INTRODUCTION TO NS3.5

Outline

INTRODUCTION TO NS3

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.
ns-3 takes the view that all of these verbosity levels are useful and we provide a selectable, multi-level approach to message logging. Logging can be disabled completely, enabled on a component-by-component basis, or enabled globally; and it provides selectable verbosity levels. The ns-3 log module provides a straightforward, relatively easy to use way to get useful information out of your simulation.

ASCII Tracing

ns-3 provides helper functionality that wraps the low-level tracing system to help you with the details involved in configuring some easily understood packet traces. If you enable this functionality, you will see output in a ASCII files - thus the name. For those familiar with ns-2 output, this type of trace is analogous to the out.tr generated by many scripts.

```
AsciiTraceHelper ascii;
pointToPoint.EnableAsciiAll (ascii.CreateFileStream ("myfirst.tr"));
```

The outside call, to EnableAsciiAll(), tells the helper that you want to enable ASCII tracing on all point-to-point devices in your simulation; and you want the (provided) trace sinks to write out information about packet movement in ASCII format.

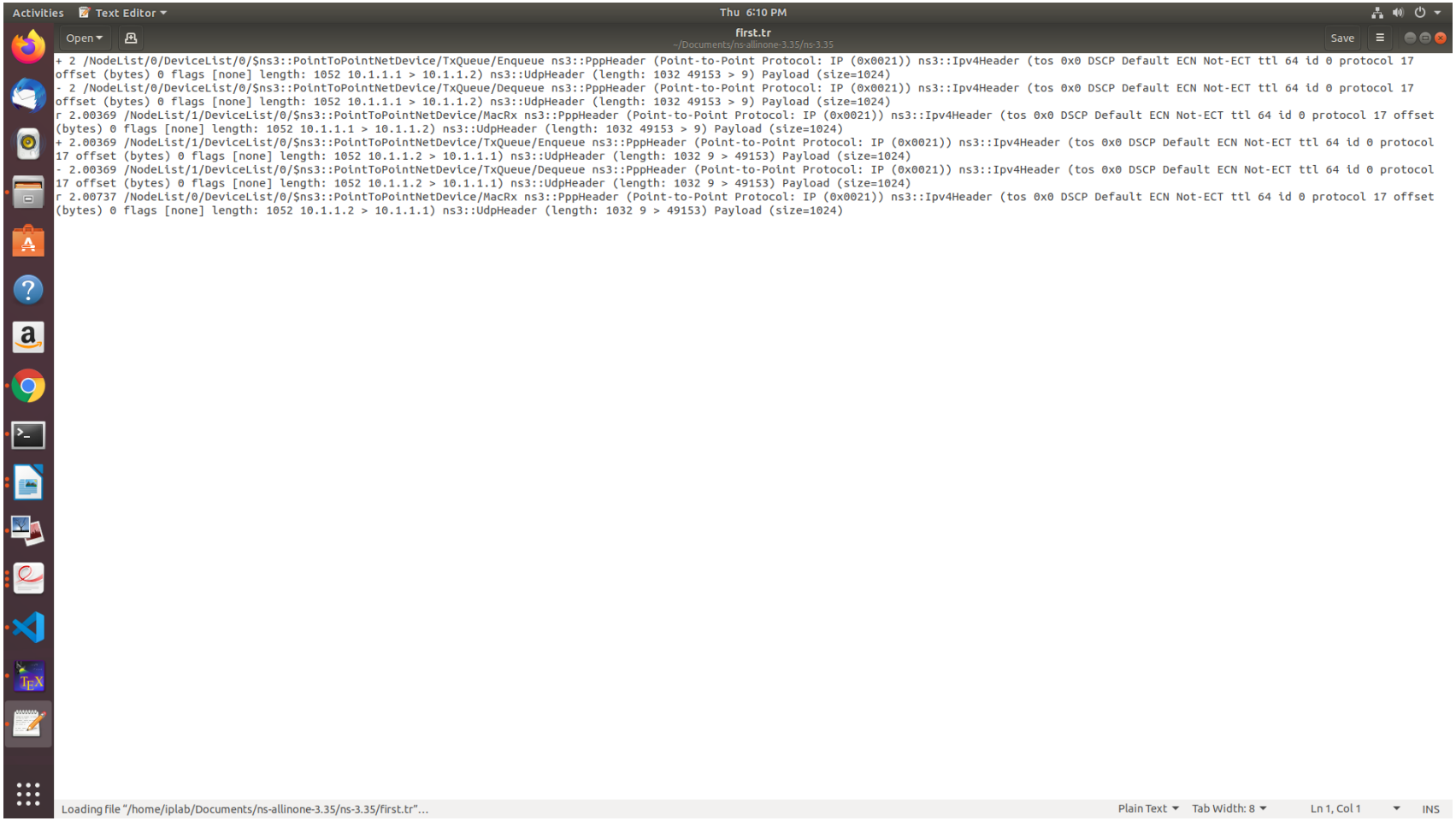

Figure 1: first.tr trace file

Parsing Ascii Traces

Each line in the file corresponds to a trace event. In this case we are tracing events on the transmit queue present in every point-to-point net device in the simulation. The transmit queue is a queue through which every packet destined for a point-to-point channel must pass. Note that each line in the trace file begins with a lone character (has a space after it). This character will have the following meaning:

1. +: An enqueue operation occurred on the device queue;
2. -: A dequeue operation occurred on the device queue;
3. d: A packet was dropped, typically because the queue was full;
4. r: A packet was received by the net device

PCAP Tracing

The ns-3 device helpers can also be used to create trace files in the .pcap format. The acronym pcap (usually written in lower case) stands for packet capture, and is actually an API that includes the definition of a .pcap file format. The most popular program that can read and display this format is Wireshark (formerly called Ethereal). However, there are many traffic trace analyzers that use this packet format. We encourage users to exploit the many tools available for analyzing pcap traces. In this tutorial, we concentrate on viewing pcap traces with tcpdump.

pcap files tracing

The code used to enable pcap tracing is a one-liner.
pointToPoint.EnablePcapAll ("myfirst");

Notice that we only passed the string "myfirst," and not "myfirst.pcap" or something similar. This is because the parameter is a prefix, not a complete file name. The helper will actually create a trace file for every point-to-point device in the simulation. The file names will be built using the prefix, the node number, the device number and a ".pcap" suffix. In our example script, we will eventually see files named "myfirst-0-0.pcap" and "myfirst-1-0.pcap" which are the pcap traces for node 0-device 0 and node 1-device 0, respectively.

