Lab3 Building Wireless topology

1. Tutorial: Building a wireless network topology

In this section we are going to further expand our knowledge of ns-3 network devices and channels to cover an example of a wireless network. ns-3 provides a set of 802.11 models that attempt to provide an accurate MAC-level implementation of the 802.11 specification and a “not-so-slow” PHY-level model of the 802.11a specification.

Create mythird.cc and add the following code (in green).

The code begins by loading include files just as was done in the second.cc example. There are a couple of new includes corresponding to the Wifi module and the mobility module which we will discuss below.

#include "ns3/core-module.h"

#include "ns3/point-to-point-module.h"

#include "ns3/network-module.h"

#include "ns3/applications-module.h"

#include "ns3/wifi-module.h"

#include "ns3/mobility-module.h"

#include "ns3/csma-module.h"

#include "ns3/internet-module.h"

The network topology illustration follows:

// Default Network Topology

//

// Wifi 10.1.3.0

// AP

// \* \* \* \*

// | | | | 10.1.1.0

// n5 n6 n7 n0 -------------------------- n1 n2 n3 n4

// point-to-point | | | |

// ================

// LAN 10.1.2.0

You can see that we are adding a new network device to the node on the left side of the point-to-point link that becomes the access point for the wireless network. A number of wireless STA nodes are created to fill out the new 10.1.3.0 network as shown on the left side of the illustration.

After the illustration, the ns-3 namespace is used and a logging component is defined. This should all look quite familiar by now.

using namespace ns3;

NS\_LOG\_COMPONENT\_DEFINE ("ThirdScriptExample");

The main program begins just like second.cc by adding some command line parameters for enabling or disabling logging components and for changing the number of devices created.

int

main (int argc, char \*argv[])

{

bool verbose = true;

uint32\_t nCsma = 3;

uint32\_t nWifi = 3;

bool tracing = true;

CommandLine cmd;

cmd.AddValue ("nCsma", "Number of \"extra\" CSMA nodes/devices", nCsma);

cmd.AddValue ("nWifi", "Number of wifi STA devices", nWifi);

cmd.AddValue ("verbose", "Tell echo applications to log if true", verbose);

cmd.AddValue ("tracing", "Enable pcap tracing", tracing);

cmd.Parse (argc,argv);

// Check for valid number of csma or wifi nodes

// 250 should be enough, otherwise IP addresses

// soon become an issue

if (nWifi > 250 || nCsma > 250)

{

std::cout << "Too many wifi or csma nodes, no more than 250 each." << std::endl;

return 1;

}

if (verbose)

{

LogComponentEnable ("UdpEchoClientApplication", LOG\_LEVEL\_INFO);

LogComponentEnable ("UdpEchoServerApplication", LOG\_LEVEL\_INFO);

}

Just as in all of the previous examples, the next step is to create two nodes that we will connect via the point-to-point link.

NodeContainer p2pNodes;

p2pNodes.Create (2);

Next, we see an old friend. We instantiate a PointToPointHelper and set the associated default Attributes so that we create a five megabit per second transmitter on devices created using the helper and a two millisecond delay on channels created by the helper. We then Install the devices on the nodes and the channel between them.

PointToPointHelper pointToPoint;

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

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

NetDeviceContainer p2pDevices;

p2pDevices = pointToPoint.Install (p2pNodes);

Next, we declare another NodeContainer to hold the nodes that will be part of the bus (CSMA) network.

NodeContainer csmaNodes;

csmaNodes.Add (p2pNodes.Get (1));

csmaNodes.Create (nCsma);

The next line of code Gets the first node (as in having an index of one) from the point-to-point node container and adds it to the container of nodes that will get CSMA devices. The node in question is going to end up with a point-to-point device and a CSMA device. We then create a number of “extra” nodes that compose the remainder of the CSMA network.

We then instantiate a CsmaHelper and set its Attributes as we did in the previous example. We create a

NetDeviceContainer to keep track of the created CSMA net devices and then we Install CSMA devices on the selected nodes.

CsmaHelper csma;

csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));

csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));

NetDeviceContainer csmaDevices;

csmaDevices = csma.Install (csmaNodes);

Next, we are going to create the nodes that will be part of the Wifi network. We are going to create a number of “station” nodes as specified by the command line argument, and we are going to use the “leftmost” node of the point-to-point link as the node for the access point.

