; we recommend Command Line Tools instead.

In addition to Command Line Tools, some *ns-3* extensions require third-party libraries; we recommend either Homebrew or MacPorts. If you prefer, you can probably avoid installing Command Line Tools and install the compiler of your choice and any other tools you may need using Homebrew or MacPorts.

In general, documentation on the web suggests to use either, but not both, Homebrew or MacPorts on a particular system. It has been noted that Homebrew tends to install the GUI version of certain applications without easily supporting the command-line equivalent, such as for the dia application; see ns-3 MR 1247 for discussion about this.

Finally, regarding Python, some *ns-3* maintainers prefer to use a virtualenv to guard against incompatibilities that might arise from the native macOS Python and versions that may be installed by Homebrew or Anaconda. Some *ns-3* users never use Python bindings or visualizer, but if your *ns-3* workflow requires more heavy use of Python, please keep the possibility of a virtualenv in mind if you run into Python difficulties. For a short guide on virtual environments, please see this link.

Due to an upstream limitation with Cppyy, Python bindings do not work on macOS machines with Apple silicon (M1 and M2 processors).

5.1 Requirements

Installing ns-3 on macOS requires two fundamental things: 1) C++/Python development tools, and 2) CMake build system with at least one underlying build tool. These can either be installed via binary package installation from the macOS App Store (Xcode development tools) or the web (CMake binary package), or from Homebrew or MacPorts.

macOS Xcode uses the Clang/LLVM compiler toolchain. It is possible to install the GNU compiler gcc/g++ from Homebrew and MacPorts, but macOS will not provide it due to licensing issues. If you do not install Xcode you will have to install build tools via Homebrew or MacPorts. *ns-3* works on recent versions of both `clang++` and `g++`, so for macOS, there is no need to install `g++`.

The following table provides package names for installing CMake and Ninja build system from Homebrew or MacPorts.

ns-3 Version	Homebrew packages	MacPorts packages
3.36 and later	`cmake ninja`	`cmake ninja`
3.35 and earlier	None	None

You will know you are done when you can successfully type *clang++ -v* at the command line, and when you type *cmake –help* and it identifies that you have at least one installed generator (in the below example, Unix Makefiles):

```
Generators
The following generators are available on this platform (* marks default):
* Unix Makefiles               = Generates standard UNIX makefiles.
  Ninja                        = Generates build.ninja files.
  Ninja Multi-Config           = Generates build-<Config>.ninja files.
  Watcom WMake                 = Generates Watcom WMake makefiles.
  Xcode                        = Generate Xcode project files.
```

5.2 Recommended

Feature	Homebrew packages	MacPorts packages
Compiler cache optimization	`ccache`	`ccache`
Code linting	clang-format included with `llvm`, need to select `llvm@XX` version	clang-format included with `clang`, need to select `clang-XX llvm-XX` versions

Note: macOS development tools are based on clang, so installing llvm and clang using Homebrew or MacPorts is typically unnecessary. However, `clang-tidy` might be missing, and the `clang-format` version might be not the expected one. In these cases it is suggested to install the `llvm` package either though Homebrew or MacPorts.

Note: Homebrew provides a `clang-format` package, but its version might be incompatible with the one used by *ns-3*.

The `llvm` Homebrew package provides `clang-tidy` and `clang-format`, but the binary is placed at `/opt/homebrew/opt/llvm@XX/bin/clang-tidy` (where `XX` is the installed version number such as `18`), so you will need to add this path to your `$PATH` variable.

Note: Likewise, when using MacPorts, the `clang-tidy` and `clang-format` binaries will be placed in `/opt/local/libexec/llvm-XX/bin` (where `XX` is the installed version number such as `18`), so you will need to add this to your `$PATH` variable.

Note: For debugging, `lldb` is the default debugger for llvm. Memory checkers such as Memory Graph exist for macOS, but the ns-3 team doesn't have experience with it as a substitution for `valgrind` (which is reported to not work on M1 Macs).

5.3 Optional

Please see below subsections for Python-related package requirements.

