June 17-18, 2019

ns-3 training software

This document contains notes about the use of ns-3 in the training offered in Florence on June 17-18, 2019.

Working along with the instructors is encouraged but not required. We should have enough documentation to allow one to try the sample programs again later.

The minimal requirement is to have a working C++ compiler and Python installation on a Linux or macOS system. Many distributions have this by default.

- **Linux users:** 'g++', 'python', and 'git' must work on your system.
- **macOS users:** You must install Xcode from the App Store. This will provide you with the 'clang++' compiler and with Python and git.
- **Windows 10 users:** Windows (native operation using Visual Studio) is not a supported platform for ns-3. Some people use a Linux virtual machine when using Windows. Others have had success using the 'Bash Subsystem for Linux' for Windows 10, which provides an Ubuntu Linux-like environment.

To make fuller use of ns-3, many other optional packages can be installed. The Installation wiki provides detailed guides for several distributions and for macOS.

https://www.nsnam.org/wiki/Installation

Software location

ns-3 repositories are stored in GitLab.com repositories, under the project name 'nsnam'. The main repository is called 'ns-3-dev', and from this main tree, different releases are created.

Software for this training is stored in a special repository, which is a 'fork' of the main project repository.

To clone the repository, type:

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

However, ns-3 is also used with other software, some of which is maintained alongside ns-3. We provide a different repository called 'ns-3-allinone' for download and build scripts to tie together the main packages of interest. To obtain this software, instead type:

```
$ git clone https://gitlab.com/tomhenderson/ns-3-allinone.git
```

June 17-18, 2019

Quick start

If you are familiar with git, you can try this to obtain and build the training code:

```
$ git clone -b training https://gitlab.com/tomhenderson/ns-3-allinone.git
$ cd ns-3-allinone
$ ./download.py -b training
$ ./build.py -- --enable-examples --enable-tests
$ cd ns-3-dev
```

Note that the arguments passed to ./build.py (--enable-examples --enable-tests) are separated by two dash characters '--'. This is not a typo; this ensures that those last two options are passed on to the Waf build system.

After the above, you should have a working ns-3 installation (debug build) that is in sync with a very recent changeset of ns-3-dev. You can check it by running some of the tutorial programs:

```
$ ./waf --run first
```

To update the code at a future date, you can perform a 'git pull' while in ns-3-dev. After the initial download of ns-3-allinone (which fetches pybindgen and netanim also), this directory is no longer used and we will live in the ns-3-dev directory for the rest of training.

The remaining instructions go into some more detail into issues that are encapsulated in the quick start instructions above.

git branches

Please refer to a git tutorial online if you are unfamiliar with basic git. We will use basic git to manage the software.

One key concept is that of a 'branch' of code. We will use the training branch in each of the above repositories. To 'check out' a branch, type:

```
$ cd ns-3-allinone
$ git checkout -b training origin/training
$ git branch
```

It should show:

```
$ git branch
  master
  * training
```

Indicating that you are on the 'training' branch. More specifically, you now have a 'local' branch named 'training', which is set up to track a 'remote' branch on the 'origin' repository, named 'training' (origin/training).

To see all local and remote branches, type 'git branch' with the '-a' argument. Here is sample output:

```
$ git branch -a
* training
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/training
  remotes/origin/master
```

The first line above with the asterisk indicates that the repository is on a local branch (just local to my machine) named ‘training. The second line says that I also have a local branch called ‘master’. The remaining lines are references to branches that are on the ‘remote’ (i.e. on GitLab.com).

When doing the training, you will use the ‘training branch. When doing ns-3 projects in the future, you may want to start a separate local branch to store your project-specific code, and you will want to 'push' it to GitLab.com (or some other Git hosting provider) if you want to store it online as well.

Branch status

Every change to the Git repository is called a ‘commit’ and each one is identified by a hash value of the changes to the code. The most recent commit is called the ‘tip’ of the repository. The tip can be checked using the ‘git log’ command (to see all commits), and specifically, the ‘git log -1’ command will show only the last commit e.g.:

```
$ git log -1
commit 1f60946779313aca94dba73ac22f4790da340da3
Author: Tom Henderson <tomh@tomh.org>
Date:   Sat Jan 12 17:57:23 2019 -0800

    Remove priority-queue for dependence on libpcap
```

Let’s say that you have not made any changes to the code at the tip of the repository. Your repository is said to be ‘clean’. The command ‘git diff’ can be used to check whether your repository is clean or ‘dirty’.

```
$ git diff
```

If it returns with no output, there are no changes. If there are changes, then they will be displayed with the ‘-‘ character displaying old text, and ‘+’ displaying new text, such as, e.g.

```
$ git diff
diff --git a/contrib/simple-wireless/examples/wscript b/contrib/simple-
wireless/examples/wscript
index 0f8550c..52bb848 100644
--- a/contrib/simple-wireless/examples/wscript
+++ b/contrib/simple-wireless/examples/wscript
@@ -22,11 +22,6 @@ def build(bld):
     if not bld.env['ENABLE_EXAMPLES']:
         return;

-    obj = bld.create_ns3_program('queue_test',
-        ['core', 'mobility', 'network', 'internet', 'olsr', 'applications',
'simple-wireless'])
-    obj.source = 'queue_test.cc'
-    obj.env.append_value("LIB", ["pcap"])
-
     obj = bld.create_ns3_program('directional_test',
         ['core', 'mobility', 'network', 'internet', 'olsr', 'applications',
'simple-wireless'])
     obj.source = 'directional_test.cc'
```