pcap of first.cc

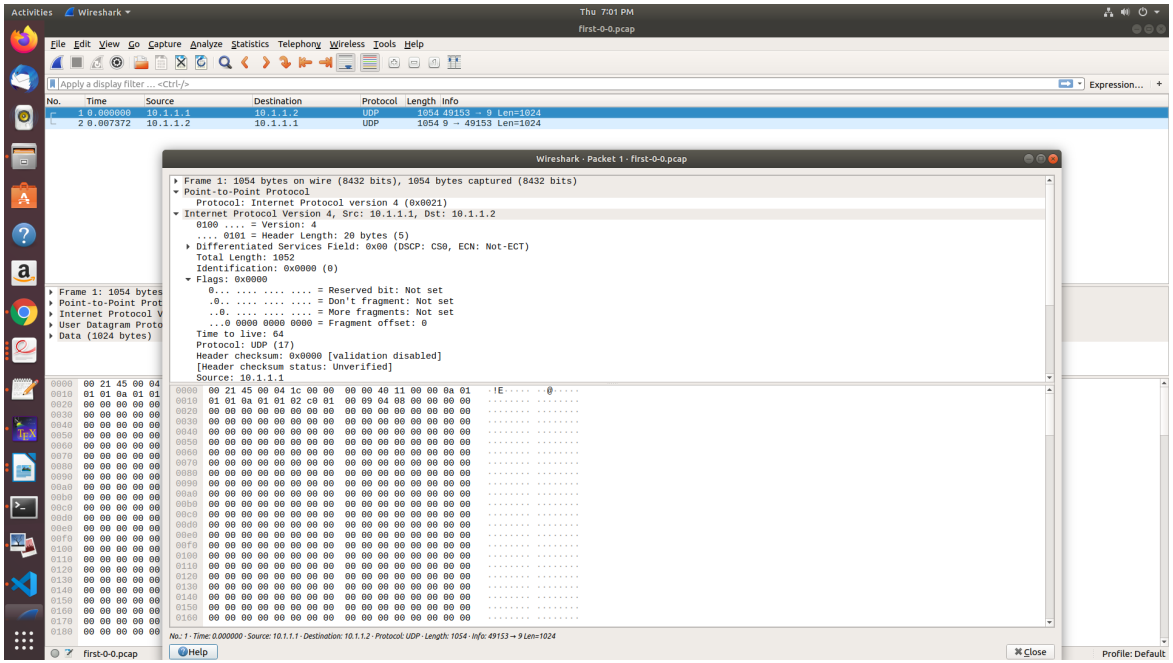

Figure 2: first.cc pcap file

Using the Logging Module

1. There are currently seven levels of log messages of increasing verbosity defined in the system.
2. LOG_ERROR - Log error messages (associated macro: NS_LOG_ERROR);
3. LOG_WARN - Log warning messages (associated macro: NS_LOG_WARN);
4. LOG_DEBUG - Log relatively rare, ad-hoc debugging messages (associated macro: NS_LOG_DEBUG);
5. LOG_INFO - Log informational messages about program progress (associated macro: NS_LOG_INFO);
6. 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);
7. LOG_LOGIC - Log messages describing logical flow within a function (associated macro: NS_LOG_LOGIC);
8. LOG_ALL - Log everything mentioned above (no associated macro).

Key Abstractions

Node

n Internet jargon, a computing device that connects to a network is called a host or sometimes an end system. Because ns-3 is a network simulator, not specifically an Internet simulator, we intentionally do not use the term host since it is closely associated with the Internet and its protocols. Instead, we use a more generic term also used by other simulators that originates in Graph Theory - the node.

In ns-3 the basic computing device abstraction is called the node. This abstraction is represented in C++ by the class Node. The Node class provides methods for managing the representations of computing devices in simulations. You should think of a Node as a computer to which you will add functionality. One adds things like applications, protocol stacks and peripheral cards with their associated drivers to enable the computer to do useful work. We use the same basic model in ns-3.

Application

In ns-3 there is no real concept of operating system and especially no concept of privilege levels or system calls. We do, however, have the idea of an application. Just as software applications run on computers to perform tasks in the ?real world,? ns-3 applications run on ns-3 Nodes to drive simulations in the simulated world.

In ns-3 the basic abstraction for a user program that generates some activity to be simulated is the application. This abstraction is represented in C++ by the class Application. The Application class provides meth- ods for managing the representations of our version of user-level applications in simulations. Developers are ex- pected to specialize the Application class in the object-oriented programming sense to create new applications. In this tutorial, we will use specializations of class Application called UdpEchoClientApplication and UdpEchoServerApplication. As you might expect, these applications compose a client/server application set used to generate and echo simulated network packets

Channel

In the real world, one can connect a computer to a network. Often the media over which data flows in these networks are called channels. When you connect your Ethernet cable to the plug in the wall, you are connecting your computer to an Ethernet communication channel. In the simulated world of ns-3, one connects a Node to an object representing a communication channel. Here the basic communication subnetwork abstraction is called the channel and is represented in C++ by the class Channel.

The Channel class provides methods for managing communication subnetwork objects and connecting nodes to them. Channels may also be specialized by developers in the object oriented programming sense. A Channel specialization may model something as simple as a wire. The specialized Channel can also model things as complicated as a large Ethernet switch, or three-dimensional space full of obstructions in the case of wireless networks.

We will use specialized versions of the Channel called CsmaChannel, PointToPointChannel and WifiChannel in this tutorial. The CsmaChannel, for example, models a version of a communication subnetwork that implements a carrier sense multiple access communication medium. This gives us Ethernet-like functionality.

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 hard- ware 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.

NetDevice

In ns-3 the net device abstraction covers both the software driver and the simulated hardware. A net device is "installed" in a Node in order to enable the Node to communicate with other Nodes in the simulation via Channels. Just as in a real computer, a Node may be connected to more than one Channel via multiple NetDevices.

The net device abstraction is represented in C++ by the class NetDevice. The NetDevice class provides methods for managing connections to Node and Channel objects; and may be specialized by developers in the object-oriented programming sense. We will use the several specialized versions of the NetDevice called CsmaNetDevice, PointToPointNetDevice, and WifiNetDevice in this tutorial. Just as an Ethernet NIC is designed to work with an Ethernet network, the CsmaNetDevice is designed to work with a CsmaChannel; the PointToPointNetDevice is designed to work with a PointToPointChannel and a WifiNetNevice is designed to work with a WifiChannel.

