ns-3 Cheat Book

Adil Alsuhaim (aalsuha@clemson.edu)

December 4, 2020

Contents

Preface

ns-3 is an open-source discrete network simulator written in C++.

Chapter 1

Getting Started

1.1 Installation

You will need to download C++ sources and build them on your computer. ns-3 also use tools that use python3. Therefore, your system must satisfy some prerequisites.

1.1.1 ns-3 prerequisites

A Ubuntu/Debian

These instructions are for Ubuntu/Debian based Linux installation that use the apt package manager. For other flavors of Linux, refer to ns-3 installation instructions page https://www.nsnam.org/wiki/Installation.

- **git** This is not mandatory, as you can simply download a compressed file containing the ns-3 sources. Previous versions of ns-3 used mercurial (which is used with the executable hg). To install both on Ubuntu, run

```
sudo apt-get install git mercurial
```

- **C++ & python** Your system need to be able to compile and build C++ source files. You use any C++ compiler, but keep in mind that ns-3 is tested with gcc and clang. You also need to install python3 since the ns-3 build tool use python3.

```
sudo apt-get install g++ python3
```

- **More python components.** ns-3 use an optional live simulation visualization tool written in python. This tool uses python bindings to ns-3 API. Install the following:

```
sudo apt-get install python3-dev pkg-config sqlite3 python3-setuptools
sudo apt-get install gir1.2-goocanvas-2.0 python-gi python-gi-cairo python-pygraphviz python3-gi python3
    -gi-cairo python3-pygraphviz gir1.2-gtk-3.0 ipython ipython3
```

- **Qt for NetAnim.** ns-3 has an off-line visualization tool that works with an XML trace file that can be generated by the simulation. This tool is written in C++ and uses Qt libraries which provide GUI functionalities. The current version of NetAnim is 3.108, and uses Qt5. Previous versions of NetAnim used Qt4, and there are backward compatibility issues between Qt4 & Qt5. To install Qt5 on Ubuntu

```
sudo apt-get install qt5-default
```

B macOS

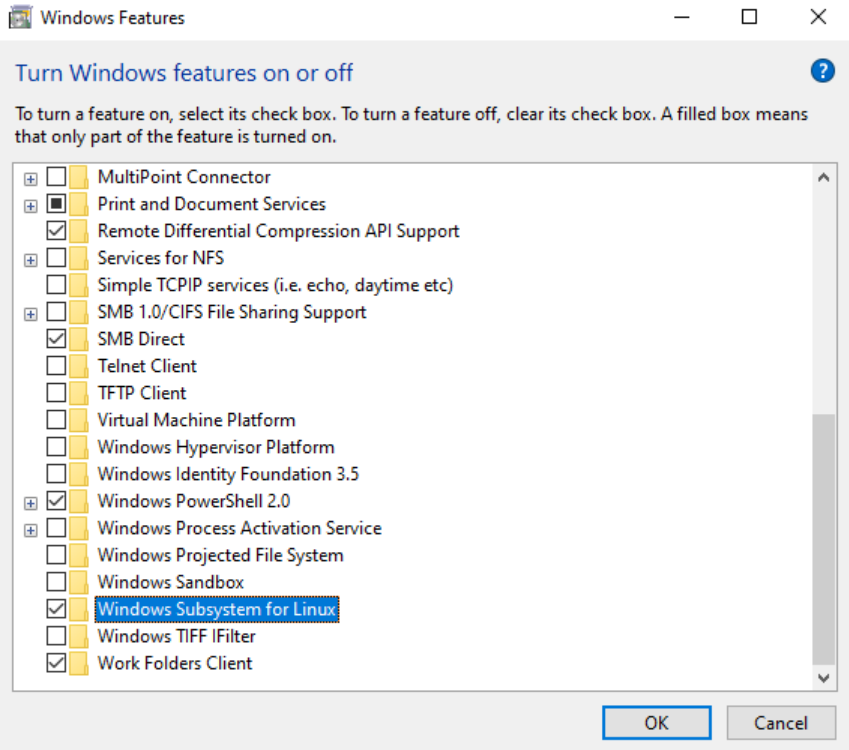
If you hate working with virtual machines like me, you'll be pleased to know that I also use a MacBook Pro, and I install `ns-3` directly to it. `macOS` does not have a built-in package manager, but you can use `HomeBrew` and `MacPorts` to install dependencies. Once the dependencies are met, the process of installing `ns-3` is the same.

Reach out to me if you want to install `ns-3` on `macOS`. I'll update this with instruction when I have the time.

C Windows 10

Windows 10 is not natively supported. You will have to run it on a virtual machine. However, I suggest using "Windows Subsystem for Linux" (WSL), which is a tool that allows Windows to have a Unix shell that can also access your Windows files.

You need to go to Start menu, and find "Turn Windows features on or off", and check "Windows Subsystem for Linux".



This will perform some updates to Windows, and require a reboot. After the restart go to "*Microsoft Store*" and search for "Linux". You can install the latest version of Ubuntu, which requires that you have enabled (WSL) on your Windows.

Once installation of Ubuntu is done, you can start the Ubuntu bash terminal like you start any newly installed Windows application. Now that you are at the Linux subsystem, use `apt` commands to install the prerequisite packages.

1.1.2 Installation

- The most up-to-date installation instruction is on `https://www.nsnam.org/wiki/Installation`. The most recent versions of `ns3` are now hosted on GitLab, and use `Python3` instead of `Python2`. For the record, `Python2` has reached its End-Of-Life beginning with the year 2020. `Python3` is not fully backward compatible with `Python2`, so if you get unexpected errors when using `bake` tool be aware that it might be trying to use `/usr/bin/python` which might be a symbolic link to `Python2`. You can check the symbolic link by running:

```
ls -l $(which python)
```

- You will obtain the most recent version of `bake` from `https://gitlab.com/nsnam/bake` as shown in the instruction page by using `git`

```
git clone https://gitlab.com/nsnam/bake
```

You can run `./bake.py check` to check for installation prerequisites. As of November 2020, the latest `ns3` version is 3.32, you can configure `bake` to download the latest version with the command

```
./bake.py configure -e ns-3.32
```

This should create a file named `bakefile.xml` that is used by `bake` to build the version that you specified. It also enables `netanim` by default. `netanim` is an offline visualization tool, meaning that your simulations would create an `xml` file that can be later visualized by `netanim`. `netanim` is optional, and if you want it, you need to install `Qt5`.

- Please note that older versions of `bake` that you get from `code.nsnam.org` will not work with versions newer than 3.29.
- The `ns-3-dev` version is the most recent code, but we want to avoid installing that version. This is because it keeps changing, and we want to avoid having inconsistencies between versions on your machine, and versions on a colleague's machine that you might be working with. Typically, working with `ns3` versions that are slightly apart does not cause issues, and I think most issues come from using different `C++` compilers. For example, `gcc5` versus `gcc8` or `clang`.
- **PyViz** is an on-line visualization tool that is written in `Python`. Note that the instructions for `ns-3.28` and earlier use `GTK+ 2`, and versions after that use `GTK+ 3`. Install the dependencies that matches the `ns3` version you are working with.
  - In order to have `PyViz` work, Python Binding must be enabled. A common item that users forget to install for that is `python3-dev`, so install that with `apt-get install python3-dev`
  - To make sure that you have `PyViz`, `./waf configure` should show that `PyViz Visualizer` is enabled.

1.1.3 Configuration & First Run

After installing `ns3`, programs are run from the `ns3` root directory, let's call it `$NS3_ROOT_DIR`. The directory should contain directories named `bindings`, `build`, `contrib`, `doc`, `examples`, `scratch`, `src`, `utils`, and `waf-tools`, and files named `wscript`, `waf`, and `Makefile`. If you installed `ns-3.32` it should look like this

```
ns-3.32
├── src
├── build
├── contrib
├── ...
├── examples
├── scratch
├── wscript
├── waf
└── ...
```

- Before you run `ns3`, you need to first perform `./waf configure`. The `configure` option allows you to change configurations such as specifying a specific `C++` compiler, standard and flags. The `Makefile` contains a rule `configure`, which basically enables examples and tests you can do:

```
./waf configure --enable-examples --enable-tests
```

which is equivalent to running

```
make configure
```

I personally edit the `Makefile` and add a `--disable-werror` flag to disable treating warnings as errors.

