[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2 A First ns-3 Script

If you downloaded the system as was suggested above, you will have a release of ns-3 in a directory called repos under your home directory. Change into that release directory, and you should find a directory structure something like the following:

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

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.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.1 Boilerplate

The first line in the file is an emacs mode line. This tells emacs about the formatting conventions (coding style) we use in our source code.

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

This is always a somewhat controversial subject, so we might as well get it out of the way immediately. The ns-3 project, like most large projects, has adopted a coding style to which all contributed code must adhere. If you want to contribute your code to the project, you will eventually have to conform to the ns-3 coding standard as described in the file doc/codingstd.txt or shown on the project web page here.

We recommend that you, well, just get used to the look and feel of ns-3 code and adopt this standard whenever you are working with our code. All of the development team and contributors have done so with various amounts of grumbling. The emacs mode line above makes it easier to get the formatting correct if you use the emacs editor.

The ns-3 simulator is licensed using the GNU General Public License. You will see the appropriate GNU legalese at the head of every file in the ns-3 distribution. Often you will see a copyright notice for one of the institutions involved in the ns-3 project above the GPL text and an author listed below.

```
/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
```

```
*/
```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.2 Module Includes

The code proper starts with a number of include statements.

```
#include "ns3/core-module.h"
#include "ns3/simulator-module.h"
#include "ns3/node-module.h"
#include "ns3/helper-module.h"
```

To help our high-level script users deal with the large number of include files present in the system, we group includes according to relatively large modules. We provide a single include file that will recursively load all of the include files used in each module. Rather than having to look up exactly what header you need, and possibly have to get a number of dependencies right, we give you the ability to load a group of files at a large granularity. This is not the most efficient approach but it certainly makes writing scripts much easier.

Each of the `ns-3` include files is placed in a directory called `ns3` (under the build directory) during the build process to help avoid include file name collisions. The `ns3/core-module.h` file corresponds to the ns-3 module you will find in the directory `src/core` in your downloaded release distribution. If you list this directory you will find a large number of header files. When you do a build, Waf will place public header files in an `ns3` directory under the appropriate `build/debug` or `build/optimized` directory depending on your configuration. Waf will also automatically generate a module include file to load all of the public header files.

Since you are, of course, following this tutorial religiously, you will already have done a

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

in order to configure the project to perform debug builds. You will also have done a

```
./waf
```

to build the project. So now if you look in the directory `../../build/debug/ns3` you will find the four module include files shown above. You can take a look at the contents of these files and find that they do include all of the public include files in their respective modules.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.3 Ns3 Namespace

The next line in the `first.cc` script is a namespace declaration.

```
using namespace ns3;
```

The `ns-3` project is implemented in a C++ namespace called `ns3`. This groups all `ns-3`-related declarations in a scope outside the global namespace, which we hope will help with integration with other code. The C++ `using` statement introduces the `ns-3` namespace into the current (global) declarative region. This is a fancy way of saying that after this declaration, you will not have to type `ns3::` scope resolution operator before all of the `ns-3` code in order to use it. If you are unfamiliar with namespaces, please consult almost any C++ tutorial and compare the `ns3` namespace and usage here with instances of the `std` namespace and the `using namespace std;` statements you will often find in discussions of `cout`

and streams.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.4 Logging

The next line of the script is the following,

```
NS_LOG_COMPONENT_DEFINE ("FirstScriptExample");
```

We will use this statement as a convenient place to talk about our Doxygen documentation system. If you look at the project web site, ns-3 project, you will find a link to “Doxygen (ns-3-dev)” in the navigation bar. If you select this link, you will be taken to our documentation page for the current development release. There is also a link to “Doxygen (stable)” that will take you to the documentation for the latest stable release of ns-3.

Along the left side, you will find a graphical representation of the structure of the documentation. A good place to start is the NS-3 Modules “book” in the ns-3 navigation tree. If you expand Modules you will see a list of ns-3 module documentation. The concept of module here ties directly into the module include files discussed above. It turns out that the ns-3 logging subsystem is part of the core module, so go ahead and expand that documentation node. Now, expand the Debugging book and then select the Logging page.

You should now be looking at the Doxygen documentation for the Logging module. In the list of #defines 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.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.5 Main Function

The next lines of the script you will find are,

```
  int
main (int argc, char *argv[])
{
```

This is just the declaration of the main function of your program (script). Just as in any C++ program, you need to define a main function that will be the first function run. There is nothing at all special here. Your ns-3 script is just a C++ program.

The next two lines of the script are used to enable two logging components that are built into the Echo Client and Echo Server applications:

```
  LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
  LogComponentEnable("UdpEchoServerApplication", LOG_LEVEL_INFO);
```

If you have read over the Logging component documentation you will have seen that there are a number

of levels of logging verbosity/detail that you can enable on each component. These two lines of code enable debug logging at the INFO level for echo clients and servers. This will result in the application printing out messages as packets are sent and received during the simulation.

