1 Lab 04: Understanding network protocols using ns3

A *simulator* is a program that models the behaviour of a given system, which allows you to study and analyze complex systems like networks. Simulators consist of objects that represent the entities in the system, and timed “events” that represent the changes happening in the system. Events are created according to some rates and times specified by the user. Events change the states of objects in the simulator, and this change can result in more events. This is how a simulation continues.

Any *network* simulator typically takes as input the following:

- Description of the physical topology of the network - the nodes, the links, the bandwidth and delay specifications of the links.
- Description of “flows” from end hosts in this network. The application type, the packet size, packet generation rate, etc.
- Specification of protocols to be simulated by the hosts and routers of the network. Protocol parameters if any (e.g. window size).

A network simulator then creates the underlying node objects, link objects, packet source and sink objects and dynamically creates packet objects at the specified rates, and queues these packets at links or nodes at times corresponding to given bandwidths and delays. It records the timestamps of various events (e.g. packet arrivals, drops, etc) as they happen in the system and logs them into a *trace file*. You can then process this trace file to get aggregate quantities such as throughputs, packet delays, drop rates etc.

1.1 Exercise 1: ns3- Network Simulator-3 Introduction tutorial

The first task today is to learn how to run an ns3 simulation, then study the simplest possible simulator code file, and ‘play with it’ for a little while.

Let’s get started. In your labuser home directory go to the ns directory

```
$ cd ns-allinone-3.29/ns-3.29
```

Here, first give these commands:

```
$ ./waf -d debug --enable-examples --enable-tests configure
$./waf
```

Now just see if you can run two very basic ns3 simulations, one is a Hello World, another is a trivial 2-node, 1 link client-server simulation.

For that, first copy the “hello simulator” program from another directory, to the “scratch” directory

```
$cp examples/tutorial/hello-simulator.cc  scratch
$./waf --run hello-simulator
```

You should see “Hello Simulator” written to screen (in addition to many compiling/linking alerts).

Now do:

```
$cp examples/tutorial/first.cc  scratch
$./waf --run first
```

You should see this written to screen (in addition to many compiling/linking alerts).

```
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
```

In the following we reproduce verbatim relevant portions of from the ns3 tutorial, with some minor edits

1.1.1 Conceptual Overview

The first thing we need to do before actually starting to look at or write ns-3 code is to explain a few core concepts and abstractions in the system. Much of this may appear transparently obvious to some, but we recommend taking the time to read through this section just to ensure you are starting on a firm foundation.

1.1.1.1 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 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

Typically, computer software is divided into two broad classes. System Software organizes various computer resources such as memory, processor cycles, disk, network, etc., according to some computing model. System software usually does not use those resources to complete tasks that directly benefit a user. A user would typically run an application that acquires and uses the resources controlled by the system software to accomplish some goal.

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 methods for managing the representations of our version of user-level applications in simulations. Developers are expected 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 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.

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.

1.1.2 First ns-3 Script

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.**
For the rest of the lab, divide your screen into two: have these instructions open in half, and open the code files in the other half.

Module Includes
The code proper starts with a number of include statements.

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
```

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 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.

Logging
The next line of the script is the following,

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

The ns-3 logging subsystem we'll get to it later in this tutorial, 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.

Main Function
The next lines of the script you will find are. No surprises here, as ns3 script is C++

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

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

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

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 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);
```

Ns3 has 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.

Topology Helpers

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);
```

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 NIC 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.

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 network Interface 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"));
```

It's obvious here that we have a 'pointToPoint' object here which has attributes of DataRate and Delay, which we have set to 5Mbps and 2ms respectively.

NetDeviceContainer

Now, we have the link helper object, now 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. 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 one creates a device container with two devices (because it is a pointToPoint helper), puts each device in the two nodes in the container "nodes", creates and attaches the point to point link to these devices, with attributes as set earlier.

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

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 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 (subnet 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. Recall that these “devices” are a NetDeviceContainer object we made earlier.

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 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.

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);
```

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*. The argument 9 specifies the port number where this UDP server will run.

```
ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));
```

echoServer.Install installeds 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.

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

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.

These two lines 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.

UdpEchoClientHelper

We start with-

```
UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);
```

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. Recall that we just started an echo server at the Node 1, at Port 9.

So, in the first line of code (from above), we are creating the helper and telling it so set the remote address (the server) of the client to be the IP address assigned to the node on which the server resides and to send packets to port nine on that server.

```
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));
```

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.

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

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).

1.1.3 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;
}
```

1.1.4 When the simulator will stop?

ns-3 is a Discrete Event (DE) simulator. In such a simulator, each event is associated with its execution time, and the simulation proceeds by executing events in the temporal order of simulation time. Events may cause future events to be scheduled (for example, a timer may reschedule itself to expire at the next interval).