- If you enabled examples, we can run the first tutorial example in `examples/tutorials/first.cc` as follows:

```
./waf --run examples/tutorial/first
```

- ***Important Note.*** All C++ files in ns-3 have the extension .cc

1.1.4 Your first script

The scratch directory is where your projects should be. I personally prefer having nothing but directories under scratch since I work with multiple projects. This is cleaner than having scattered files under scratch that belong to multiple projects. Every directory represent a simulation project, and every directory must at least contain one file that has a main function, or ./waf will fail with a message that says undefined reference to 'main'. For example, if I work on three different projects named WifiExample, VANETLossExample and UdpExample, my scratch folder would have three sub-folders as follows

```
ns-3.32
├── waf
├── src
├── build
├── src
├── ..
├── scratch
│   ├── WifiExample
│   │   └── my-wifi-example.cc
│   ├── VANETLossExample
│   │   ├── loss-monitor-application.h
│   │   ├── loss-monitor-application.cc
│   │   └── main-loss-program.cc
│   └── UdpExample
│       └── udp-main.cc
└── ...
```

- Note that every sub-directory is treated as an ns-3 program, and must contain one .cc file that has main function or the build will fail.
- You can run a project by using its directory name under scratch. To run UdpExample, then you should run it with:

```
./waf --run UdpExample
```

- If you want to run it with PyViz, then pass –vis

```
./waf --run UdpExample --vis
```

  However, this requires that you use CommandLine interface as described in 3.3, where I discuss passing command-line arguments to ns-3 programs.

- Let's create a folder under scratch and call it HelloWorld and in it, we'll create a file name hello-main.cc. The project simply prints "Hello World" every second for 10 seconds with a timestamp

```
#include "ns3/core-module.h"
using namespace ns3;
void HelloPrinter ()
{
  std::cout <<"At " << Now().GetSeconds() <<  " : Hello World" << std::endl;
  Simulator::Schedule (Seconds(1), &HelloPrinter);
}

int main (int argc, char *argv[])
{
  CommandLine cmd;
  cmd.Parse (argc, argv);
  //Schedule first call to HelloPrinter
  Simulator::Schedule (Seconds(0), &HelloPrinter);
  //Schedule simulator stop time after 10 seconds of starting.
  Simulator::Stop (Seconds (10));
```

```
  Simulator::Run ();
  Simulator::Destroy();
}
```

and run your code by using the folder name

```
./waf --run HelloWorld
```

- You are good to go!

1.2 Basics

ns3 network simulator is written in C++, and uses scripts written in Python to compile & run code. You write your simulations in C++, and use the waf build tool (written in Python) to build and to run the code. It is also possible to write simulation scripts in Python but you since most users use C++, you might not be able to get help easily from other users.

1.2.1 Simulation Components

To understand how ns3 simulation work, we need to understand how a simulation is built. The main components we need to get started with are:

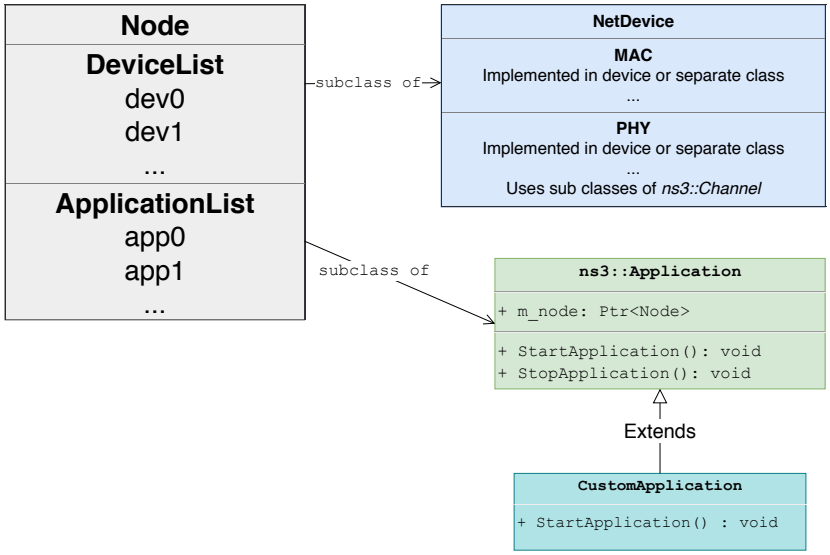
- **Nodes.** A node represents a network host. It is defined as a type named ns3::Node. When a node is created, it is created without a position. A node's position is only needed if it uses a wireless communication device since that's how signal propagation is calculated.

  The main component of a nodes are:

  - **Net devices.** A node can contain a list of net devices (as a subclass of ns3::NetDevice). Net devices have access to the pointer to their node, meaning if you have a pointer to an object of a net device, you can access the node in which they are contained by using GetNode() function.

    Net devices can contain the implementation of PHY & MAC within the same class (such as CsmaNetDevice) or they may have PHY & MAC implemented in separate classes (as in WifiNetDevice)

  - **A list of applications.** A node can contain a list of application (as a subclass of ns3::Application). Users can set application's stop and start times. Similar to net devices, applications can access their node with GetNode, and so they can also access the node's net devices.



- **Channels.** In order to send data between devices, there need to be a channel object (as a subclass of ns3::Channel). Net devices on the same *broadcast domain* will share one channel object. For example, if we have an Ethernet network (Ethernet devices are implemented in ns3::CsmaNetDevice), then all devices on the same network will share one ns3::CsmaChannel object.

  We can specify channel properties such as propagation delay, and data rate. For WiFi simulation, all nodes share one channel object of type ns3::YansWifiChannel, and propagation loss & delay is specified by separate classes such as ns3::LogDistancePropagationLossModel and ns3::ConstantSpeedPropagationDelayModel, and this requires that nodes have a MobilityModel installed to them to specify the wireless signal loss and delay based on the distance between them.

- **Packets.** Data is exchange with objects of type Packet. In ns-3, IP segments & data-link frames are also packets. Headers can be added and removed from packet, effectively changing its size. You can also attach some

information to a packet with the use of packet tags, which does not change the content of the packet. I will talk about packet tags later in this document.

- **Events.** If you want something to happen after a certain amount of time, you need to schedule an event to run a function. This is done by using the function's pointer (using the & symbol), and the parameters to pass to that function (if any). Consider the following:

```
void DoSomething (double x, double y)
{
  std::cout << Now().GetSeconds() << " : x=" << x << " y=" << y << std::endl;
}
...
int main (int args, char* argv[])
{
  ...
  Simulator::Schedule (Seconds (3), &DoSomething, x, y);
  ...
}
```

This is how we can schedule to run the function `DoSomething` to run after 3 seconds. This also passes the arguments `x` and `y`. Noting that this is for a simple case, where we are only dealing with functions not contained within a `C++` class.

You may find yourself needing to run some code periodically. For example, you may want to periodically print some simulation information or write to a file every 1 second. It is possible to have a function that schedules a call to itself every 1 second, and do not forget to schedule a first call to the function as follows

```
void PrintTheTime ()
{
  std::cout << "Simulation time is: " << Now().GetSeconds() << std::endl;

  //Schedule the next call after 1 second from now.
  Simulator::Schedule (Seconds (1), &PrintTheTime);
}
...
int main (int args, char* argv[])
{
  ...
  //schedule the first call after 1 second from the start of the simulation
  Simulator::Schedule (Seconds (1), &PrintTheTime);
  ...
}
```

If the function to invoke is in another class, then use `&ClassName::FunctionName` and a pointer to an object of that class. Let us say we have a class called `MyClass` that has a function `DoSomething` that takes two parameters. Let us also say that the `DoSomething` will schedule a call to itself every 1 second. The following code snippet of the `C++` class and a call from main function will do just that:

```
class MyClass
{
  public:
    void DoSomething (double x, double y)
    {
      std::cout << "Time: " << Now().GetSeconds() << " x=" << x << " and y=" << y << std::endl;
      //replace x by x-1, and y by y-2
      Simulator::Schedule (Seconds (1), &MyClass::DoSomething, this, x-1, y-2);
    }
};
//...
int main (int argc, char *argv[])
{
  //...
  MyClass *obj = new MyClass();
  Simulator::Schedule (Seconds (1.0), &MyClass::DoSomething, obj, 10, 20);
  Simulator::Schedule (Seconds (1.0), &MyClass::DoSomething, obj, 55, 88);
  //...
}
```

Notice that within the C++ class, we used the keyword `this` refering to *this* instance of the object, instead of the object name.