For MacPorts packages we show the most recent package version available as of early 2023.

Feature	Homebrew packages	MacPort packages
Reading pcap traces	`wireshark`	`wireshark4`
Database support	`sqlite`	`sqlite3`
NetAnim animator	`qt@5`	`qt513`
MPI-based distributed simulation	`open-mpi`	`openmpi`
Building Doxygen	`doxygen graphviz imagemagick`	`doxygen graphviz ImageMagick`
Sphinx documentation	`sphinx-doc texlive`	`dia texlive texlive-fonts-extra texlive-latex-extra py3XX-sphinx`, with *XX* ` the Python minor version such as *12*
Eigen3	`eigen`	`eigen3`
GNU Scientific Library	`gsl`	`gsl`
XML config store	`libxml2`	`libxml2`
GTK-based config store	`gtk+3`	`gtk3` or `gtk4`
Emulation with virtual machines	Not available for macOS	Not available for macOS
Support for openflow, CircularApertureAntennaModel	`boost`	`boost`

5.4 Caveats and troubleshooting

CHAPTER

SIX

WINDOWS

This chapter describes installation steps specific to Microsoft Windows (version 10) and its derivatives (e.g. Home, Pro, Enterprise) using the Msys2/MinGW64 toolchain.

There are two documented ways to use *ns-3* on Windows: the Windows Subsystem for Linux (WSL) or via the Msys2/MinGW64 toolchain. Both options are listed below; users may choose to install either WSL or the Msys2/MinGW64 toolchain (installing both is not required).

Note: *ns-3* is not fully compatible with Visual Studio IDE / MSVC compiler; only Visual Studio Code editor, the Msys2/MinGW64 toolchain, and WSL, as explained below.

6.1 Windows Subsystem for Linux

Windows Subsystem for Linux (WSL), particularly WSL2, can be used on Windows for *ns-3*. WSL2 runs a real Linux kernel on Windows's Hyper-V hypervisor, providing 100% code compatibility with Linux and seamless integration with Windows. VS Code has excellent support and integration with WSL, enabling Windows users to develop for ns-3 in a native environment. It is recommended to install the WSL extension in VS Code for this purpose.

Users starting with WSL2 can follow the Linux installation instructions to fill out other package prerequisites. Note that *ns-3* emulation features using WSL2 are not tested/supported.

For more information:

- WSL
- Developing in WSL with VS Code
- WSL tutorial

6.2 Windows 10 package prerequisites

The following instructions are relevant to the ns-3.37 release and Windows 10.

6.2.1 Installation of the Msys2 environment

The Msys2 includes ports of Unix tools for Windows built with multiple toolchains, including: MinGW32, MinGW64, Clang64, UCRT.

ns-3 has been tested with the MinGW64 (GCC) toolchain. MinGW32 is 32-bit, which *ns-3* does not support. The project's Windows maintainer has tested Clang64 unsuccessfully, and has not tested the UCRT toolchain (which may work).

The Msys2 installer can be found on their site. Msys2 will be installed by default in the `C:\msys64` directory.

The next required step is adding the binaries directories from the MinGW64 toolchain and generic Msys2 tools to the PATH environment variable. This can be accomplished via the GUI (search for system environment variable), or via the following command (assuming it was installed to the default directory):

```
C:\> setx PATH "C:\msys64\mingw64\bin;C:\msys64\usr\bin;%PATH%;" /m
```

Note: if the MinGW64 binary directory doesn't precede the Windows/System32 directory (already in `%PATH%`), the documentation build will fail since Windows has a conflicting `convert` command (FAT-to-NTFS). Similarly, if the Msys64 binary directory doesn't precede the Windows/System2 directory, running the `bash` command will result in Windows trying to run the Windows Subsystem for Linux (WSL) bash shell.

6.2.2 Accessing the MinGW64 shell

After installing Msys2 and adding the binary directories to the `PATH`, we can access the Unix-like MinGW64 shell and use the Pacman package manager.