Updating the training branch

The basic command to update the repository is 'git pull'. If we have pushed new changes to the GitLab.com repository, they can be pulled to your copy; below is some sample output.

```
$ git pull
remote: Enumerating objects: 25, done.
remote: Counting objects: 100% (25/25), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 15 (delta 8), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From https://gitlab.com/tomhenderson/ns-3-allinone
   956d078..1f60946  training     -> origin/training
Updating 956d078..1f60946
Fast-forward
 contrib/simple-wireless/examples/queue_test.cc     | 414 ------------------
--
 contrib/simple-wireless/examples/wscript           |   5 -
 .../link-performance/run-link-performance.sh       |  12 +-
 contrib/simple-wireless/model/priority-queue.cc    |  28 --
 contrib/simple-wireless/model/priority-queue.h     | 327 ----------------
 contrib/simple-wireless/wscript                    |   3 -
 6 files changed, 8 insertions(+), 781 deletions(-)
 delete mode 100644 contrib/simple-wireless/examples/queue_test.cc
 delete mode 100644 contrib/simple-wireless/model/priority-queue.cc
 delete mode 100644 contrib/simple-wireless/model/priority-queue.h
```

If there is nothing new to pull from the master repository, it will show this:

```
$ git pull
```

```
Already up-to-date.
```

Managing your local changes

You have a few choices for managing your code changes.

1) Each time you work on a new feature, start from scratch by cloning a fresh version of the repository

Issue: Will use up a lot of disk space if you keep all of them around

2) Every time you have a new feature, stay on the trainng branch and create a 'diff' of what you changed, save the diff, and then reset your branch before pulling.

Example:

Suppose you changed the M/M/1 queue program and want to save it.

git status shows you this:

```
$ git status
On branch training
Your branch is up-to-date with 'origin/training'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

      modified:   contrib/training-2019/examples/mm1-queue.cc
```

git diff shows you this:

```
$ git diff
diff --git a/contrib/training-2019/examples/mm1-queue.cc b/contrib/simple-
wireless/examples/mm1-queue.cc
index af0796383..375335143 100644
--- a/contrib/training-2019/examples/mm1-queue.cc
+++ b/contrib/training-2019/examples/mm1-queue.cc
@@ -154,6 +154,7 @@ QueueTracer::DropTracer (Ptr<OutputStreamWrapper>
stream, Ptr<Queue<Packet> > qu
 {
   NS_LOG_DEBUG ("Trace drop at time " << Simulator::Now ().GetSeconds
());
   *stream->GetStream () << Simulator::Now ().GetSeconds () << " d " <<
queue->GetNPackets () << std::endl;
+  Simulation::Stop ();
 }
```

If you want to save this patch for future reference, you can direct the output of git diff into a patch file.

```
$ git diff > mm1queue-changes.patch
```

and then move this somewhere, and reset your repository.

You can either checkout again the changed file (which will wipe out your local changes).

```
$ git checkout contrib/training-2019/examples/mm1-queue.cc
```

Or you can reset the whole repository if you want to restore multiple files all at once:

```
$ git reset --hard HEAD
```

Now your 'git status' should show a clean repository, and you can go ahead and do 'git pull'.

3) Create a new branch for every assignment and project

This is probably the solution most philosophically aligned with Git, because branching is a cheap operation, and Git was really designed to allow users to have multiple branches. This works as follows.

- do not create a local training branch, but instead, branch from 'origin/training' each time you need a new branch
- for each fresh experiment, create a new branch, branching from the training branch. You can do this as follows:

```
$ git pull
$ git checkout -b my-first-training origin/training
$ git log
```

This tells git to create a new 'my-first-training' branch, starting from the origin/training branch. There should never be a merge conflict with this pull operation, because what pull does is that it downloads the new changesets to origin/training and stores them in your local repository. The last command is a reminder to check that you have the right changeset at the tip of your new branch.

Now, you can make your changes and eventually commit them. When done:

```
$ git commit -m"Finished making changes" -a
$ git pull
$ git checkout -b my-next-training origin/training
```

The first command commits all changed files. If you created new files that you want to save, you should do a 'git add' before this.

If you are already working on 'my-first-training' branch, for example, and we announce an update to the training branch (e.g. new scripts, a bug fix) that you want to merge in, you can do this (from your notional 'my-first-training' branch):

```
$ git fetch
$ git merge origin/training
```

NetAnim

Note: This is optional and will not be used in the training.

The NetAnim animator that will be demonstrated can be installed also (optionally-- not needed to complete training). It relies on the Qt development libraries. (Qt version 5, from the Finnish company Qt).

In Ubuntu Linux, you should install these libraries (and on other distributions, please just search for how to install Qt5 development libraries):

```
sudo apt-get install qt5-default
```

Once Qt5 is installed, you can install netanim by downloading the 'ns-3-allinone' package from the ns-3 project, and the './build.py' script will build it.

You can 'cd' into the 'netanim' directory and run NetAnim as follows:

```
./NetAnim
```

Then you need to load an XML file (that you may have generated from the wifi-dcf program); there is a little folder icon in the upper left hand corner that you can use to search for this.