1.2.2 Smart Pointers and `ns3::Object`

In C++, we can create a pointer to an object with by using the * symbol and the `new` keyword as in the previous section. ns-3 implements a smart pointer concept, provided that the C++ classes are sub-class of `ns3::Object`

You might have noticed we used `Ptr` with a class template. This is how ns-3 implements smart pointers (similar to `boost`). Most classes in ns-3 use `Ptr<T>`, where `T` is the type name, to create pointers to objects. There are functionalities associated with this, such as type checking, casting, and calling constructors.

A Object Creation

We can create object instances of an ns-3 class that is a descendant of `ns3::Object` by using the `CreateObject` function. For example, to create a single node (`ns3::Node`) we do

```
Ptr<Node> some_node = CreateObject <Node> ();
```

which effectively calls the default, parameterless constructor of the class `Node`. Keep in mind that we usually prefer to create nodes with `NodeContainer` as follows:

```
NodeContainer nodes;
nodes.Create (4); //Creates 4 nodes
Ptr<Node> n0 = nodes.Get (0); //pointer to node 0
```

Some ns-3 classes are subclasses of `ns3::SimpleRefCount<T>`, not `ns3::Object`. A class that you might use in this category is the `Packet` class, and creating objects of such type is done using the `Create` function.

```
//Create a zero-payload packet
Ptr <Packet> p1 = Create <Packet> ();
//Create a packet with a payload of size 100 bytes (all zeros)
Ptr <Packet> p2 = Create <Packet> (100);
```

B Type checking & casting

C++ is an object-oriented programming language that has inheritance of classes. In ns-3, we have many abstract classes that are extended throughout ns-3. For example, `ns3::NetDevice` is an abstract class because it has a pure virtual function `Send`. Some of the subclasses of `NetDevice` are `WifiNetDevice`, `CsmaNetDevice` and many others. You will see in the examples that we have a `NetDeviceContainer`, which contains pointers to objects of types extending `NetDevice`, and `ApplicationContainer` which contains pointers to objects of types extending the `ns3::Application` class.

You can examine the run-time type of an ns-3 object by using the function `GetInstanceTypeId()`, which returns a value of type `ns3::TypeId`. Also, most ns-3 classes are designed with a `static` function that returns the `TypeId` of a class.

Recall that nodes in ns-3 can have multiple net devices. The following code iterates through the net devices of a node, and perform type casting on them.

```
//NodeList is a global list of all simulation nodes.
Ptr<Node> node = NodeList::GetNode (0);
for (uint32_t i=0; i<node->GetNDevices(); i++)
{
  NetDevice dev = node->GetDevice (i);
  if ( dev->GetInstanceTypeId() == WifiNetDevice::GetTypeId() )
  {
    Ptr<WifiNetDevice> wifi_dev = DynamicCast <WifiNetDevice> (dev);
    //Maybe do something with it
  }
  else if (dev->GetInstanceTypeId() == CsmaNetDevice::GetTypeId())
  {
```

```
    Ptr<CsmaNetDevice> wifi_dev = DynamicCast <CsmaNetDevice> (dev);
    //Maybe do something with it.
  }
}
```

Similar casting can be done with objects in `ApplicationContainer`, as `ns-3` applications are subclasses of `ns3::Application`.

```
ApplicationContainer apps = //some code;
Ptr<UdpEchoServer> server0 = DynamicCast <UdpEchoServer> (apps.Get(0));
```

If you need to create a `C++` class that is compatible with `ns-3` smart pointer functionalities, you need to extend the `ns3::Object` and create the functions `GetInstanceTypeId` and `GetTypeId`. I have more on this later in this document when I talk about creating your own modules.

C Object Aggregation

In `ns-3`, an object of a type that is a subclass of `ns3::Object` can be associated with another object that is a subclass of `ns3::Object` using the `AggregateObject`. You can only aggregate a single object of a given type to another object. When two objects are aggregated, you can use the function `GetObject` on one object to retrieve a pointer to the other.

For example if you have two classes `ClassOne` and `ClassTwo` that are both subclasses of `ns3::Object`, you can perform the following:

```
Ptr <ClassOne> obj_one = CreateObject <ClassOne> ();
Ptr <ClassTwo> obj_two = CreateObject <ClassTwo> ();

obj_one->AggregateObject (obj_two); //Now the two objects are associated
//...
Ptr <ClassTwo> s1 = obj_one->GetObject <ClassTwo> (); //s1 points to the same address as obj_two
Ptr <ClassOne> s2 = obj_two->GetObject <ClassOne> (); //s1 points to the same address as obj_one

//The following will cause a runtime error because you can't aggregate two objects of the same type
Ptr <ClassTwo> tmp_two = CreateObject <ClassTwo> ();
obj_one->AggregateObject (tmp_two); // error!
```

Object aggregate is used to implement *mobility* in `ns-3`. Nodes created in `ns-3` have no mobility information, so no position or velocity. Nodes without mobility can only be used in wired communication because it does not need position information to determine reception. You do not need to use `AggregateObject` yourself to give nodes position, but you may need to use `GetObject` to determine the mobility information of a node.

```
//...
#include "ns3/mobility-module.h"
//...
NodeContainer nodes;
nodes.Create (4);

MobilityHelper mob;
mob.SetMobilityModel ("ns3::ConstantPositionMobilityModel"); //stationary nodes
mob.Install (nodes); // Perform aggregation between every Node and ConstantPositionMobilityModel

//So far, all nodes are positioned at 0,0,0. Let's change the position of node 0
Ptr <Node> n0 = nodes.Get (0);
//the following works because MobilityModel is a subclass of ConstantPositionMobilityModel
Ptr <MobilityModel> mob_n0 = n0->GetObject <MobilityModel> ();
Vector previous_pos = mob_n0->GetPosition ();
//Set the position to 10,15,0. The unit is meters
mob_n0->SetPosition ( Vector (10,15,0));
std::cout << "Node was at " << previous_pos << " now in " << mob_n0->GetPosition () << std::endl;
```

This is convenient because all simulation nodes are contained in the global `NodeList` container, and so you can inquire about nodes positions anywhere in your code. In addition, when you become more adept with `ns-3`, you may aggregate objects of your own user-defined classes to nodes, and have that information available anywhere in the code.

1.2.3 ns-3 modules

An ns3 module consists of C++ sources .cc and headers .h, and a Python script stored in a file named wscript (with no file extension). The wscript file is used by ns3's waf tool to build the module. There are many wscript files in ns-3.

1. **$NS3_ROOT_DIR/wscript** which is the main configuration of ns3, it will look into **/src**, and **/contrib** directories for other wscript files. You should not modify this file
2. **$NS3_ROOT_DIR/src/wscript** this file should not be modified. It will go through all subdirectory under src, and run the wscript in each directory.
3. **$NS3_ROOT_DIR/contrib/wscript** Similar to the previous one, but this is for contributed modules.
4. **$NS3_ROOT_DIR/src/module_name/wscript** Every module named module_name has its own wscript file that points to all the module's source files. If the module is written or modified by you, this file will need to be modified to include newly added source & header files. There is also a similar wscript under contrib/module_name for contributed modules.

   $NS3_ROOT_DIR/src/module_name/examples/wscript If the module contains examples, and ns3 was configured with -enable-examples option, this will be used to build the examples. The same goes for contributed modules in contrib directory.

Let us look at the ns3 module named point-to-point and the files under it for example:

```
point-to-point
├── wscript
├── bindings
│   ├── callbacks_list.py
│   └── ...
├── examples
│   ├── wscript
│   └── main-attribute-value.cc
├── helper
│   ├── point-to-point-helper.cc
│   └── point-to-point-helper.h
├── model
│   ├── point-to-point-channel.cc
│   └── ..
└── test
    ├── examples-to-run.py
    └── point-to-point-test.cc
```

The wscript file under src/point-to-point contains the relative paths to the source & header files.

It is optional to break down your sources into directories like that, and you can choose any name for your subdirctory names, however, examples is different as it would contain examples with main function, and wscript files are typically configured to build them if ns-3 is configured with --enable-examples option.