NodeContainer wifiStaNodes;

wifiStaNodes.Create (nWifi);

NodeContainer wifiApNode = p2pNodes.Get (0);

The next bit of code constructs the wifi devices and the interconnection channel between these wifi nodes. First, we configure the PHY and channel helpers:

YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();

YansWifiPhyHelper phy = YansWifiPhyHelper::Default ();

For simplicity, this code uses the default PHY layer configuration and channel models which are documented in the API doxygen documentation for the YansWifiChannelHelper::Default and YansWifiPhyHelper::Default methods. Once these objects are created, we create a channel object and associate it to our PHY layer object manager to make sure that all the PHY layer objects created by the YansWifiPhyHelper share the same underlying channel, that is, they share the same wireless medium and can communication and interfere:

phy.SetChannel (channel.Create ());

Once the PHY helper is configured, we can focus on the MAC layer. Here we choose to work with non-Qos MACs so we use a NqosWifiMacHelper object to set MAC parameters.

WifiHelper wifi;

wifi.SetRemoteStationManager ("ns3::AarfWifiManager");

WifiMacHelper mac;

The SetRemoteStationManager method tells the helper the type of rate control algorithm to use. Here, it is asking the helper to use the AARF algorithm— details are, of course, available in Doxygen.

Next, we configure the type of MAC, the SSID of the infrastructure network we want to setup and make sure that our stations don’t perform active probing:

Ssid ssid = Ssid ("ns-3-ssid");

mac.SetType ("ns3::StaWifiMac",

"Ssid", SsidValue (ssid),

"ActiveProbing", BooleanValue (false));

This code first creates an 802.11 service set identifier (SSID) object that will be used to set the value of the “Ssid” Attribute of the MAC layer implementation. The particular kind of MAC layer that will be created by the helper is specified by Attribute as being of the “ns3::StaWifiMac” type. The use of NqosWifiMacHelper will ensure that the “QosSupported” Attribute for created MAC objects is set false. The combination of these two configurations means that the MAC instance next created will be a non-QoS non-AP station (STA) in an infrastructure BSS (i.e., a BSS with an AP). Finally, the “ActiveProbing” Attribute is set to false. This means that probe requests will not be sent by MACs created by this helper.

Once all the station-specific parameters are fully configured, both at the MAC and PHY layers, we can invoke our now-familiar Install method to create the wifi devices of these stations:

NetDeviceContainer staDevices;

staDevices = wifi.Install (phy, mac, wifiStaNodes);

We have configured Wifi for all of our STA nodes, and now we need to configure the AP (access point) node. We begin this process by changing the default Attributes of the WifiMacHelper to reflect the requirements of the AP.

mac.SetType ("ns3::ApWifiMac",

"Ssid", SsidValue (ssid));

In this case, the WifiMacHelper is going to create MAC layers of the “ns3::ApWifiMac”, the latter specifying that a MAC instance configured as an AP should be created. We do not change the default setting of “QosSupported” Attribute, so it remains false - disabling 802.11e/WMM-style QoS support at created APs.

The next lines create the single AP which shares the same set of PHY-level Attributes (and channel) as the stations:

NetDeviceContainer apDevices;

apDevices = wifi.Install (phy, mac, wifiApNode);

Now, we are going to add mobility models. We want the STA nodes to be mobile, wandering around inside a bounding box, and we want to make the AP node stationary. We use the MobilityHelper to make this easy for us. First, we instantiate a MobilityHelper object and set some Attributes controlling the “position allocator” functionality.

MobilityHelper mobility;

mobility.SetPositionAllocator ("ns3::GridPositionAllocator",

"MinX", DoubleValue (0.0),

"MinY", DoubleValue (0.0),

"DeltaX", DoubleValue (5.0),

"DeltaY", DoubleValue (10.0),

"GridWidth", UintegerValue (3),

"LayoutType", StringValue ("RowFirst"));

This code tells the mobility helper to use a two-dimensional grid to initially place the STA nodes. Feel free to explore the Doxygen for class ns3::GridPositionAllocator to see exactly what is being done.

We have arranged our nodes on an initial grid, but now we need to tell them how to move. We choose the RandomWalk2dMobilityModel which has the nodes move in a random direction at a random speed around inside a bounding box.

mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",

"Bounds", RectangleValue (Rectangle (-50, 50, -50, 50)));

We now tell the MobilityHelper to install the mobility models on the STA nodes.

mobility.Install (wifiStaNodes);

We want the access point to remain in a fixed position during the simulation. We accomplish this by setting the mobility model for this node to be the ns3::ConstantPositionMobilityModel:

mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");

mobility.Install (wifiApNode);

We now have our nodes, devices and channels created, and mobility models chosen for theWifi nodes, but we have no protocol stacks present. Just as we have done previously many times, we will use the InternetStackHelper to install these stacks.