The initial events are usually triggered by each object, e.g., IPv6 will schedule Router Advertisements, Neighbor Solicitations, etc., an Application schedule the first packet sending event, etc.

When an event is processed, it may generate zero, one or more events. As a simulation executes, events are consumed, but more events may (or may not) be generated. The simulation will stop automatically when no further events are in the event queue, or when a special Stop event is found. The Stop event is created through the Simulator::Stop (stopTime); function.

There is a typical case where Simulator::Stop is absolutely necessary to stop the simulation: when there is a self-sustaining event. Self-sustaining (or recurring) events are events that always reschedule themselves. As a consequence, they always keep the event queue non-empty.

There are many protocols and modules containing recurring events, e.g.:

FlowMonitor - periodic check for lost packets
RIPng - periodic broadcast of routing tables update
etc.
In these cases, Simulator::Stop is necessary to gracefully stop the simulation.

Many of the simulation programs in the tutorial do not explicitly call Simulator::Stop, since the event queue will automatically run out of events. However, these programs will also accept a call to Simulator::Stop. For example, the following additional statement in the first example program will schedule an explicit stop at 11 seconds:

```
+  Simulator::Stop (Seconds (11.0));
   Simulator::Run ();
   Simulator::Destroy ();
   return 0;
 }
```

The above will not actually change the behavior of this program, since this particular simulation naturally ends after 10 seconds. But if you were to change the stop time in the above statement from 11 seconds to 1 second, you would notice that the simulation stops before any output is printed to the screen (since the output occurs around time 2 seconds of simulation time).

It is important to call Simulator::Stop before calling Simulator::Run; otherwise, Simulator::Run may never return control to the main program to execute the stop!

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.

1.1.5 Logging/Tracing

The above log is useful, but for more detailed log:

```
$export 'NS_LOG=UdpEchoClientApplication=level_all|prefix_func|prefix_time:
            UdpEchoServerApplication=level_all|prefix_func|prefix_time'
```

(type on one line)

This produces a more detailed timestamped log.

You can add more helpful log statements as follows: Open scratch/myfirst.cc in your favorite editor and add the line,

```
NS_LOG_INFO ("Creating Topology");
```

right before the lines,

```
NodeContainer nodes;
nodes.Create (2);
```

Now build the script using waf and clear the NS_LOG variable to turn off the high level of logging we previously enabled:

```
$ ./waf
$ export NS_LOG=
```

Now, if you run the script,

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

you will not see your new message since its associated logging component (FirstScriptExample) has not been enabled. In order to see your message you will have to enable the FirstScriptExample logging component with a level greater than or equal to NS_LOG_INFO. If you just want to see this particular level of logging, you can enable it by,

```
$ export NS_LOG=FirstScriptExample=info
```

If you now run the script you will see your new “Creating Topology” log message,

```
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 (0.404s)
Creating Topology
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.1.2
```

1.1.6 Using the Tracing System

Ascii Tracing

The whole point of simulation is to generate output for further study, and the ns-3 tracing system is a primary mechanism for this. 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.

Let’s just jump right in and add some ASCII tracing output to our scratch/myfirst.cc script. Right before the call to Simulator::Run (), add the following lines of code:

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

This 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

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

You will see some messages from Waf and then “‘build’ finished successfully” with some number of messages from the running program.

When it ran, the program will have created a file named myfirst.tr.

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:

+: An enqueue operation occurred on the device queue;
-: A dequeue operation occurred on the device queue;
d: A packet was dropped, typically because the queue was full;
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 which can be read in Wireshark
The code used to enable pcap tracing is a one-liner:

```
pointToPoint.EnablePcapAll ("myfirst");
```

Go ahead and insert this line of code after the ASCII tracing code we just added to scratch/myfirst.cc. Notice that we only passed the string “myfirst,” and not “myfirst.pcap” or something similar.

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.

Once you have added the line of code to enable pcap tracing, you can run the script in the usual way:

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

If you look at the top level directory of your distribution, you should now see three log files: myfirst.tr is the ASCII trace file we have previously examined. myfirst-0-0.pcap and myfirst-1-0.pcap are the new pcap files we just generated.

You can open these files in wireshark - they look just like real packets!
Many of the questions in the exercises can also be answered through wireshark.

We will not look into more details of these mechanisms as we are going to use an advanced (contributed) module FlowMonitor for statistics.

Congratulations, you are done with the warm-up task of Lab 04!