1.3 Tutorial Examples

There are many tutorial examples in `examples/tutorial` that you can use to learn `ns-3`. You can copy a file over to a folder you create under `scratch` and then modify it there.

```
mkdir scratch/MyExample
cp examples/tutorial/first.cc scratch/MyExample/first-modified.cc
```

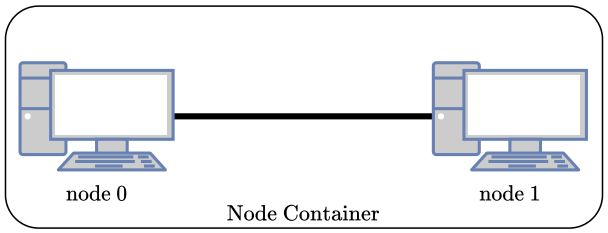
You can then modify the code in `scratch/MyExample/first-modified.cc` and then run it with

```
./waf --run MyExample
```

In addition, every `ns-3` module under `src` directory contains an `examples` folder. You can do the same process of copying a file to your scratch folder and modifying it over there.

1.3.1 Examining `first.cc`

The tutorial example in `examples/tutorial/first.cc` involves two nodes with a point-to-point connection between them. Every node will have a net device of type `PointToPointNetDevice`.



We create the two nodes using `NodeContainer`, and we make the link between them by using `PointToPointHelper`

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

PointToPointHelper p2p;
p2p.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
p2p.SetChannelAttribute ("Delay", StringValue ("2ms"));
NetDeviceContainer p2p_devs;
p2p_devs = p2p.Install (nodes);
```

Notice that we used `StringValue`, meaning that `ns-3` parses the values accordingly. `ns-3` has a base class named `AttributeValue` and many sub-classes of it, such as `StringValue`, `DataRateValue`, `TimeValue`, `DoubleValue` and others. The previous attribute setup can be done alternatively as:

```
p2p.SetDeviceAttribute ("DataRate", DataRateValue (DataRate (5000000))); //in bps
p2p.SetChannelAttribute ("Delay", TimeValue (MilliSeconds (2)));
```

So far, the code created two nodes, and attached `PointToPointNetDevice` objects to each of them, and created a link between the two nodes. Note that `PointToPointHelper` only works with node containers that has exactly two nodes. It is also possible to use two node pointers instead of a 2-node container as follows:

```
Ptr<Node> n0 = NodeList::GetNode (0);
Ptr<Node> n1 = NodeList::GetNode (1);
p2p_devs = p2p.Install (n0,n1);
```

So far, there are no data being transmitted between the nodes. We need to first enable the communication stack.

```
InternetStackHelper stack;
stack.Install (nodes);
```

I'll spare you the details of what happens here, but you know about object aggregate, and what this code does is aggregate every node in `nodes` with an object of type `Ipv4` and `Ipv6`.

To assign IP address, we will use IPv4 addresses. `ns-3` has helper named `Ipv4AddressHelper`, and we can use it as follows

```
Ipv4AddressHelper address;
address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces = address.Assign (devices);
```

By now, the two nodes are capable of exchanging information. But we need to actually add something that triggers information exchange. The examples installs applications to the nodes, a `UdpEchoClient` in node 1, and `UdpEchoServer` in node 0, both are subclasses of `ns3::Application`. These two applications create sockets that are used to exchange information just like you would do in socket programming on Unix.

For the server installation, we will use the `UdpEchoServerHelper` which can install an application in a single node, or in all nodes of a `NodeContainer`. In both cases, the `Install` function call will return an `ApplicationContainer`, containing pointers to objects of type `UdpEchoServer`.

```
UdpEchoServerHelper echoServer (9); //listening on port 9
ApplicationContainer serverApps = echoServer.Install (nodes.Get (1)); //Install at node id 1.
serverApps.Start (Seconds (1.0)); //starting time of all applications in the container.
serverApps.Stop (Seconds (10.0)); //stopping time
```

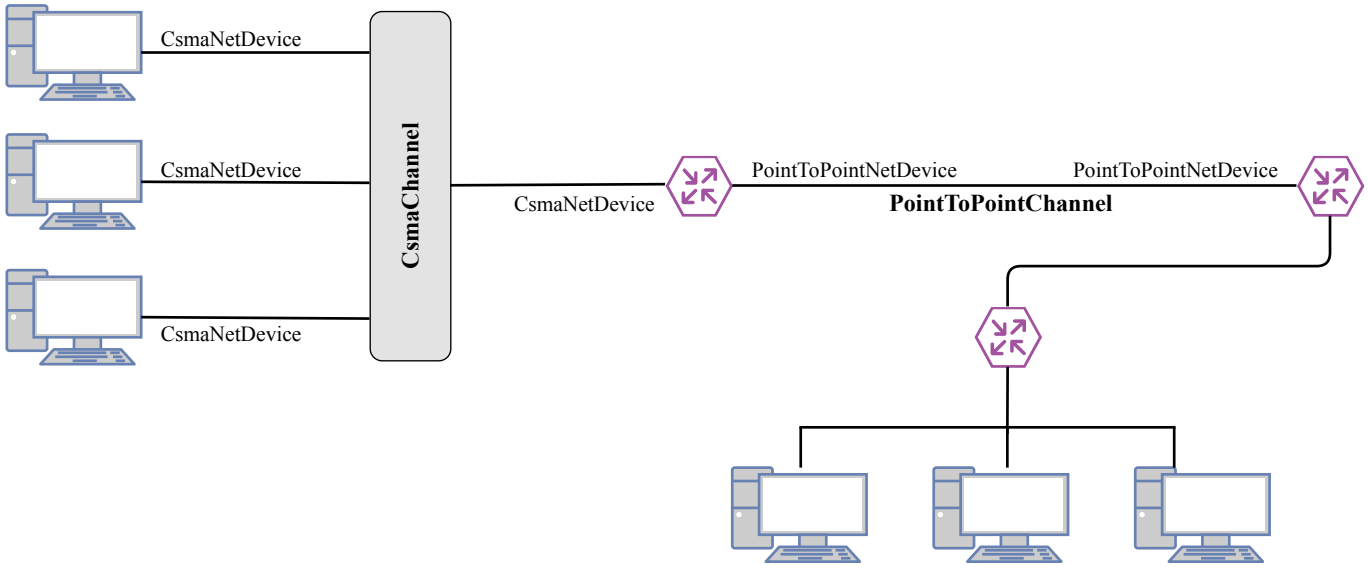
We will install a `UdpEchoClient` at node 1, targeting the IP address and port of the server application at node 0.

```
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)); //install in node 0
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

1.3.2 Let's create two LAN networks

Let us create two LAN networks connected by multiple routers as shown in this figure



First, let us create the simulation nodes

```
NodeContainer lan1_nodes;
lan1_nodes.Create (3);

NodeContainer lan2_nodes;
lan2_nodes.Create (3);

//Creating the three routers. We could use a NodeContainer, but I think this is clearer way of doing it.
Ptr <Node> r1 = CreateObject <Node> ();
Ptr <Node> r2 = CreateObject <Node> ();
Ptr <Node> r3 = CreateObject <Node> ();

//router 1 is also part of LAN 1, so we'll add it to the NodeContainer
lan1_nodes.Add (r1);
//router 3 is also part of LAN 2, so we do the same
lan2_nodes.Add (r3);
```

Let us now create a point-to-point link between router 1 and router 2, and another link between router 2 and router 3. We have seen this before in `first.cc`

```
PointToPointHelper p2p_link1;
p2p_link1.SetDeviceAttribute ("DataRate", StringValue ("20Mbps"));
p2p_link1.SetChannelAttribute ("Delay", StringValue ("3ms"));
NetDeviceContainer p2p_devs1 = p2p_link1.Install (r1, r2);

PointToPointHelper p2p_link2;
p2p_link2.SetDeviceAttribute ("DataRate", StringValue ("10Mbps"));
p2p_link2.SetChannelAttribute ("Delay", StringValue ("2ms"));
NetDeviceContainer p2p_devs2 = p2p_link2.Install (r2, r3);
```

To create nodes with Ethernet connection, we will use `CsmaHelper` (requires inclusion of `"ns3/csma-module"`)

```
CsmaHelper csma;
csma.SetChannelAttribute ("DataRate", StringValue ("1Gbps"));
csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));
NetDeviceContainer lan1_devs = csma.Install (lan1_nodes);
//Let's assume LAN 2 has the same attributes, we'll reuse the same helper
NetDeviceContainer lan2_devs = csma.Install (lan2_nodes);
```