The Pacman package manager is similar to the one used by Arch Linux, and can be accessed via one of the Msys2 shells. In this case, we will be using the MinGW64 shell. We can take this opportunity to update the package cache and packages.

```
C:\ns-3-dev\> set MSYSTEM MINGW64
C:\ns-3-dev\> bash
/c/ns-3-dev/ MINGW64$ pacman -Syu
```

Pacman will request you to close the shell and re-open it to proceed after the upgrade.

6.2.3 Minimal requirements for C++ (release)

This is the minimal set of packages needed to run ns-3 C++ programs from a released tarball.

```
/c/ns-3-dev/ MINGW64$ pacman -S \
mingw-w64-x86_64-toolchain \
mingw-w64-x86_64-cmake \
mingw-w64-x86_64-ninja \
mingw-w64-x86_64-grep \
mingw-w64-x86_64-sed \
mingw-w64-x86_64-python
```

6.2.4 Netanim animator

Qt5 or Qt6 development tools are needed for Netanim animator.

```
/c/ns-3-dev/ MINGW64$ pacman -S mingw-w64-x86_64-qt5 git
```

6.2.5 Support for MPI-based distributed emulation

The MPI setup requires two parts.

The first part is the Microsoft MPI SDK required to build the MPI applications, which is distributed via Msys2.

```
/c/ns-3-dev/ MINGW64$ pacman -S mingw-w64-x86_64-msmpi
```

The second part is the Microsoft MPI executors (mpiexec, mpirun) package, which is distributed as an installable (`msmpisetup.exe`).

After installing it, the path containing the executors also need to be included to `PATH` environment variable of the Windows and/or the MinGW64 shell, depending on whether you want to run MPI programs in either shell or both of them.

```
C:\ns-3-dev\> setx PATH "%PATH%;C:\Program Files\Microsoft MPI\Bin" /m
C:\ns-3-dev\> set MSYSTEM MINGW64
C:\ns-3-dev\> bash
/c/ns-3-dev/ MINGW64$ echo "export PATH=$PATH:/c/Program\ Files/Microsoft\ MPI/Bin" >>
↪ ~/.bashrc
```

6.2.6 Debugging

GDB is installed along with the mingw-w64-x86_64-toolchain package.

6.2.7 Doxygen and related inline documentation

To build the Doxygen-based documentation, we need doxygen and a Latex toolchain. Getting Doxygen from Msys2 is straightforward.

```
/c/ns-3-dev/ MINGW64$ pacman -S mingw-w64-x86_64-imagemagick mingw-w64-x86_64-doxygen␣
↪mingw-w64-x86_64-graphviz mingw-w64-x86_64-texlive-full
```

6.2.8 The ns-3 manual, models and tutorial

These documents are written in reStructuredText for Sphinx (doc/tutorial, doc/manual, doc/models). The figures are typically written in Dia.

The documents can be generated into multiple formats, one of them being pdf, which requires the same Latex setup for doxygen.

Sphinx and Dia can be installed via Msys2's package manager:

```
/c/ns-3-dev/ MINGW64$ pacman -S mingw-w64-x86_64-python-sphinx mingw-w64-x86_64-dia␣
↪mingw-w64-x86_64-texlive-full
```

6.2.9 GNU Scientific Library (GSL)

GSL is used to provide more accurate WiFi error models and can be installed with:

```
/c/ns-3-dev/ MINGW64$ pacman -S mingw-w64-x86_64-gsl
```

6.2.10 Database support for statistics framework

SQLite3 is installed along with the mingw-w64-x86_64-toolchain package.

6.2.11 Xml-based version of the config store

Requires libxml2 >= version 2.7, which can be installed with the following.

```
/c/ns-3-dev/ MINGW64$ pacman -S mingw-w64-x86_64-libxml2
```

6.2.12 Support for openflow module

Requires some boost libraries that can be installed with the following.

```
/c/ns-3-dev/ MINGW64$ pacman -S mingw-w64-x86_64-boost
```

6.2.13 Support for click module

Requires autotools, which can be installed with the following.

```
/c/ns-3-dev/ MINGW64$ pacman -S mingw-w64-x86_64-autotools
```

6.3 Windows 10 Docker container

Docker containers are not as useful for Windows, since only Windows hosts can use them, however we add directions on how to use the Windows container and how to update the Docker image for reference.

