Experiment 4: To create two nodes and simulate the communication between them using NetAnim.

Object Overview

ns-3 is fundamentally a C++ object-based system.By this we mean that new C++ classes(types) can be declared ,defined and subclassed as usual.
Many ns-3 objects inherit from the object base class.These objects have some additional properties that we exploit for organizing the system and improving the memory management of our objects:

- "Metadata" system that links the class name to a lot of meta-information about the object, Including:
  - The base class of the subclass,
  - The set of accessible construction in the subclass,
  - The set of "attributes" of the subclass,
  - Whether each attribute can be set, or is read-only,
  - The allowed range of values for each attribute.

  Reference counting smart pointer implementation, for memory management.

ns-3 objects that use the attribute system derive from either Object or ObjectBase. Most ns-3 object we will discuss derive from Object, but a few that are outside the smart pointer memory management framework derive from ObjectBase.

1.Module Includes

While simulating different devices in a network we try to impart their features to their respective objects which we create in the programs, to do this we need to include all the libraries and modules which help in implementing the device's methods.
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"
```

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 collisons. 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 depending on your configuration. Waf will also automatically generate a module include file to laod 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.

2.Ns3 Namespace

The next line in the first.cc script is 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.

3.Logging

The next line of the script is the following,

```
NS_LOG_COMPONENT_DEFINE(“FirstScriptExample”);
```

The ns-3 logging facility can be used to monitor or debug the progress of simulation programs. Logging output can be enabled by program statements in your main() program or by the use of the NS_LOG environment variable. 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.

4.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 application:

```
LogComponentEnable(“UdpEchoClientApplication”, LOG_LEVEL_INFO);
LogComponentEnable(“UdpEchoClientApplication”, 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 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.

5.Topology Helpers

5.1NodeContainer

keep track of a set of node pointers.

Typically ns-3 helpers operate on more than one node at a time. For example a device helper may want to install devices on a large number of similar nodes. The helper Install methods usually take a NodeContainer as a parameter. NodeContainers hold the multiple Ptr<Node> which are used to refer to the nodes.

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

5.2PointToPointHelper

Build a set of PointToPointNetDevice objects.Normally we eschew multiple inheritance, however, the classes PcapUserHelperForDevice and AsciiTraceUserHelperForDevice are "mixins".
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.SetDevicesAttrubute(“DataRate”,StringValue(“5Mbps”));
PointToPoint.SetChannelAttribute(“Delay”,StringValue(“2ms”));
```

5.3 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 createing, 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 NetDeviceConatainer to hold them just as we used a NodeContainer to hold the nodes we created.
The following two lines of code, will finish configuring the devices and channel.

```
NetDeviceContainer devices;
Devices= PointToPoint.Install(nodes);
```

5.4 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.

5.5 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 10.1.1.22 etc.

6.Applications

Another one of the core abstractions of the ns-3 system is the Application. In this script we use tow 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 UdpEchoServeHelper and UdpEchoClientHelper objects to make our lives easier.

6.1 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));
severapps.start(seconds(1.0));
severapps.stop(seconds(10.0));
```

In the first line of the code the parameter passed is the port number of the client i.e it tells the server from which it receives the packets from. The 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. Application 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 times are set using the ApplicationContainer methods Start and Stop.

6.2 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 manged 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(2.0));
```

In the first line of the 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. Just as in 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).

7. 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.

8. 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. Lets try it. Copy/examples/tutorial/first.cc into the scratch directory after changing back into 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 build successfully.
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
```

Aim: Simulation of simple network with two nodes.

Description: In this we have 2 nodes with point to point connection of which node 0 is acting as client and node 1 is acting as server.

Source Code:

```
#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"
#include "ns3/netanim-module.h"
using namespace ns3;
//NS_LOG_COMPONENT_DEFINE("FirstScriptExample");
int
main (int argc, char *argv[])
{
	Time::SetResolution (Time::NS);
	LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
	LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);
	NodeContainer nodes;
	nodes.Create (2);
	PointToPointHelper pointToPoint;
	pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
	pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
	NetDeviceContainer devices;
```