By now, we have completed building the network. Notice that router 1 has two net devices of type `CsmaNetDevice` and `PointToPointNetDevice`, similarly with router 2. Router 3 has two `PointToPointNetDevice` objects attached to it. Next, we will be setting up the internet stack. We can use `InternetStackHelper`'s `InstallAll` to install the stack to all simulation nodes.

```
InternetStackHelper stack;
stack.InstallAll();
```

Setting up the IP addresses will be different. Let us have LAN 1 IP addresses start with `10.1.1.*` and LAN 2 IP addresses start with `192.168.1.*`. We will also assign other IPv4 patterns to the router links

```
Ipv4AddressHelper address;
address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer lan1_interface = address.Assign (lan1_devs);

address.SetBase ("192.168.1.0", "255.255.255.0");
Ipv4InterfaceContainer lan2_interface = address.Assign (lan2_devs);

address.SetBase ("11.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer link1_interface = address.Assign (p2p_devs1);

address.SetBase ("22.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer link2_interface = address.Assign (p2p_devs2);
```

By this point, there is no data transmission going on, so we need to setup some applications to transmit data. Before we do that, let us enable `pcap` file generation on all Ethernet devices

```
csma.EnablePcapAll ("traffic"); //'traffic' will be the file names prefix followed by node id and device id.
```

Let us use the same UDP Echo applications of `first.cc` and install clients in LAN 1, and servers in LAN 2

```
UdpEchoServerHelper echoServer (7777);
ApplicationContainer serverApps = echoServer.Install (lan2_nodes);
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
//All clients will target the IP & port of the first node in LAN 2
UdpEchoClientHelper echoClient (lan2_interface.GetAddress (0), 7777);
echoClient.SetAttribute ("MaxPackets", UintegerValue (3));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));
ApplicationContainer clientApps = echoClient.Install (lan1_nodes);
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

Before you are able to have packets traverse the network routers, you need to create routing rules. This can be done with a single line of code as follows:

```
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
```

Notes. You can see that `UdpEchoClientHelper` has a limitation: it creates a set of applications that all target the same IP address. It is possible to add code to change the IP address of the target for every application. Try this:

```
for (uint32_t i=0 ; i<clientApps.GetN(); i++)
{
  Ptr <UdpEchoClient> client_i = DynamicCast <UdpEchoClient> (clientApps.Get (i));
  client_i->SetRemote (lan2_interface.GetAddress (i), 7777);
}
```

It is also possible to do the same functionality manually, i.e. without using `UdpEchoClientHelper` as follows:

```
for (uint32_t i=0 ; i<lan2_nodes.GetN(); i++)
{
  Ptr<Node> n_i = lan1_nodes.Get (i);
  Ptr<UdpEchoClient> udp_i = CreateObject <UdpEchoClient> ();
  udp_i->SetDataSize (1024);
  udp_i->SetRemote (lan2_interface.GetAddress (i), 7777);
  /*
  There are no functions to set MaxPacket & Interval. This is an ns-3 design flaw in my opinion
  We can use SetAttribute to set these parameters
  */
  udp_i->SetAttribute ("MaxPackets", UintegerValue (3));
  udp_i->SetAttribute ("Interval", TimeValue (Seconds (1)));
  //Set the start & stop times
  udp_i->SetStartTime (Seconds (2.0));
  udp_i->SetStopTime (Seconds (10.0));
  //Add the created application to node i
  n_i->AddApplication (udp_i);
}
```

Notice that we used `SetAttribute` for `MaxPackets` and `Interval`. This is because the designers of `UdpEchoClient` did not put public `C++` functions to set these values. The function `SetAttribute` works with any object of type that is a subclass of `ns3::Object` and has a `public static` function called `GetTypeId`. The attributes are added in `UdpEchoClient`'s `GetTypeId` function, which has access to the `private` instance variables of the `UdpEchoClient`'s `private` instance variables.

Another way to create the `UdpEchoClient` applications is by using an object factory pattern. This is the method that `ns-3` developers use inside the *Helper* classes, which uses `ns3::ObjectFactory`

```
ObjectFactory fact;
fact.SetTypeId ("ns3::UdpEchoClient");
fact.Set ("MaxPackets", UintegerValue (3));
fact.Set ("Interval", TimeValue (Seconds (1.0)));
fact.Set ("PacketSize", UintegerValue (1024));
fact.Set ("RemotePort", UintegerValue (7777));

for (uint32_t i=0 ; i<lan2_nodes.GetN(); i++)
{
```

```
  Ptr <Node> n_i = lan2_nodes.Get (i);
  fact.Set ("RemoteAddress", AddressValue (lan2_interface.GetAddress (i)));

  Ptr<UdpEchoClient> udp_i = fact.Create <UdpEchoClient> ();

  udp_i->SetStartTime (Seconds (2.0));
  udp_i->SetStopTime (Seconds (10.0));
  n_i->AddApplication (udp_i);
}
```

Chapter 2

ns3::Config namespace

In Object-Oriented programming, a class can have *characteristics/attributes* and *operations/functions*. The attributes of a class are created as instance variables. In `ns-3`, attributes are indeed are created as instance variables of their respective classes, however, it adds a new higher layer to access and modify these attributes. `ns-3` uses Object Factory pattern in many of its helper classes to construct objects, and that method relies on `ns-3` "attributes", which are defined in a class's `GetTypeId` function. For example, when we used the `UdpEchoClientHelper`, we used the attribute `MaxPackets` to set the maximum number of packets the client would send. Attributes are defined as a subclass of a class named `AttributeValue`, and subclasses have a name ending with `-Value`, such as `DoubleValue`, `TimeValue`, `EnumValue`, `PointerValue`, and `ObjectPtrContainerValue`, to name a few.

Note

One would expect that classes in `ns-3` are defined with a full set of accessor and mutator functions. However, this is not the case. Some attributes are only accessible through some tedious ways, requiring the use of `GetAttribute` & `SetAttribute` functions. For example, the `UdpEchoClient` class defines an attribute `MaxPackets` linked to an instance variable `m_count` for which there are no accessor or mutator functions as we have seen previously.
I personally see a problem with this design. It is always better to have checking done at compilation time rather than run-time. It is possible to pass `SetAttribute` the wrong attribute name, or the wrong attribute type, the code will compile as long as the attribute name is passed as `std::string` and the attribute value is passed as a subclass of `AttributeValue`

One of the main advantages of using `Config` is the ability to access simulation parameters with shorter code. Simulations would typically involve many nodes, each having their own net devices, and every net device will have its own attributes, and so users would use loops and check class types on the way to reaching an attribute or a TraceSource buried deep in a node. Instead, the `Config` allows us to specify a *path* string to access the attribute or TraceSource, and `ns-3` would perform regular expression matching to get to it.

2.1 Working with ns3 Attributes

Classes of ns3 are written in C++, and can have attributes and TraceSources. The root class for most of ns3 classes is ns3::Object which contains two important functions:

1. GetTypeId() a static function that is used to get a ns3::TypeId object representing the class. This is where attributes and TraceSources are added with .AddAttribute and .AddTraceSource respectively.
2. GetInstanceTypeId() : returns the run-time TypeId of the object. This is useful when we are working with inheritance.

Some classes, like ns3::Packet are not a subclass of ns3::Object and therefore does not have these two functions. Some classes in ns3 are a subclass of ns3::Object, but do not implement these two functions, for example ns3::ChannelAccessManager in the wifi module.

Attributes are added to a class in the GetTypeId function and they provide a way to access and modify private instance variables of the class by using the GetAttribute and SetAttribute functions. In typical object-oriented programming, we typically use what is called *accessor* and *mutator* functions to get and set values of a class's private instance variables. The attributes in ns-3 are useful because of the use of object factory to create objects, and one can set default values for attributes of objects of a certain class before we even create any object.

2.1.1 Using ns3 Attributes

Consider the following snippet for GetTypeId function of the WifiMacQueue class in ns-3. Attributes are defined with AddAttribute, and the code creates two attributes MaxSize and MaxDelay.