First, gather all dependencies previously mentioned to cover all supported features. Install them to a base directory for the container (e.g. `C:\tools`).

Save the following Dockerfile to the base directory.

```
# It is really unfortunate we need a 16 GB base image just to get the installers␣
↪working, but such is life
FROM mcr.microsoft.com/windows:20H2

# Copy the current host directory to the container
COPY .\\ C:/tools
WORKDIR C:\\tools

# Create temporary dir
RUN mkdir C:\\tools\\temp
```

(continues on next page)

(continued from previous page)

```
# Export environment variables
RUN setx PATH "C:\\tools\\msys64\\mingw64\\bin;C:\\tools\\msys64\\usr\\bin;%PATH%" /m
RUN setx PATH "%PATH%;C:\\Program Files\\Microsoft MPI\\bin;C:\\tools\\dia\bin;C:\
↪tools\texlive\2022\bin\win32" /m
RUN setx MSYSTEM "MINGW64" /m

# Install Msys2
RUN .\\msys2-x86_64-20220503.exe in --confirm-command --accept-messages --root C:\\
↪tools\\msys64

# Update base packages
RUN C:\\tools\\msys64\\usr\\bin\\pacman -Syyuu --noconfirm

# Install base packages
RUN bash -c "echo export PATH=$PATH:/c/Program\ Files/Microsoft\ MPI/Bin >> /c/tools/
↪msys64/home/$USER/.bashrc" && \
    bash -c "echo export PATH=$PATH:/c/tools/dia/bin >> /c/tools/msys64/home/$USER/.
↪bashrc" && \
    bash -c "echo export PATH=$PATH:/c/tools/texlive/2022/bin/win32 >> /c/tools/
↪msys64/home/$USER/.bashrc" && \
    bash -c "pacman -S mingw-w64-x86_64-toolchain \
                    mingw-w64-x86_64-cmake \
                    mingw-w64-x86_64-ninja \
                    mingw-w64-x86_64-grep \
                    mingw-w64-x86_64-sed \
                    mingw-w64-x86_64-qt5 \
                    git \
                    mingw-w64-x86_64-msmpi \
                    mingw-w64-x86_64-imagemagick \
                    mingw-w64-x86_64-doxygen \
                    mingw-w64-x86_64-graphviz \
                    mingw-w64-x86_64-python-sphinx \
                    mingw-w64-x86_64-gsl \
                    mingw-w64-x86_64-libxml2 \
                    mingw-w64-x86_64-boost \
                    --noconfirm"

# Install Microsoft MPI
RUN .\\msmpisetup.exe -unattend -force -full -verbose

# Install TexLive
RUN .\\install-tl-20220526\\install-tl-windows.bat --no-gui --lang en -profile .\\
↪texlive.profile

# Move working directory to temp and start cmd
WORKDIR C:\\tools\\temp
ENTRYPOINT ["cmd"]
```

Now you should be able to run `docker build -t username/image .`.

After building the container image, you should be able to use it:

```
$ docker run -it username/image
C:\tools\temp$ git clone https://gitlab.com/nsnam/ns-3-dev
C:\tools\temp$ cd ns-3-dev
C:\tools\temp\ns-3-dev$ python ns3 configure --enable-tests --enable-examples
C:\tools\temp\ns-3-dev$ python ns3 build
```

(continues on next page)

(continued from previous page)

```
C:\tools\temp\ns-3-dev$ python test.py
```

If testing succeeds, the container image can then be pushed to the Docker Hub using `docker push username/image`.

6.4 Windows 10 Vagrant

As an alternative to manually setting up all dependencies required by ns-3, one can use a pre-packaged virtual machine. There are many ways to do that, but for automation, the most used certainly is Vagrant.

Vagrant supports multiple virtual machine providers, is available in all platforms and is fairly straightforward to use and configure.