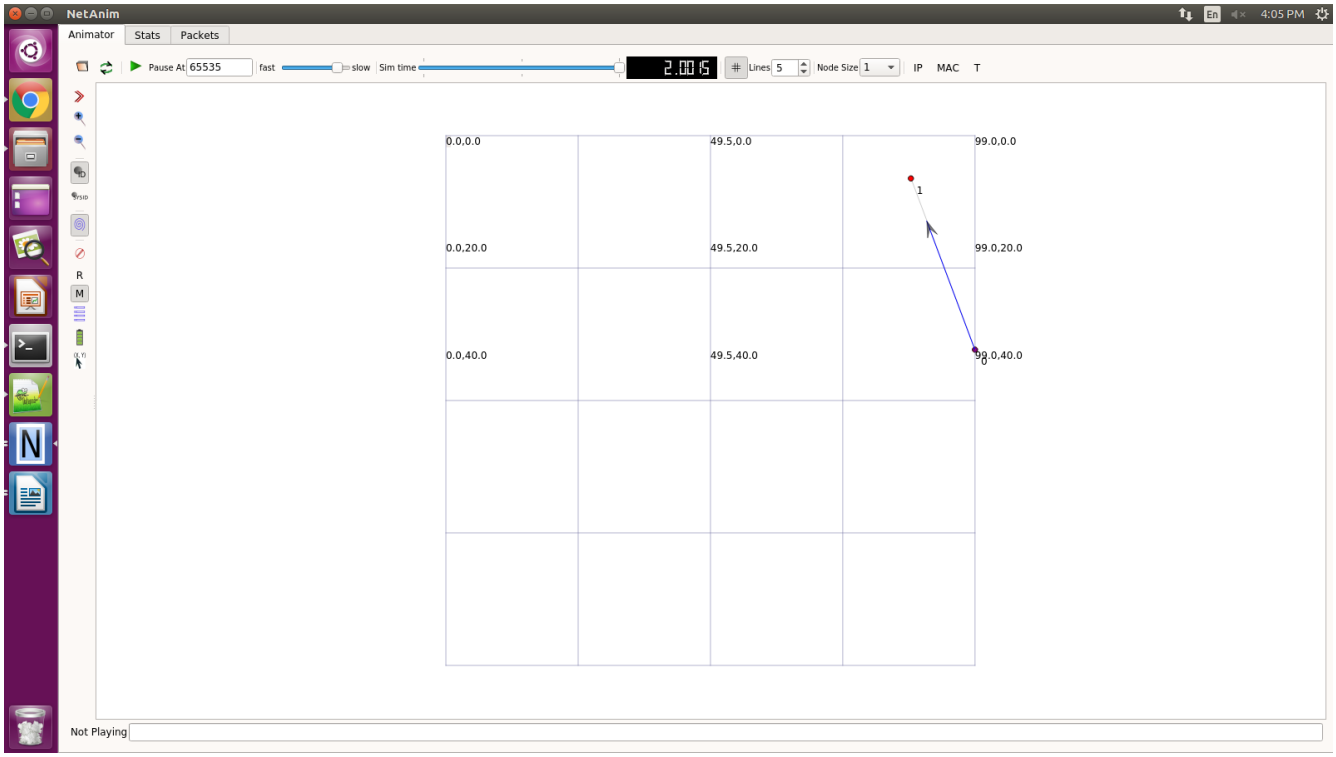
```
        devices = pointToPoint.Install (nodes);
        InternetStackHelper stack;
        stack.Install (nodes);
        Ipv4AddressHelper address;
        address.SetBase ("10.1.1.0", "255.255.255.0");
        Ipv4InterfaceContainer interfaces = address.Assign (devices);
        UdpEchoServerHelper echoServer (9);
        ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));
        serverApps.Start (Seconds (1.0));
        serverApps.Stop (Seconds (10.0));
        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));
        AnimationInterface anim("twonodes.xml");
Simulator::Run ();
        Simulator::Destroy ();
return 0;
}
```

Output:

When we run the command after building the nsallinone-3.28 we use the ./waf --run scratch/twonodes we get the output as

```
root@cbit-OptiPlex-3060:/home/student/Downloads/ns-allinone-3.28/ns-3.28# ./waf --run scratch/twonodes
Waf: Entering directory `/home/student/Downloads/ns-allinone-3.28/ns-3.28/build'
[2305/2702] Compiling scratch/twonodes.cc
[2690/2702] Linking build/scratch/twonodes
Waf: Leaving directory `/home/student/Downloads/ns-allinone-3.28/ns-3.28/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (2.689s)
AnimationInterface WARNING:Node:0 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:1 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:0 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:1 Does not have a mobility model. Use SetConstantPosition if it is stationary
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
```

And we want to see the output through animation in netanim we get output as



Conclusion: Therfore we have created two nodes ,established a connection between them and sent and recieved packets of data.