```
TypeId WifiMacQueue::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::WifiMacQueue")
    .SetParent<Queue<WifiMacQueueItem> > ()
    .SetGroupName ("Wifi")
    .AddConstructor<WifiMacQueue> ()
    .AddAttribute ("MaxSize",
      "The max queue size",
      QueueSizeValue (QueueSize ("500p")),
      MakeQueueSizeAccessor (&QueueBase::SetMaxSize,
      &QueueBase::GetMaxSize),
      MakeQueueSizeChecker ())
    .AddAttribute ("MaxDelay", "If a packet stays longer than this delay in the queue, it is dropped.",
      TimeValue (MilliSeconds (500)),
      MakeTimeAccessor (&WifiMacQueue::SetMaxDelay),
      MakeTimeChecker ())
    //rest of the code...
return tid;
}
```

Notice that the default maximum queue size is set with QueueSize ("500p"), meaning 500 packets, and the maximum queuing delay is set with MilliSeconds (500), meaning 500ms. You can change these default values in your script before you build your WiFi network using Config::SetDefault function. This causes newly created WifiNetDevice objects to have the defaults set by the user (More accurately, this affects newly created WifiMacQueue within WifiNetDevice objects)

We can modify the value of MaxDelay by using the SetAttribute() on objects of type WifiMacQueue.

```
Ptr<WifiMacQueue> wifi_queue = .....; //You get this somehow. I explain a bit later.
TimeValue tv;
tv.Set( MilliSeconds(250) );
wifi_queue->SetAttribute ("MaxDelay", tv ); //tv is passed by reference
```

However, the class does provide a public function, SetMaxDelay() that we can use to set the maximum delay if we have an object of type WifiMacQueue, so why do we use this method at all?

The answer is: it can be tedious to get the correct object. When you are working with a simulation involving a WifiMacQueue, users do not interact with the queue directly with the queue. You will need to obtain the corresponding NetDevice from a Node object, and since a node can contain multiple devices, you will need to loop through

them to find the appropriate device, and perform a type cast with DynamicCast, and then from there you need to get the MacEntity, and so on (as detailed in the next code listing.)

However, if we can use ns3's configuration to easily access the required resource, and modify it. You will not need the WifiMacQueue object and can use a path string of type std::string instead.

Let us say that we want to change the MaxDelay for WifiMacQueue of VI Access Category for all the nodes in our simulation. The normal programmatic way to do this is to loop through the nodes, and try to get a pointer to the intended WifiMacQueue object. The body of your loop would look like this, going over all nodes of a NodeContainer :

```
for (uint32_t i=0 ; i< wifiStaNodes.GetN() ; i++){
  Ptr<Node> node  = wifiStaNodes.Get(i);
  /* We have no idea if device 0 is a WifiNetDevice or not.
  The right way is to loop through the NetDeviceContainer of the Node, and check type */
  for (uint32_t j=0 ; j<node->GetNDevices() ; j++)
  {
    Ptr<NetDevice> device = node->GetDevice(j);
    //check if this device is WifiNetDevice, because nodes can have multiple devices
    if ( device->GetInstanceTypeId() == WifiNetDevice::GetTypeId() )
    {
      Ptr<WifiNetDevice> wifi_dev = DynamicCast<WifiNetDevice>(device);
      Ptr <WifiMac> mac = wifi_dev->GetMac ();
      //Wifi queues are defined in the abstract class RegularWifiMac, a subclass of WifiMac
      Ptr<RegularWifiMac> reg_mac = DynamicCast <RegularWifiMac> (mac);
      if (reg_mac) // if the dynamic cast failed, reg_mac would be 0 (NULL)
      {
        //There's no function to access the MAC queue. We'll use GetAttribute as follows:
        PointerValue val;
        reg_mac->GetAttribute ("Txop", val); //val is passed by reference
        Ptr <Txop> txop = DynamicCast <Txop> (val.GetObject ());
        //WifiMacQueue is extracted from the Txop object
        Ptr <WifiMacQueue>  txop_queue = txop->GetWifiMacQueue ();
        txop_queue->SetMaxDelay ( MilliSeconds (250));
      }
    }
  }
}
```

This looks like a lot of work, and making changes can be tedious. It should be done *after* you have already setup the WiFi devices. Also, this only changes the value for the non-QoS queue of the WifiNetDevice. In WiFi, there are five wifi MAC queues, a regular non-QoS queue, and 4 queues for the access categories: VO, VI, BE, and BK. Refer to the IEEE 802.11e standard for more on this. If you also want to change MaxSize of the VO access category queue, then you should go GetAttribute with VO_Txop instead of Txop.

Now, consider using the Config namespace with Set to set the AttributeValue for all 5 queues in one go!

```
TimeValue t;
t.Set(MilliSeconds(250));
Config::Set("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/*/Queue/MaxDelay",t);
```

which will change the MaxDelay value for the WifiMacQueue of all the AC queues in all nodes. The Config namespace performs matching to existing simulation components. Let us example the structure of the path.

- **"NodeList/*"** The global list of all nodes in the simulation. The asterix wildcard indicates matching all nodes. You can restrict the matching to a certain node id by specifying a number. For example to do it only for node with id number 3, then change that portion to "NodeList/3/". You can also use string concatenation, as in the case where you designed your own class that has a GetNodeId function to obtain the nodes id

  ```
  path = "/NodeList/" + std::to_string (GetNodeId()) + "/DeviceList ....
  ```

  The attribute NodeList is defined in a class called ns3::NodeListPriv. Take a look at the GetTypeId in that class.

  Now to continue writing the path, what are the possible values after the first asterix? Well, since NodeList is a vector of values of type Node, we need to look at the definition of the Node class, particularly, the GetTypeId(), since that's where attributes are defined. You can find it in the file named /src/network/model/node.cc or in the ns3 doxygen documentation website https://www.nsnam.org/doxygen/index.html, and you should

find that the class `ns3::Node` defines the following attributes:

1. **`DeviceList`**: The list of devices associated to this Node.
2. **`ApplicationList`**: The list of applications associated to this Node.
3. **`Id`**: The id (unique integer) of this Node.
4. **`SystemId`**: The systemId of this node: a unique integer used for parallel simulations.

We want to delve into devices, so we will go down the path of `DeviceList`

- **`"DeviceList/*/$ns3::WifiNetDevice"`** A list of `NetDevice` objects attached to a node. A node can have multiple net devices, for example a `WifiNetDevice` and an `LteNetDevice`. The devices are indexed from from 0 onwards. We may not know the order in which the devices were added to the node, but luckily we can use the `TypeId` to only match the desired entities.

  We use the `$ns3::WifiNetDevice` to only match with that type. If we are sure that all our net devices are of type `WifiNetDevice`, we can simply use the shorter path:

```
"/NodeList/*/DeviceList/*/Mac/*/Queue/MaxDelay"
```

  Going further from here, we can take a look at the attributes defined in `WifiNetDevice` to see a few attributes like `Mtu`, `Channel`, `Phy`, `Mac`, etc. We picked `Mac` as you can see in the path string.

- **`"Mac/*"`** The `WifiNetDevice` defines the attribute `Mac` in its `GetTypeId`, and the type is `PointerValue` of a class that is a subclass of the abstract class `WifiMac`. The asterix wild card will match with all attributes of the actual run-time type of this object. In a simple case of WiFi access points and stations, this will match to `ApWifiMac` for the access point and `StaWifiMac` for the stations connected through the access point. Both of these classes are descendant of `RegularWifiMac`, which in turn is a subclass of `WifiMac`. Subclass inherit the attributes defined in their ancestor classes.

  `WifiMacQueue` objects are extracted from `Txop` objects defined in `RegularWifiMac`. The asterix here matches all attributes of the inheritance path from `WifiMac` all the way to either of `StaWifiMac` and `ApWifiMac`. The attributes that are matched are the ones defining a `Queue` attribute, and in this case, there will be 5 matches per net device, namely, `VO_Txop`, `VI_Txop`, `BE_Txop`, and `BK_Txop` and the non-QoS `Txop`. The QoS `Txop` objects are of type `ns3::QosTxop` while the non-QoS object is of type `ns3::Txop`. If you only want to match the QoS queues, the path strings should be