Now we will get directly to the business of creating a topology and running a simulation. We use the topology helper objects to make this job as easy as possible.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.6 Topology Helpers

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.6.1 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.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.6.2 PointToPointHelper

We are constructing a point to point link, and, in a pattern which will become quite familiar to you, we use a topology helper object to do the low-level work required to put the link together. Recall that two of our key abstractions are the NetDevice and the Channel. In the real world, these terms correspond roughly to peripheral cards and network cables. Typically these two things are intimately tied together and one cannot expect to interchange, for example, Ethernet devices and wireless channels. Our Topology Helpers follow this intimate coupling and therefore you will use a single PointToPointHelper to configure and connect ns-3 PointToPointNetDevice and PointToPointChannel objects in this script.

The next three lines in the script are,

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

The first line,

```
PointToPointHelper pointToPoint;
```

instantiates a PointToPointHelper object on the stack. From a high-level perspective the next line,

```
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
```

tells the PointToPointHelper object to use the value “5Mbps” (five megabits per second) as the “DataRate” when it creates a PointToPointNetDevice object.

From a more detailed perspective, the string “DataRate” corresponds to what we call an Attribute of the PointToPointNetDevice. If you look at the Doxygen for class ns3::PointToPointNetDevice and find the documentation for the GetTypeId method, you will find a list of Attributes defined for the device. Among these is the “DataRate” Attribute. Most user-visible ns-3 objects have similar lists of Attributes. We use this mechanism to easily configure simulations without recompiling as you will see in a following section.

Similar to the “DataRate” on the PointToPointNetDevice you will find a “Delay” Attribute associated with the PointToPointChannel. The final line,

```
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

tells the PointToPointHelper to use the value “2ms” (two milliseconds) as the value of the transmission delay of every point to point channel it subsequently creates.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.6.3 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.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.6.4 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.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.6.5 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.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.7 Applications

Another one of the core abstractions of the ns-3 system is the `Application`. In this script we use two specializations of the core `ns-3` class `Application` called `UdpEchoServerApplication` and `UdpEchoClientApplication`. Just as we have in our previous explanations, we use helper objects to help configure and manage the underlying objects. Here, we use `UdpEchoServerHelper` and `UdpEchoClientHelper` objects to make our lives easier.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.7.1 UdpEchoServerHelper

The following lines of code in our example script, `first.cc`, are used to set up a UDP echo server application on one of the nodes we have previously created.

```
UdpEchoServerHelper echoServer (9);

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

The first line of code in the above snippet declares the `UdpEchoServerHelper`. As usual, this isn't the application itself, it is an object used to help us create the actual applications. One of our conventions is to place *required* `Attributes` in the helper constructor. In this case, the helper can't do anything useful unless it is provided with a port number that the client also knows about. Rather than just picking one and hoping it all works out, we require the port number as a parameter to the constructor. The constructor, in turn, simply does a `SetAttribute` with the passed value. If you want, you can set the "Port" `Attribute` to another value later using `SetAttribute`.

Similar to many other helper objects, the `UdpEchoServerHelper` object has an `Install` method. It is the execution of this method that actually causes the underlying echo server application to be instantiated and attached to a node. Interestingly, the `Install` method takes a `NodeContainter` as a parameter just as the other `Install` methods we have seen. This is actually what is passed to the method even though it doesn't look so in this case. There is a C++ *implicit conversion* at work here that takes the result of `nodes.Get (1)` (which returns a smart pointer to a node object — `Ptr<Node>`) and uses that in a constructor for an unnamed `NodeContainer` that is then passed to `Install`. If you are ever at a loss to find a particular method signature in C++ code that compiles and runs just fine, look for these kinds of implicit conversions.

We now see that `echoServer.Install` is going to install a `UdpEchoServerApplication` on the node found at index number one of the `NodeContainer` we used to manage our nodes. `Install` will return a container that holds pointers to all of the applications (one in this case since we passed a `NodeContainer` containing one node) created by the helper.