There are many boxes available offering guests operating systems such as BSD, Mac, Linux and Windows.

6.4.1 Using the pre-packaged Vagrant box

The provider for the ns-3 Vagrant box is VirtualBox.

The reference Windows virtual machine can be downloaded via the following Vagrant command

```
~/mingw64_test $ vagrant init gabrielcarvfer/ns3_win10_mingw64
```

After that, a Vagrantfile will be created in the current directory (in this case, mingw64_test).

The file can be modified to adjust the number of processors and memory available to the virtual machine (VM).

```
~/mingw64_test $ cat Vagrantfile
# -*- mode: ruby -*-
# vi: set ft=ruby :

# All Vagrant configuration is done below. The "2" in Vagrant.configure
# configures the configuration version (we support older styles for
# backwards compatibility). Please don't change it unless you know what
# you're doing.
Vagrant.configure("2") do |config|
# The most common configuration options are documented and commented below.
# For a complete reference, please see the online documentation at
# https://docs.vagrantup.com.

# Every Vagrant development environment requires a box. You can search for
# boxes at https://vagrantcloud.com/search.
config.vm.box = "gabrielcarvfer/ns3_win10_mingw64"

# Disable automatic box update checking. If you disable this, then
# boxes will only be checked for updates when the user runs
# `vagrant box outdated`. This is not recommended.
# config.vm.box_check_update = false

# Create a forwarded port mapping which allows access to a specific port
# within the machine from a port on the host machine. In the example below,
# accessing "localhost:8080" will access port 80 on the guest machine.
# NOTE: This will enable public access to the opened port
# config.vm.network "forwarded_port", guest: 80, host: 8080
```

(continues on next page)

(continued from previous page)

```
# Create a forwarded port mapping which allows access to a specific port
# within the machine from a port on the host machine and only allow access
# via 127.0.0.1 to disable public access
# config.vm.network "forwarded_port", guest: 80, host: 8080, host_ip: "127.0.0.1"

# Create a private network, which allows host-only access to the machine
# using a specific IP.
# config.vm.network "private_network", ip: "192.168.33.10"

# Create a public network, which generally matched to bridged network.
# Bridged networks make the machine appear as another physical device on
# your network.
# config.vm.network "public_network"

# Share an additional folder to the guest VM. The first argument is
# the path on the host to the actual folder. The second argument is
# the path on the guest to mount the folder. And the optional third
# argument is a set of non-required options.
# config.vm.synced_folder "../data", "/vagrant_data"

# Provider-specific configuration so you can fine-tune various
# backing providers for Vagrant. These expose provider-specific options.
# Example for VirtualBox:
#
# config.vm.provider "virtualbox" do |vb|
#   # Display the VirtualBox GUI when booting the machine
#   vb.gui = true
#
#   # Customize the amount of memory on the VM:
#   vb.memory = "1024"
# end
#
# View the documentation for the provider you are using for more
# information on available options.

# Enable provisioning with a shell script. Additional provisioners such as
# Ansible, Chef, Docker, Puppet and Salt are also available. Please see the
# documentation for more information about their specific syntax and use.
# config.vm.provision "shell", inline: <<-SHELL
#   apt-get update
#   apt-get install -y apache2
# SHELL
end
```

We can uncomment the virtualbox provider block and change vCPUs and RAM. It is recommended never to match the number of vCPUs to the number of thread of the machine, or the host operating system can become unresponsive. For compilation workloads, it is recommended to allocate 1-2 GB of RAM per vCPU.

```
~/mingw64_test/ $ cat Vagrantfile
# -*- mode: ruby -*-
# vi: set ft=ruby :
Vagrant.configure("2") do |config|
config.vm.box = "gabrielcarvfer/ns3_win10_mingw64"
  config.vm.provider "virtualbox" do |vb|
    vb.cpus = "8"
```

(continues on next page)

(continued from previous page)

```
    vb.memory = "8096" # 8GB of RAM
  end
end
```