Topology Helpers

In a real network, you will find host computers with added (or built-in) NICs. In ns-3 we would say that you will find Nodes with attached NetDevices. In a large simulated network you will need to arrange many connections between Nodes, NetDevices and Channels. Since connecting NetDevices to Nodes, NetDevices to Channels, assigning IP addresses, etc., are such common tasks in ns-3, we provide what we call topology helpers to make this as easy as possible. For example, it may take many distinct ns-3 core operations to create a NetDevice, add a MAC address, install that net device on a Node, configure the node's protocol stack, and then connect the NetDevice to a Channel. Even more operations would be required to connect multiple devices onto multipoint channels and then to connect individual networks together into internetworks. We provide topology helper objects that combine those many distinct operations into an easy to use model for your convenience.

Building a Bus Network Topology

Topology Helpers

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);
```

You may recall that one of our key abstractions is the Node. This represents a computer to which we are going to add things like protocol stacks, applications and peripheral cards. The NodeContainer topology helper provides a convenient way to create, manage and access any Node objects that we create in order to run a simulation. The first line above just declares a NodeContainer which we call nodes. The second line calls the Create method on the nodes object and asks the container to create two nodes. As described in the Doxygen, the container calls down into the ns-3 system proper to create two Node objects and stores pointers to those objects internally.

The nodes as they stand in the script do nothing. The next step in constructing a topology is to connect our nodes together into a network. The simplest form of network we support is a single point-to-point link between two nodes. We'll construct one of those links here.

PointToPointHelper

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

The next three lines in the script are

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

PointToPointHelper pointToPoint;
instantiates a PointToPointHelper object on the stack. From a high-level perspective the next line,
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
tells the PointToPointHelper object to use the value "5Mbps" (five megabits per second) as the "DataRate" when it creates a PointToPointNetDevice object.
From a more detailed perspective, the string "DataRate" corresponds to what we call an Attribute of the PointToPointNetDevice. If you look at the Doxygen for class ns3::PointToPointNetDevice and find the documentation for the GetTypeId method, you will find a list of Attributes defined for the device. Among these is the "DataRate" Attribute. Most user-visible ns-3 objects have similar lists of Attributes. We use this mechanism to easily configure simulations without recompiling as you will see in a following section.

Similar to the "DataRate" on the PointToPointNetDevice you will find a "Delay" Attribute associated with the PointToPointChannel. The final line,
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
tells the PointToPointHelper to use the value "2ms" (two milliseconds) as the value of the propagation delay of every point to point channel it subsequently creates.

NetDeviceContainer

At this point in the script, we have a NodeContainer that contains two nodes. We have a PointToPointHelper that is primed and ready to make PointToPointNetDevices and wire PointToPointChannel objects between them. Just as we used the NodeContainer topology helper object to create the Nodes for our simulation, we will ask the PointToPointHelper to do the work involved in creating, configuring and installing our devices for us. We will need to have a list of all of the NetDevice objects that are created, so we use a NetDeviceContainer to hold them just as we used a NodeContainer to hold the nodes we created. The following two lines of code,

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

NetDeviceContainer

The first line declares the device container mentioned above and the second does the heavy lifting. The Install method of the PointToPointHelper takes a NodeContainer as a parameter. Internally, a NetDeviceContainer is created. For each node in the NodeContainer (there must be exactly two for a point-to-point link) a PointToPointNetDevice is created and saved in the device container. A PointToPointChannel is created and the two PointToPointNetDevices are attached. When objects are created by the PointToPointHelper, the Attributes previously set in the helper are used to initialize the corresponding Attributes in the created objects.

After executing the pointToPoint.Install (nodes) call we will have two nodes, each with an installed point-to-point net device and a single point-to-point channel between them. Both devices will be configured to transmit data at five megabits per second over the channel which has a two millisecond transmission delay.

InternetStackHelper

InternetStackHelper stack;
stack.Install (nodes);
The InternetStackHelper is a topology helper that is to internet stacks what the PointToPointHelper is to point-to-point net devices. The Install method takes a NodeContainer as a parameter. When it is executed, it will install an Internet Stack (TCP, UDP, IP, etc.) on each of the nodes in the node container.

Ipv4AddressHelper

Next we need to associate the devices on our nodes with IP addresses. We provide a topology helper to manage the allocation of IP addresses. The only user-visible API is to set the base IP address and network mask to use when performing the actual address allocation (which is done at a lower level inside the helper).

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).

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.

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 UdpEchoServer-Application and UdpEchoClientApplication. Just as we have in our previous explanations, we use helper objects to help configure and manage the underlying objects. Here, we use UdpEchoServerHelper and UdpEchoClientHelper objects to make our lives easier.

UdpEchoServerHelper

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

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

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));
```

Simulator

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

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


THANK YOU