1.2 Exercise 2: Play with FTP flow (for sliding window, etc)

As has been observed correctly by many of you, there are hardly any point-to-point protocols left that employ sliding windows any more, since RDT is an overkill for the low link error rates of today's guided physical media (fiber, etc). Sliding window is mainly used now at the *transport layer* - i.e. 'end-to-end' at process-to-process communication level. We would still like to study sliding windows at a directly connected layer, so we will study TCP over a direct link, while disabling TCP's advanced mechanism where it dynamically changes its sliding window size. This will essentially achieve the effect of the use of a sliding window protocol at link layer.

The file lab4FTPonly.cc (//insert link) is an implementation of a simple direct link topology in a source node does a 'bulk transfer' over TCP to a sink node.

In this file, we use the Flow Monitor for throughput analysis. For now, you should ignore that code, it will just produce the answers you need.

You should now try to understand this file, it is similar to 'first.cc', and well commented.

Topology:

- One source node (host)
- One destination node (host)
- One point-to-point link in between

The Flow

- An FTP flow ("bulk sender") between hosts 0 and 1
  - start and end time mentioned in the tcl file
  - Maximum number of bytes (file size) to be transferred is

FTP flow is Over TCP

- Uses sliding window protocol between the two ends (0 and 1), over the "logical connection", but since only one link, it's like sliding window on this linnk

- TCP Sliding window size is given using two parameters which should have the same value. Do not worry about the meaning of these parameters. For now they are just the window size.

Run this file as ./waf --run lab4FTPonly

Play with the sliding window size of the ftp flow to see its impact on its throughput (always change BOTH the parameters). “Play with” means you change the window parameter, run ns3, see how it impacts the throughput.

Keep increasing the window size to increase the throughput. Enter throughput obtained for 3 different window sizes. Answer the following questions

1. For the given link data rate, what is the maximum throughput the ftp flow can achieve?
   7757 Kbps
2. What should be the theoretically calculated window size (in bytes) that will achieve this throughput
   20000
3. What is experimentally seen minimum window size at which this throughput was obtained?
   179 KB
4. How does the maximum achieved throughput compare with the raw data rate of the link?
   .9696 times
5. How does changing link delay affect all the above? Write down the parameters and measurements of the experiments you did to study the impact of link delay.
   Throughput obviously decreases...

1.3 Excercise 3: Only CBR Flow

“CBR” stands for Constant Bit Rate. This is the kind of traffic generated by non-coded voice flows (voice is sampled at a fixed rate, a packet assembled of the digitized voice, and it is sent off). This type of traffic is time-sensitive but can do with a few losses hence RDT is not used for such flows. In the Internet stack, they are sent over UDP, which has no ACKs etc. We’ll study this flow to see how it behaves in contrast to a flow under sliding window control.

Study the 3rd file provided. lab3CBRonly.cc . Copy this to the scratch folder, and run waf in the higher directory as

./waf --run lab3CBRonly

Understand the code (very similar to FTP simulator code). Play with the CBR flow parameters and check the throughput and delays

Give the following answers

1. Enter 3 different throughput values (in increasing order), for 3 different sets of flow parameters

463,4623,7760 for CBRdatarate 448,4488,44888

2. What is the maximum throughput this flow can achieve?
   7768.12Kbps
3. How does this compare with the raw data rate of the bottleneck link?
   Around 8 Mbps. So well and good

1.4 Exercise 4: Study impact of CBR on Sliding window flow

We will do this analysis in the next lab.

1.5 FlowMonitor for analysis

(The following verbatim copied with some edits Flowmonitor Webpage)

The Flow Monitor module goal is to provide a flexible system to measure the performance of network protocols. The module uses probes, installed in network nodes, to track the packets exchanged by the nodes, and it will measure a number of parameters. Packets are divided according to the flow they belong to, where each flow is defined according to the probe's characteristics (e.g., for IP, a flow is defined as the packets with the same {protocol, source (IP, port), destination (IP, port)} tuple.

The statistics are collected for each flow can be exported in XML format. Moreover, the user can access the probes directly to request specific stats about each flow.

Scope and Limitations

At the moment, probes and classifiers are available only for IPv4 and IPv6.

IPv4 and IPv6 probes will classify packets in four points:

When a packet is sent (SendOutgoing IPv[4,6] traces)
When a packet is forwarded (UnicastForward IPv[4,6] traces)
When a packet is received (LocalDeliver IPv[4,6] traces)
When a packet is dropped (Drop IPv[4,6] traces)

Since the packets are tracked at IP level, any retransmission caused by L4 protocols (e.g., TCP) will be seen by the probe as a new packet.

A Tag will be added to the packet (ns3::Ipv[4,6]FlowProbeTag). The tag will carry basic packet's data, useful for the packet's classification.