After changing the settings, we can start the VM and login via ssh. The default password is "vagrant".

```
~/mingw64_test/ $ vagrant up
~/mingw64_test/ $ vagrant ssh
C:\Users\vagrant>
```

We are now logged into the machine and ready to work. If you prefer to update the tools, get into the MinGW64 shell and run pacman.

```
C:\Users\vagrant\> set MSYSTEM MINGW64
C:\Users\vagrant\> bash
/c/Users/vagrant/ MINGW64$ pacman -Syu
/c/Users/vagrant/ MINGW64$ exit
C:\Users\vagrant\>
```

At this point, we can clone ns-3 locally:

```
C:\Users\vagrant> git clone `https://gitlab.com/nsnam/ns-3-dev`
C:\Users\vagrant> cd ns-3-dev
C:\Users\vagrant\ns-3-dev> python3 ns3 configure --enable-tests --enable-examples --
↪enable-mpi
C:\Users\vagrant\ns-3-dev> python3 test.py
```

We can also access the ~/mingw64_test/ from the host machine, where the Vagrantfile resides, by accessing the synchronized folder C:vagrant. If the Vagrantfile is in the host ns-3-dev directory, we can continue working on it.

```
C:\Users\vagrant> cd C:\vagrant
C:\vagrant\> python3 ns3 configure --enable-tests --enable-examples --enable-mpi
C:\vagrant\> python3 test.py
```

If all the PATH variables were set for the MinGW64 shell, we can also use it instead of the default CMD shell.

```
C:\vagrant\> set MSYSTEM=MINGW64
C:\vagrant\> bash
/c/vagrant/ MINGW64$ ./ns3 clean
/c/vagrant/ MINGW64$ ./ns3 configure --enable-tests --enable-examples --enable-mpi
/c/vagrant/ MINGW64$ ./test.py
```

To stop the Vagrant machine, we should close the SSH session then halt.

```
/c/vagrant/ MINGW64$ exit
C:\vagrant\> exit
~/mingw64_test/ vagrant halt
```

To destroy the machine (e.g. to restore the default settings), use the following.

```
vagrant destroy
```

6.4.2 Packaging a new Vagrant box

BEWARE: DO NOT CHANGE THE SETTINGS MENTIONED ON A REAL MACHINE

THE SETTINGS ARE MEANT FOR A DISPOSABLE VIRTUAL MACHINE

Start by downloading the Windows 10 ISO.

Then install VirtualBox.

Configure a VirtualBox VM and use the Windows 10 ISO file as the install source.

During the installation, create a local user named "vagrant" and set its password to "vagrant".

Check for any Windows updates and install them.

The following commands assume administrative permissions and a PowerShell shell.

Install the VirtualBox guest extensions

On the VirtualBox GUI, click on `Devices->Insert Guest Additions CD Image...` to download the VirtualBox guest extensions ISO and mount it as a CD drive on the guest VM.

Run the installer to enable USB-passthrough, folder syncing and others.

After installing, unmount the drive by removing it from the VM. Click on `Settings->Storage`, select the guest drive and remove it clicking the button with an red `x`.

Install the OpenSSH server

Open PowerShell and run the following to install OpenSSH server, then set it to start automatically and open the firewall ports.

```
Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0
Add-WindowsCapability -Online -Name OpenSSH.Server~~~~0.0.1.0
Start-Service sshd
Set-Service -Name sshd -StartupType 'Automatic'
if (!(Get-NetFirewallRule -Name "OpenSSH-Server-In-TCP" -ErrorAction SilentlyContinue␣
↪| Select-Object Name, Enabled)) {
    Write-Output "Firewall Rule 'OpenSSH-Server-In-TCP' does not exist, creating it...
↪"
    New-NetFirewallRule -Name 'OpenSSH-Server-In-TCP' -DisplayName 'OpenSSH Server␣
↪(sshd)' -Enabled True -Direction Inbound -Protocol TCP -Action Allow -LocalPort 22
} else {
    Write-Output "Firewall rule 'OpenSSH-Server-In-TCP' has been created and exists."
}
```

