NS3 Simulator Tutorial

Release 3.30

Data Communication Networks

By Dr. Pakravan

TA: Mohammad Zangooei

Sharif University of Technology

Department of Electrical Engineering

Fall 2020

Contents

Introduction 1
Development Environment 1
Getting Started 1
Installation 2
Dependencies 2
Download 2
Build 2
Testing 2
Conceptual Overview 3
Key Abstractions 3
A First ns-3 Script 3
Main Function 4
NodeContainer 4
PointToPointHelper 4
NetDeviceContainer 5
InternetStackHelper 5
Ipv4AddressHelper 5
UdpEchoServerHelper 6
UdpEchoClientHelper 6
Simulator 6
Run 7
Some Points 7
Log Levels 7
Hooking Your Own Values ..... **Error! Bookmark not defined.**
Useful links 8

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.

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.

Development Environment

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.

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

Typically, an ns-3 author will work in Linux or a Linux-like environment. For those running under Windows, there do exist environments which simulate the Linux environment to various degrees.

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.

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. 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.

Installation

Dependencies

The ns-3 system as a whole is a fairly complex system and has a number of dependencies on other components. You can find full details on the original website but we suggest followings for Ubuntu systems.

```
$ sudo apt install gcc g++ python python3 python3-dev qt5-default mercurial
```

Download

You can download the source code using tarballs:

```
$ cd
$ mkdir ns3
$ cd ns3
$ wget http://www.nsnam.org/release/ns-allinone-3.30.tar.bz2
$ tar xjf ns-allinone-3.30.tar.bz2
```

If you change into the directory ns-allinone-3.30 you should see a number of files:

```
bake       constants.py   ns-3.30            README
build.py   netanim-3.108  pybindgen-0.20.0   util.py
```

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

Build

In ns3 directory, you can execute this command to build

```
$ ./build.py --enable-tests --enable-examples
```

Here, we enable tests, so that we could check whether the installation is correct.

Testing

Do the following:

```
$ cd ns-3.30
$ ./test.py
```

It will take quite a long time, and at last you will see the results.

Moreover, you can run a sample script which allows the build system to ensure that the shared library paths are set correctly and that the libraries are available at run time.

To run a program, simply use the --run option in Waf.

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

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

```
Hello Simulator
```

Congratulations! You are now an ns-3 user!

Conceptual Overview

Key Abstractions

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

- **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
- **Application**: 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`.
- **Channel**: Often the media over which data flows in these networks are called channels. 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`
- **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`.
- **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`.

A First ns-3 Script

Let's investigate an example. You can find it in the `examples/tutorial` directory under the title of `first.cc`. 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.

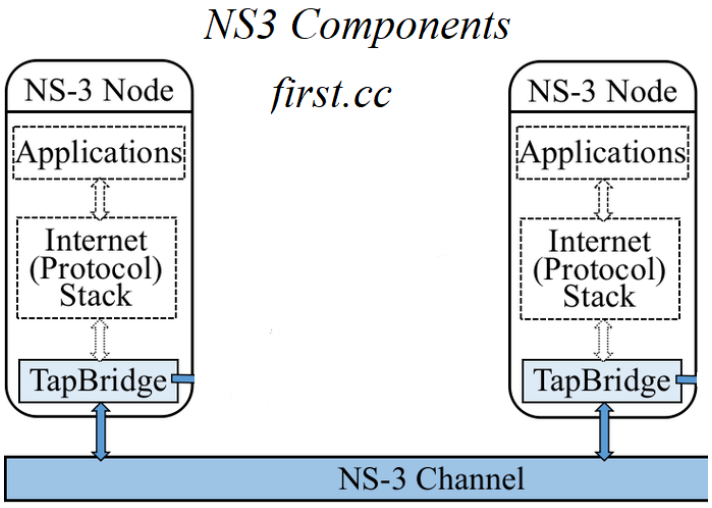
Main Function

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

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

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. (see the list of all available components here) This will result in the application printing out messages as packets are sent and received during the simulation. 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);
```



NodeContainer

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 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);
```

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.

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. The next three lines in the script are,

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

The first line instantiates a `PointToPointHelper` object on the stack, the next line, tells the `PointToPointHelper` object to use the value "5Mbps" (five megabits per second) as the "DataRate". From a more detailed perspective, the string "DataRate" corresponds to what we call an `Attribute` of the `PointToPointNetDevice`. Similar to the "DataRate" on the `PointToPointNetDevice` you will find a "Delay" `Attribute` associated with the `PointToPointChannel`.

NetDeviceContainer

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 codes will finish configuring the devices and channel.

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

After executing above commands 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. The next two lines of code in our example 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.

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

The next line of code, performs the actual address assignment.

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

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.

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.

```
UdpEchoServerHelper echoServer (9);
ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
```

The first line of code in the above snippet declares the `UdpEchoServerHelper`. As usual, this isn't the application itself, it is an object used to help us create the actual applications. 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. Applications require a time to "start" generating traffic and may take an optional time to "stop". We provide both.

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 (1.0)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));
ApplicationContainer clientApps = echoClient.Install (nodes.Get (0));
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

For the echo client, however, we need to set five different `Attributes`. The first two `Attributes` are set during construction of the `UdpEchoClientHelper` and used (internally to the helper) to set the "RemoteAddress" and "RemotePort" (UDP Server). 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).

Simulator

What we need to do at this point is to actually run the simulation.

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

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. 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.

When 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

Run

Copy the file to scratch directory.

```
$ cp ~/ns3/ns-allinone-3.30/ns-3.30/examples/tutorial/first.cc ~/ns3/ns-allinone-3.30/ns-3.30/scratch
```

Change your working directory to `~/ns3/ns-allinone-3.30/ns-3.30` then run the script

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

After that you will see the result below:

```
dn-ns3@dn-ns3:~/ns3/ns-allinone-3.30/ns-3.30$ ./waf --run "scratch/first"
Waf: Entering directory `/home/dn-ns3/ns3/ns-allinone-3.30/ns-3.30/build'
[2729/2782] Compiling scratch/subdir/scratch-simulator-subdir.cc
[2731/2782] Compiling scratch/first.cc
[2741/2782] Linking build/scratch/subdir/subdir
[2742/2782] Linking build/scratch/first
Waf: Leaving directory `/home/dn-ns3/ns3/ns-allinone-3.30/ns-3.30/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (6.030s)
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
```

Some Points

Log Levels

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 easyto use way to get useful information out of your simulation.

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_DEBUG — Log relatively rare, ad-hoc debugging messages (associated macro: NS_LOG_DEBUG);
• 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_ALL — Log everything mentioned above (no associated macro).

Using Command Line Arguments

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*.

```
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);
...
```

Recall the following in our previous code:

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

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

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

Useful links

Original Documentation https://www.nsnam.org/releases/ns-3-30/documentation/

API Documentation: https://www.nsnam.org/docs/release/3.30/doxygen/index.html

A useful YouTube channel: https://www.youtube.com/c/AdilAlsuhaim/videos