InternetStackHelper stack;

stack.Install (csmaNodes);

stack.Install (wifiApNode);

stack.Install (wifiStaNodes);

Just as in the second.cc example script, we are going to use the Ipv4AddressHelper to assign IP addresses to our device interfaces. First we use the network 10.1.1.0 to create the two addresses needed for our two point-to-point devices. Then we use network 10.1.2.0 to assign addresses to the CSMA network and then we assign addresses from network 10.1.3.0 to both the STA devices and the AP on the wireless network.

Ipv4AddressHelper address;

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

Ipv4InterfaceContainer p2pInterfaces;

p2pInterfaces = address.Assign (p2pDevices);

address.SetBase ("10.1.2.0", "255.255.255.0");

Ipv4InterfaceContainer csmaInterfaces;

csmaInterfaces = address.Assign (csmaDevices);

address.SetBase ("10.1.3.0", "255.255.255.0");

address.Assign (staDevices);

address.Assign (apDevices);

We put the echo server on the “rightmost” node in the illustration at the start of the file. We have done this before.

UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (csmaNodes.Get (nCsma));

serverApps.Start (Seconds (1.0));

serverApps.Stop (Seconds (10.0));

And we put the echo client on the last STA node we created, pointing it to the server on the CSMA network. We have also seen similar operations before.

UdpEchoClientHelper echoClient (csmaInterfaces.GetAddress (nCsma), 9);

echoClient.SetAttribute ("MaxPackets", UintegerValue (1));

echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));

echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer clientApps =

echoClient.Install (wifiStaNodes.Get (nWifi - 1));

clientApps.Start (Seconds (2.0));

clientApps.Stop (Seconds (10.0));

Since we have built an internetwork here, we need to enable internetwork routing just as we did in the second.cc example script.

Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

One thing that can surprise some users is the fact that the simulation we just created will never “naturally” stop. This is because we asked the wireless access point to generate beacons. It will generate beacons forever, and this will result in simulator events being scheduled into the future indefinitely, so we must tell the simulator to stop even though it may have beacon generation events scheduled. The following line of code tells the simulator to stop so that we don’t simulate beacons forever and enter what is essentially an endless loop.

Simulator::Stop (Seconds (10.0));

We create just enough tracing to cover all three networks:

pointToPoint.EnablePcapAll ("mythird");

phy.EnablePcap ("mythird", apDevices.Get (0));

csma.EnablePcap ("mythird", csmaDevices.Get (0), true);

These three lines of code will start pcap tracing on both of the point-to-point nodes that serves as our backbone, will start a promiscuous (monitor) mode trace on the Wifi network, and will start a promiscuous trace on the CSMA network. This will let us see all of the traffic with a minimum number of trace files.

Finally, we actually run the simulation, clean up and then exit the program.

Simulator::Run ();

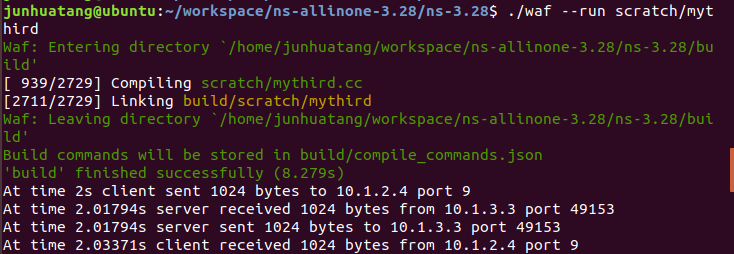
Simulator::Destroy ();

return 0;