Enable essential services and disable unnecessary ones

Ensure the following services are set to **automatic** from the Services panel(services.msc):

- Base Filtering Engine
- Remote Procedure Call (RPC)
- DCOM Server Process Launcher
- RPC Endpoint Mapper
- Windows Firewall

Ensure the following services are set to **disabled** from the Services panel(services.msc):

- Windows Update
- Windows Update Remediation
- Windows Search

The same can be accomplished via the command-line with the following commands:

```
Set-Service -Name BFE -StartupType 'Automatic'
Set-Service -Name RpcSs -StartupType 'Automatic'
Set-Service -Name DcomLaunch -StartupType 'Automatic'
Set-Service -Name RpcEptMapper -StartupType 'Automatic'
Set-Service -Name mpssvc -StartupType 'Automatic'
Set-Service -Name wuauserv -StartupType 'Disabled'
Set-Service -Name WaaSMedicSvc -StartupType 'Disabled'
Set-Service -Name WSearch -StartupType 'Disabled'
```

Install the packages you need

In this step we install all the software required by ns-3, as listed in the Section *Windows 10 package prerequisites*.

Disable Windows Defender

After installing everything, it should be safe to disable the Windows security.

Enter in the Windows Security settings and disable "anti-tamper protection". It rollbacks changes to security settings periodically.

Enter in the Group Policy Editor (gpedit.msc) and disable:

- Realtime protection
- Behavior monitoring
- Scanning of archives, removable drives, network files, scripts
- Windows defender

The same can be accomplished with the following command-line commands.

```
Set-MpPreference -DisableArchiveScanning 1 -ErrorAction SilentlyContinue
Set-MpPreference -DisableBehaviorMonitoring 1 -ErrorAction SilentlyContinue
Set-MpPreference -DisableIntrusionPreventionSystem 1 -ErrorAction SilentlyContinue
Set-MpPreference -DisableIOAVProtection 1 -ErrorAction SilentlyContinue
Set-MpPreference -DisableRemovableDriveScanning 1 -ErrorAction SilentlyContinue
Set-MpPreference -DisableBlockAtFirstSeen 1 -ErrorAction SilentlyContinue
Set-MpPreference -DisableScanningMappedNetworkDrivesForFullScan 1 -ErrorAction␣
↪SilentlyContinue
Set-MpPreference -DisableScanningNetworkFiles 1 -ErrorAction SilentlyContinue
Set-MpPreference -DisableScriptScanning 1 -ErrorAction SilentlyContinue
Set-MpPreference -DisableRealtimeMonitoring 1 -ErrorAction SilentlyContinue
Set-Service -Name WdNisSvc -StartupType 'Disabled'
Set-Service -Name WinDefend -StartupType 'Disabled'
Set-Service -Name Sense -StartupType 'Disabled'
Set-ItemProperty -Path "HKLM:\SOFTWARE\Microsoft\Windows Defender\Real-Time Protection
↪" -Name SpyNetReporting -Value 0
Set-ItemProperty -Path "HKLM:\SOFTWARE\Microsoft\Windows Defender\Real-Time Protection
↪" -Name SubmitSamplesConsent -Value 0
```

(continues on next page)

(continued from previous page)

```
Set-ItemProperty -Path "HKLM:\SOFTWARE\Microsoft\Windows Defender\Features" -Name␣
↪TamperProtection -Value 4
Set-ItemProperty -Path "HKLM:\SOFTWARE\Microsoft\Windows Defender" -Name␣
↪DisableAntiSpyware -Value 1
Set-ItemProperty -Path "HKLM:\SOFTWARE\Policies\Microsoft\Windows Defender" -Name␣
↪DisableAntiSpyware -Value 1
```

Note: the previous commands were an excerpt from the complete script in: https://github.com/jeremybeaume/tools/blob/master/disable-defender.ps1

Turn off UAC notifications

The UAC notifications are the popups where you can give your OK to elevation to administrative privileges. It can be disabled via User Account Control Settings, or via the following commands.

```
reg ADD HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System /v EnableLUA /
↪t REG_DWORD /d 0 /f
```

Change the strong password security Policy

Open the Local Security Policy management window. Under Security Settings/Account Policy/Password Policy, disable the option saying "Password must meet complexity requirements".