Applications require a time to "start" generating traffic and may take an optional time to "stop". We provide both. These times are set using the `ApplicationContainer` methods `Start` and `Stop`. These methods take `Time` parameters. In this case, we use an *explicit* C++ conversion sequence to take the C++ double 1.0 and convert it to an `ns-3 Time` object using a `Seconds` cast. Be aware that the conversion rules may be controlled by the model author, and C++ has its own rules, so you can't always just assume that parameters will be happily converted for you. The two lines,

```
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
```

will cause the echo server application to `Start` (enable itself) at one second into the simulation and to

Stop (disable itself) at ten seconds into the simulation. By virtue of the fact that we have declared a simulation event (the application stop event) to be executed at ten seconds, the simulation will last *at least* ten seconds.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.7.2 UdpEchoClientHelper

The echo client application is set up in a method substantially similar to that for the server. There is an underlying UdpEchoClientApplication that is managed by an UdpEchoClientHelper.

```
UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer clientApps = echoClient.Install (nodes.Get (0));
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

For the echo client, however, we need to set five different Attributes. The first two Attributes are set during construction of the UdpEchoClientHelper. We pass parameters that are used (internally to the helper) to set the “RemoteAddress” and “RemotePort” Attributes in accordance with our convention to make required Attributes parameters in the helper constructors.

Recall that we used an Ipv4InterfaceContainer to keep track of the IP addresses we assigned to our devices. The zeroth interface in the interfaces container is going to correspond to the IP address of the zeroth node in the nodes container. The first interface in the interfaces container corresponds to the IP address of the first node in the nodes container. So, in the first line of code (from above), we are creating the helper and telling it so set the remote address of the client to be the IP address assigned to the node on which the server resides. We also tell it to arrange to send packets to port nine.

The “MaxPackets” Attribute tells the client the maximum number of packets we allow it to send during the simulation. The “Interval” Attribute tells the client how long to wait between packets, and the “PacketSize” Attribute tells the client how large its packet payloads should be. With this particular combination of Attributes, we are telling the client to send one 1024-byte packet.

Just as in the case of the echo server, we tell the echo client to Start and Stop, but here we start the client one second after the server is enabled (at two seconds into the simulation).

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.8 Simulator

What we need to do at this point is to actually run the simulation. This is done using the global function Simulator::Run.

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

When we previously called the methods,

```
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
...
clientApps.Start (Seconds (2.0));
```

```
  clientApps.Stop (Seconds (10.0));
```

we actually scheduled events in the simulator at 1.0 seconds, 2.0 seconds and two events at 10.0 seconds. When `Simulator::Run` is called, the system will begin looking through the list of scheduled events and executing them. First it will run the event at 1.0 seconds, which will enable the echo server application (this event may, in turn, schedule many other events). Then it will run the event scheduled for t=2.0 seconds which will start the echo client application. Again, this event may schedule many more events. The start event implementation in the echo client application will begin the data transfer phase of the simulation by sending a packet to the server.

The act of sending the packet to the server will trigger a chain of events that will be automatically scheduled behind the scenes and which will perform the mechanics of the packet echo according to the various timing parameters that we have set in the script.

Eventually, since we only send one packet (recall the `MaxPackets Attribute` was set to one), the chain of events triggered by that single client echo request will taper off and the simulation will go idle. Once this happens, the remaining events will be the `Stop` events for the server and the client. When these events are executed, there are no further events to process and `Simulator::Run` returns. The simulation is then complete.

All that remains is to clean up. This is done by calling the global function `Simulator::Destroy`. As the helper functions (or low level `ns-3` code) executed, they arranged it so that hooks were inserted in the simulator to destroy all of the objects that were created. You did not have to keep track of any of these objects yourself — all you had to do was to call `Simulator::Destroy` and exit. The `ns-3` system took care of the hard part for you. The remaining lines of our first `ns-3` script, `first.cc`, do just that:

```
  Simulator::Destroy ();
  return 0;
}
```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4.2.9 Building Your Script

We have made it trivial to build your simple scripts. All you have to do is to drop your script into the scratch directory and it will automatically be built if you run Waf. Let's try it. Copy `examples/tutorial/first.cc` into the `scratch` directory after changing back into the top level directory.

```
cd ..
cp examples/tutorial/first.cc scratch/myfirst.cc
```

Now build your first example script using waf:

```
./waf
```

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

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
[614/708] cxx: scratch/myfirst.cc -> build/debug/scratch/myfirst_3.o
[706/708] cxx_link: build/debug/scratch/myfirst_3.o -> build/debug/scratch/myfirst
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (2.357s)
```

You can now run the example (note that if you build your program in the scratch directory you must run it out of the scratch directory):

```
./waf --run scratch/myfirst
```

You should see some output:

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.418s)
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.1.2
```

Here you see that the build system checks to make sure that the file has been build and then runs it. You see the logging component on the echo client indicate that it has sent one 1024 byte packet to the Echo Server on 10.1.1.2. You also see the logging component on the echo server say that it has received the 1024 bytes from 10.1.1.1. The echo server silently echoes the packet and you see the echo client log that it has received its packet back from the server.

[<] [>] [<<] [Up] [>>]

This document was generated on *April 21, 2010* using *texi2html 1.82*.