It must be underlined that only L4 (TCP, UDP) packets are, so far, classified. Moreover, only unicast packets will be classified. These limitations may be removed in the future.

The data collected for each flow are:

timeFirstTxPacket: when the first packet in the flow was transmitted;
timeLastTxPacket: when the last packet in the flow was transmitted;
timeFirstRxPacket: when the first packet in the flow was received by an end node;
timeLastRxPacket: when the last packet in the flow was received;
delaySum: the sum of all end-to-end delays for all received packets of the flow;
jitterSum: the sum of all end-to-end delay jitter (delay variation) values for all received packets of the flow, as defined in RFC 3393;
txBytes, txPackets: total number of transmitted bytes / packets for the flow;
rxBytes, rxPackets: total number of received bytes / packets for the flow;
lostPackets: total number of packets that are assumed to be lost (not reported over 10 seconds);
timesForwarded: the number of times a packet has been reportedly forwarded;
delayHistogram, jitterHistogram, packetSizeHistogram: histogram versions for the delay, jitter, and packet sizes, respectively;
packetsDropped, bytesDropped: the number of lost packets and bytes, divided according to the loss reason code (defined in the probe).
It is worth pointing out that the probes measure the packet bytes including IP headers. The L2 headers are not included in the measure.

These stats will be written in XML form upon request (see the Usage section).

The “lost” packets problem

At the end of a simulation, Flow Monitor could report about “lost” packets, i.e., packets that Flow Monitor have lost track of.

It is important to keep in mind that Flow Monitor records the packets statistics by intercepting them at a given network level - let’s say at IP level. When the simulation ends, any packet queued for transmission below the IP level will be considered as lost.

It is strongly suggested to consider this point when using Flow Monitor. The user Stop the Applications before the actual Simulation End time, leaving enough time between the two for the queued packets to be processed. The second method is the suggested one. Usually a few seconds are enough (the exact value depends on the network type).

It is important to stress that “lost” packets could be anywhere in the network, and could count toward the received packets or the dropped ones. Ideally, their number should be zero or a minimal fraction of the other ones, i.e., they should be “statistically irrelevant”.
One important thing is: the ns3::FlowMonitorHelper must be instantiated only once in the main.

Here is an example of code snippet from a file which has two flows - “FTP” and “CBR” for directly using the stats to find overall metrics of the simulation run :

```
 // Flow monitor
    Ptr<FlowMonitor> flowMonitor;
    FlowMonitorHelper flowHelper;
    flowMonitor = flowHelper.InstallAll();
    Simulator::Stop (Seconds (endTime));
    Simulator::Run ();

    Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier> (flowHelper.GetClassifier ());
    std::map<FlowId, FlowMonitor::FlowStats> stats = flowMonitor->GetFlowStats ();

   std::cout << std::endl << "===================== Flow monitor statistics ====================== " << std::endl;
   std::cout << std::endl ;

    for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator iter = stats.begin (); iter != stats.end (); ++iter)
    {
      Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (iter->first);
         std::cout << "Flow ID: " << iter->first << "\nSrc Addr: " << t.sourceAddress << " ----- Dst Addr: " << t.destinationAddress << std::endl;
         std::cout << std::endl ;
         std::cout << "  Tx Bytes\t\t" << iter->second.txBytes << std::endl;
         std::cout << "  Tx Packets\t\t" << iter->second.txPackets << std::endl;
         std::cout << "  Rx Bytes\t\t" << iter->second.rxBytes << std::endl;
         std::cout << "  Rx Packet\t\t" << iter->second.rxPackets << std::endl;
         std::cout << "  Input Load\t\t" << iter->second.txBytes * 8.0 / (iter->second.timeLastTxPacket.GetSeconds () - iter->second.timeFirstTxPacket.GetSeconds ()) / 1024 << " Kbps" << std::endl;
         std::cout << "  Observed Throughput\t" << iter->second.rxBytes * 8.0 / (iter->second.timeLastRxPacket.GetSeconds()-iter->second.timeFirstTxPacket.GetSeconds()) / 1024  << " Kbps" << std::endl;
         std::cout << "  Mean delay\t\t" << iter->second.delaySum.GetSeconds () / iter->second.rxPackets << std::endl;
         std::cout << "  Mean jitter\t\t" << iter->second.jitterSum.GetSeconds () / (iter->second.rxPackets - 1) << std::endl;
        }

    }
    std::cout << std::endl << std::endl ;
    flowMonitor->SerializeToXmlFile("data.flowmon", true, true);
    Simulator::Destroy ();
    NS_LOG_INFO ("Done.");
```