Testing

After you reach this point, reboot your machine then log back in.

Test if all your packages are working as expected.

In the case of ns-3, try to enable all supported features, run the test.py and test-ns3.py suites.

If everything works, then try to log in via SSH.

If everything works, shut down the machine and prepare for packaging.

The network interface configured should be a NAT. Other interfaces won't work correctly.

Default Vagrantfile

Vagrant can package VirtualBox VMs into Vagrant boxes without much more work. However, it still needs one more file to do that: the default Vagrantfile.

This file will be used by Vagrant to configure the VM later on and how to connect to it.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
config.vm.box = "BOX_FILE.box" # name of the box
config.vm.communicator = "winssh" # indicate that we are talking to a windows box via␣
↪ssh
config.vm.guest = :windows # indicate that the guest is a windows machine
config.vm.network :forwarded_port, guest: 3389, host: 3389, id: "rdp", auto_correct:␣
↪true
```

(continues on next page)

(continued from previous page)

```
config.ssh.password = "vagrant" # give the default password, so that it stops trying␣
↪to use a .ssh key-pair
config.ssh.insert_key = false # let the user use a written password
config.ssh.keys_only = false
config.winssh.shell = "cmd" # select the default shell (could be cmd or powershell)
  config.vm.provider :virtualbox do |v, override|
      #v.gui = true # do not show the VirtualBox GUI if unset or set to false
      v.customize ["modifyvm", :id, "--memory", 8096] # the default settings for the␣
↪VM are 8GB of RAM
      v.customize ["modifyvm", :id, "--cpus", 8] # the default settings for the VM␣
↪are 8 vCPUs
      v.customize ["modifyvm", :id, "--vram", 128] # 128 MB or vGPU RAM
      v.customize ["modifyvm", :id, "--clipboard", "bidirectional"]
      v.customize ["setextradata", "global", "GUI/SuppressMessages", "all" ]
  end
end
```

This Vagrantfile will be baked into the Vagrant box, and can be modified by the *user Vagrantfile*. After writing the Vagrantfile, we can call the following command.

```
vagrant package --vagrantfile Vagrantfile --base VIRTUALBOX_VM_NAME --output BOX_FILE.
↪box
```

It will take an awful long time depending on your drive.

After it finishes, we can add the box to test it.

```
vagrant box add BOX_NAME BOX_FILE
vagrant up BOX_NAME
vagrant ssh
```

If it can connect to the box, you are ready to upload it to the Vagrant servers.

Publishing the Vagrant box

Create an account in https://app.vagrantup.com/ or log in with yours. In the dashboard, you can create a new box named BOX_NAME or select an existing one to update.

After you select your box, click to add a provider. Pick Virtualbox. Calculate the MD5 hash of your BOX_FILE.box and fill the field then click to proceed.

Upload the box.

Now you should be able to download your box from the Vagrant servers via the the following command.

```
vagrant init yourUserName/BOX_NAME
vagrant up
vagrant ssh
```

More information on Windows packaging to Vagrant boxes is available here:

- https://www.vagrantup.com/docs/boxes/base
- https://www.vagrantup.com/docs/vagrantfile/machine_settings
- https://www.vagrantup.com/docs/vagrantfile/ssh_settings
- https://www.vagrantup.com/docs/vagrantfile/winssh_settings

- https://github.com/pghalliday/windows-vagrant-boxes

CHAPTER

SEVEN

INSTALLING BAKE

Bake is a build system orchestration tool that was primarily designed for installing ns-3 Direct Code Execution but can be used more generally to install ns-3 third-party libraries and apps.

Bake is a Python 3 program that requires the `distro` and `requests` packages; to start using Bake, it is generally sufficient to add those packages to your Python 3 installation using `pip`.

Bake is further documented elsewhere, including here and in the Getting Started chapter of the *ns-3* tutorial.