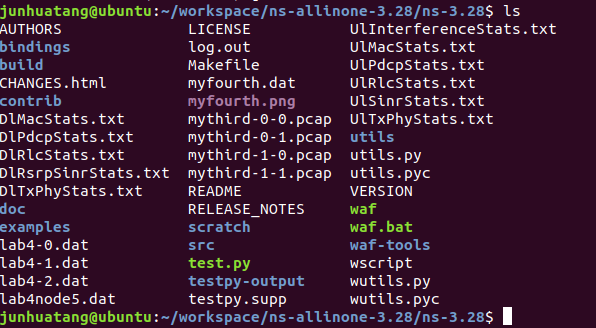
}

$ ./waf

$ ./waf --run scratch/mythird



$ls

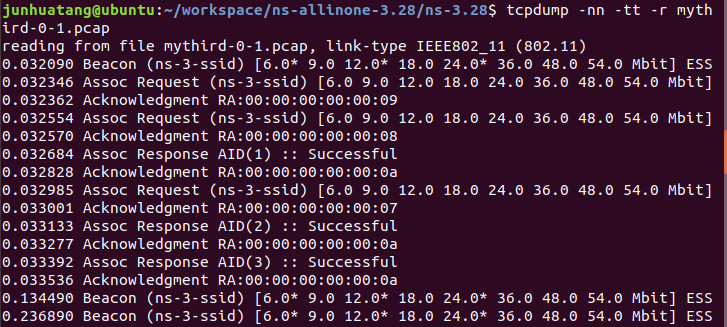


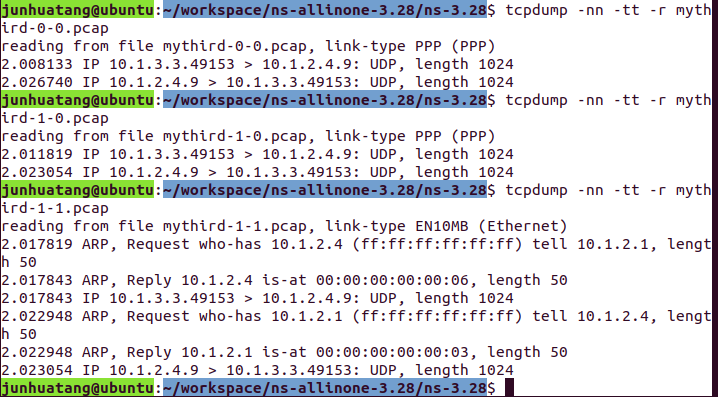
You will find four trace files from this simulation, two from node zero and two from node one:

mythird-0-0.pcap, mythird-0-1.pcap, mythird-1-0.pcap, mythird-1-1.pcap

The file “mythird-0-0.pcap” corresponds to the point-to-point device on node zero – the left side of the “backbone”. The file “mythird-1-0.pcap” corresponds to the point-to-point device on node one – the right side of the “backbone”. The file “mythird-0-1.pcap” will be the promiscuous (monitor mode) trace from the Wifi network and the file “mythird-1-1.pcap” will be the promiscuous trace from the CSMA network.

$ tcpdump -nn -tt -r mythird-0-1.pcap





1. Exercise: Create three access point A, B and C. Wire (Connect) A-B, B-C using point to point link (100Mbps, 2ms). Create three client nodes (mobile stations) for access points A and C, respectively, and allow them to move according to Random Walk mobility model. Install UdpEchoClient on a node associated with access point A, and install UdpEchoServer on a node connected to C. Let UdpEchoClient send a total number of 2 packets to the UdpEchoServer, with an Interval of 1 second, and PacketSize of 1024 bytes. Generate the traces using packet capture.

Wifi 10.1.3.0

\* \* \*

| | |

n3 n4 n5 p2p (100Mbps, 2ms) p2p(100Mbps, 2ms)

A (n0)------------------------B(n1)---------------------------C(n2)

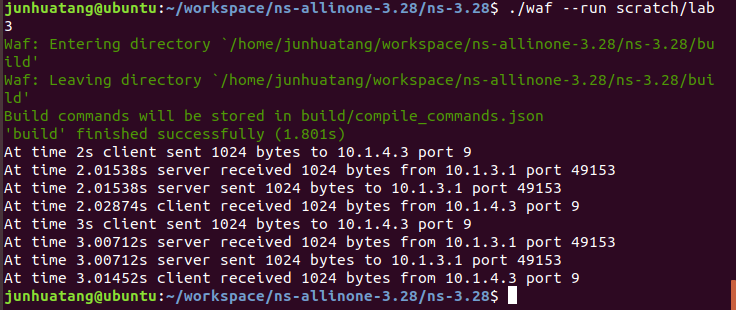
(AP) 10.1.1.0 1 0.1.2.0 (AP)

n6 n7 n8

| | |

\* \* \*

Wifi 10.1.4.0



Turn in:

1. The source code of the above exercise: lab3.cc
2. lab3-report:
3. screenshots of simulation results
4. traces at access point A
5. traces at UdpEchoClient node and UdpEchoServer node
6. a brief analysis of the traces