```
"/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/*/$ns3::QosTxop/Queue/MaxDelay"
```

  Note that if you replace `$ns3::QosTxop` with `$ns3::Txop`, all five Txop objects will be matches because `QosTxop` is a subclass of `Txop`. In case you are wondering what Txop means, it stands for *Transmission Opportunity*

- **`"Queue/MaxDelay"`** the last part of the path points to the `Queue` defined in `Txop`. The `Queue` attribute is of type `WifiMacQueue`. The attribute `MaxDelay` is defined in `WifiMacQueue`, and it is set with an `AttributeValue` of type `TimeValue`.

Caveat when using `Config::Set` and attribute paths

Using paths to an attribute is a convenient way to set simulation parameters. However, these strings are evaluated at run-time, meaning that there is the possibility of a run-time error if you messed it up. Currently in `ns-3.32`, if there is an error in the string, nothing would happen. That is, the value is not set, and you won't get a run-time error. You can replace `Set` with `SetFailSafe` which returns 0 if it did not succeed and 1 on success. I believe in recent versions of from `ns-3.31` onwards, the `Config::Connect` would cause a run-time error if the string is wrong, and a new function `ConnectFailSafe` was recently introduced which returns 0 on failure and 1 on success.

The passed value is of type `ns3::TimeValue`. This is a specialization of the base class `ns3::AttibuteValue`. If the path to the attribute is correct and you passed an `AttributeValue` other than `TimeValue`, then you will get a run-time error. This was my method of determining whether the path was correct: purposefully matching the wrong `AttributeValue` type. You should use `SetFailSafe` to make sure you are doing it correctly

Other subclassess of `AttributeValue` include `PointerValue`, `IntegerValue`, `DataRateValue`, and many more.

2.1.2 Parent's class attributes

Take a look at `ns3::OcbWifiMac`'s `GetTypeId()` function that can be found in **src/wave/model/ocb-wifi-mac.cc**

```
TypeId OcbWifiMac::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::OcbWifiMac")
    .SetParent<RegularWifiMac> ()
    .SetGroupName ("Wave")
    .AddConstructor<OcbWifiMac> ()
    ;
  return tid;
}
```

As you can see, a `TypeId` object can be used to get a reference to the class's parent class, group name[1] and a default constructor. However, we can access the attributes of its base classes, `RegularWifiMac`, and `WifiMac`. As a matter of fact, you can not add an attribute in a subclass with a name that already exists in its parent class.

2.1.3 Default Values

Before creation of `ns3` object, we can specify default values of certain attributes. For example, `ns3::QueueBase` is the basis of QoS queues used in 802.11, and it has the default size of 100. If we want net devices to be created with size 50 instead, we should use `Config::SetDefault` to set the new default value *before* we create them. If you are using a helper class to configure nodes with `WaveNetDevice`, then set the default before you use the helper.

```
//QueueSizeValue is a subclass of AttributeValue
QueueSizeValue newSize = QueueSizeValue (QueueSize ("50p"));
Config::SetDefault ("ns3::QueueBase::MaxSize", newSize );
//Everything created after will have initial queue capacity of 50
```

2.1.4 Accessing Values

The attribute `TxQueue` is defined in the class of which `net0` is an instance

```
PointerValue tmp;
net0->GetAttribute ("TxQueue", tmp);
Ptr<Object> txQueue = tmp.GetObject ();
```

When we have integer values, we do:

```
UintegerValue limit;
dtq->GetAttribute ("MaxPackets", limit);
NS_LOG_INFO ("1.  dtq limit: " << limit.Get () << " packets");
```

Accessing by `Config` namespace:

```
Config::Set ("/NodeList/0/DeviceList/0/TxQueue/MaxPackets", UintegerValue (25));
txQueue->GetAttribute ("MaxPackets", limit);
NS_LOG_INFO ("Limit changed through namespace: " << limit.Get () << " packets");
```

[1] used to sort and group documentation. Not all that important

2.2 Working with Callbacks and TraceSources

In ns3, callbacks are pointers to functions. Callbacks can be declared as a class member and may use typedef to give it a more readable name. You are probably not going to declare callbacks yourself as it is somewhat an advanced concept, but you are going to connect to TraceSources that fires when events occur, and you need to create a callback function matching how the callback was declared. Let us look at a usage example to illustrate usage.

2.2.1 Example of usage

Let us say you are running ns3 simulation that uses the wifi module. You want to trace all received packets on the physical layer level and print specific information. If you look at the ns-3 documentation for *All TraceSources* you can find a list of trace sources under the ns3::WifiPhy class such as PhyTxBegin, PhyTxDrop and MonitorSnifferRx. TraceSources are created in a ns-3 class in the GetTypeId function with the AddTraceSource function call. TraceSources are inherited by sub-classes just like attributes.

I am particularly interested in a TraceSource called MonitorSnifferRx that gets called every time a packet is received in monitor mode since it is provides more information. The trace is in WifiPhy. A WifiNetDevice contains a Phy attribute that is of type WifiPhy, and we want to connect to MonitorSnifferRx so that it calls our user-defined callback function RxPacketInfo. To connect to the TraceSource, we'll use the following regular expression string (it's called *path*) as follows

```
int main (int argc, char *argv[])
{
  ...
Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/MonitorSnifferRx",
                 MakeCallback (&RxPacketInfo)
                 );
  ...
}
```

The big question now is what is the correct signature to RxPacketInfo?. First of all, this call to Config::Connect will require that the callback function's first argument is of type std::string for *context*. The *context* is the whatever matched the regular expression and caused the trace to fire. This could be any node (since we used /NodeList/*) that has a reception on PHY-level.

We can investigate the Doxygen documentation for WifiPhy, and look under TraceSources for the trace MonitorSnifferRx, which says that the callback signature should be ns3::WifiPhy::MonitorSnifferRxTracedCallback, that is because the code for the class (in src/wifi/model/wifi-phy.cc) defines it:

```
typedef void (* MonitorSnifferRxCallback)(Ptr<const Packet> packet,
            uint16_t channelFreqMhz,
            WifiTxVector txVector,
            MpduInfo aMpdu,
            SignalNoiseDbm signalNoise);
```

Another way to find the callback is to look at WifiPhy's GetTypeId to where the trace source is added

```
  .AddTraceSource ("MonitorSnifferRx",
    "Trace source simulating a wifi device in monitor mode "
    "sniffing all received frames",
    MakeTraceSourceAccessor (&WifiPhy::m_phyMonitorSniffRxTrace),
    "ns3::WifiPhy::MonitorSnifferRxTracedCallback")
```

We want to look at the variable in MakeTraceSourceAccessor which is m_phyMonitorSniffRxTrace which is declared in wifi-phy.h

```
TracedCallback<Ptr<const Packet>, uint16_t, WifiTxVector, MpduInfo, SignalNoiseDbm, uint16_t>
    m_phyMonitorSniffRxTrace;
```

We need to match the signature exactly, including the const and the & C++ operators. So we need to create RxPacketInfo in our project, with an additional first parameter of type std::string that holds the context (the resolved trace path). By default, Config::Connect requires that the first argument is the context string. You can connect to a TraceSource without context by using TraceConnectWithoutContext to a callback function without the

context string as first argument. Most of the time, you should always connect with context because you want to know which component fired the event.

```
void RxPacketInfo(std::string context, Ptr <const Packet> packet, uint16_t channelFreqMhz,
          WifiTxVector txVector,MpduInfo aMpdu, SignalNoiseDbm signalNoise)
{
  std::cout << context << std::endl;
  std::cout << "Recv. Packet of size " << packet->GetSize() <<
    " Signal= " << signalNoise.signal << " Noise= " << signalNoise.noise << std::endl;
}
```

Printing the context string we will get the resolved path, indicating the source which fired the event

```
"/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorSnifferRx"
```

Indicating that the receiver is node 1, the device id within the node is 0 and the trace source that fired `MonitorSnifferRx`.

2.2.2 Using tracesource and connecting callbacks

`ns3` classes define trace sources in the `GetTypeId` function with `.AddTraceSource`. Trace sources must point to a callback of type `TracedCallback`. The example trace I used earlier `MonitorSnifferRx` trace is defined in the `GetTypeId` in the `WifiPhy` class

```
.AddTraceSource ("MonitorSnifferRx",
    "Trace source simulating a wifi device in monitor mode "
    "sniffing all received frames",
    MakeTraceSourceAccessor (&WifiPhy::m_phyMonitorSniffRxTrace),
    "ns3::WifiPhy::MonitorSnifferRxTracedCallback")
```

The part `ns3::WifiPhy::MonitorSnifferRxTracedCallback` refers to a callback created in `WifiPhy` with `typedef` keyword as shown in the usage example. What that means is that the function to fire the callback should have `void` as a return type, and the parameter list `Ptr<Packet>`, `uint16_t`, `WifiTxVector`, `MpduInfo` and `SignalNoiseDbm`.

The callback is declared as `m_phyMonitorSniffRxTrace` in `WifiPhy` as follows:

```
TracedCallback<Ptr<const Packet>, uint16_t, WifiTxVector, MpduInfo, SignalNoiseDbm> m_phyMonitorSniffRxTrace;
```

Keep in mind that this tutorial was written when `ns-3.29` was the most current version. `ns3` gets updated and particular callbacks and functions do change. Currently, `ns-3.30` is the upcoming version, and one can see that `WifiPhy` has undergone some changes in `ns-3-dev` that will probably be released in `ns-3.30`

2.2.3 Understanding callbacks

Call back provides a way for user-defined functions to be invoked by other components. In `ns-3`, we can set a `Receive` callback function for a `NetDevice`. For example, if you go back to the two LAN network example, we can set a receive callback function for the net devices of router `r1` as follows

```
...
//Set the receive callback
for (uint32_t i=0 ; i < r2->GetNDevices(); i++)
{
  Ptr<NetDevice> dev_i = r1->GetDevice (i);
  dev_i->SetReceiveCallback (MakeCallback(&R1DeviceRecv));
}
...
//define the function to be invoked
bool R1DeviceRecv(Ptr<NetDevice> device, Ptr<const Packet> packet, uint16_t protocol, const Address & address)
{
  std::cout << "Node: " << device->GetNode()->GetId() << " Dev: " << device->GetInstanceTypeId ()
  << " Protocol: " << std::hex << protocol << " from " << address << std::endl;

  std::cout << "\t" << packet->ToString() << std::endl; //need to do 'Packet::EnablePrinting ()' for this
  return true;
}
```

The function `SetReceiveCallback` is defined in `NetDevice` with a callback defined as `ReceiveCallback`

```
typedef Callback< bool, Ptr<NetDevice>, Ptr<const Packet>, uint16_t, const Address & > ReceiveCallback;
```

Where the return type is `bool` and the parameter list are of the types `Ptr<NetDevice>`, `Ptr<Packet>`, `uint16_t`, and `Address`. You must match the callback signature exactly, including the `const` and the `&` operator.

Using SetReceiveCallback for a NetDevice

Becareful using functions with names like `SetReceiveCallback`. This can override functionalities of sockets that actually bind to a net device. Try the example here with the two-LAN network example, and see if it still works.
If you want to capture incoming packets, then you should probably connect to TraceSources of the intended net device. For example, `CsmaNetDevice` has trace sources named `PhyRxEnd` and `PhyTxEnd` that you can connect to.

You can think of the `ReceiveCallback` of a net device as the packet being forwarded up to application layer, and therefore, the packet will not contain the MAC layer headers.

Printing a packet's information

To print a packets information such as headers & payload size, there is a function called `ToString()` that can be called on a packet's object
`std::cout « packet->ToString () « std::endl;`
However, the call will return an empty string if you haven't enabled packet printing before you started the simulation with
`Packet::EnablePrinting ();`

2.3 Useful TraceSources & Attributes

Here are some common TraceSources that you may need to connect to in an `ns-3` simulation

2.3.1 WiFi PHY Packet Drops

For WiFi packets that are dropped in the PHY-level, we are interested in connecting to a trace source called `PhyRxDrop`, which is defined in `WifiPhy`. Note that the signiture required for the function to handle this has changed in `ns-3.30`, to include the reason why a packet was dropped. If we are using a `WifiNetDevice`, for example, we will look at the class's `GetTypeId` function to see that the physical layer implemented as `WifiPhy`, and it has an attribute called `Phy`, pointing to a single `WifiPhy`. In the `WifiPhy` class, a trace source named `PhyRxDrop` is defined, and we can connect to it as follows:

```
void PhyRxDropTrace (std::string context, Ptr<const Packet> packet, WifiPhyRxfailureReason reason) //reason
    was added in ns-3.30
{
//What to do when packet loss at PHY-level happened.
//Suggestion: Maybe check some packet tags you attached to the packet
}
//...
int main (int argc, char *argv[])
{
...
std::string path="/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop";
Config::Connect (path, MakeCallback(&PhyRxDropTrace));
...
}
```

Take a look at the `doxygen` documentation of `WifiPhy` class reference, and you'll notice two `Config` paths:

```
"/NodeList/[i]/DeviceList/[i]/$ns3::WifiNetDevice/Phy"
"/NodeList/[i]/DeviceList/[i]/$ns3::WaveNetDevice/PhyEntities/[i]"
```

For your reference, this is the URL `https://www.nsnam.org/doxygen/classns3_1_1_wifi_phy.html`

This is because in `ns-3`, there are two net devices that uses `WifiPhy`, the `WifiNetDevice` of the `wifi` module, and the `WaveNetDevice` of the `wave` module for vehicular communication using Dedicated Short-Radio Communication (DSRC).

Not that if we were using a `WaveNetDevice` that the path is different. This is because the `WaveNetDevice` is designed to have one or more physical layer entity, and it is implemented as a vector of `WifiPhy` values, and the attribute used to point to that vector is called `PhyEntities`. The path for a `WaveNetDevice` would be

```
std::string path="/NodeList/*/DeviceList/*/$ns3::WaveNetDevice/PhyEntities/*/PhyRxDrop";
```

and the callback function is the same as the one we used for `WifiNetDevice`

Note

You can look up "All TraceSource" in `ns-3`'s `doxygen` for all possible TraceSources. You can probably trace packet drops on PHY & MAC layer for most net devices. I am particularly interested in WiFi PHY because I wanted to trace loss due to collision (two devices attempting to use the same channel at the same time) as part of my research.

2.3.2 WiFi Queue Information

When we use `WifiNetDevice` or `WaveNetDevice`, outgoing packets are queued in the MAC layer for transmission. Everytime a packet is queued, a `WifiMacQueueItem` is created as wrapper around the packet. Everytime a `WifiMacQueueItem` is created, a timestamp is set. `WaveNetDevice` is different than `WifiNetDevice` in that it defines a vector of `MacEntities` instead of a single `Mac` entity. `WaveNetDevice` has 7 MAC entities corresponding to the 7 DSRC channels. The paths for them are:

```
//this points to the single MAC entity
std::string wifi="/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac"
//this points to the vector of MAC entities
std::string wave="/NodeList/*/DeviceList/*/$ns3::WaveNetDevice/MacEntities/*";
```

The MAC entities are all of type `RegularWifiMac`, or one of its descendant classes. We are interested in the attributes `Txop`, for non-Qos and 4 attributes of type `QosTxop` named `VO_Txop`, `VI_Txop`, `BE_Txop` and `BK_Txop`.

The class `QosTxop` is a subclass of `Txop`. If we examine `Txop`'s attributes, we will find that it contains an attribute named `Queue`, with the type `WifiMacQueue`, whose parent is the class `Queue<WifiMacQueueItem>`. The `ns3::Queue<T>` class defines multiple trace sources named `Enqueue`, `Dequeue`, `Drop`, `DropBeforeEnqueue` and `DropAfterDequeue`

In order to run a code after a packet is dequeued, we will connect to `Dequeue` as follows (assuming a `WaveNetDevice`)

```
std::string path;
path = "/NodeList/*/DeviceList/*/$ns3::WaveNetDevice/MacEntities/*/$ns3::OcbWifiMac/*/Queue/Dequeue";
Config::Connect (path, MakeCallback(&DequeueTrace) );
...
void DequeueTrace(std::string context, Ptr<const WifiMacQueueItem> item)
{
double queuing_delay = Simulator::Now() - item->GetTimeStamp();
//Assuming queue max delay is 500ms
if (queuing_delay > MilliSeconds (500) )
{
//packet was dropped because it exceeded max delay
}
}
```

If node 0 had the packet enqueued at the MAC entity associated with channel 178, in the highest priority queue (`VI_Txop`), then the value of `context` would be:

```
"/NodeList/0/DeviceList/0/$ns3::WaveNetDevice/MacEntities/178/VI_Txop/Queue/Dequeue"
```

We can use string manipulation to get the substrings "178" and VI_Txop out of the context string, which would allows us more flexibility in tracking the queueing process. It is likely that packets queued in VI_Txop would experience much lower latency than other queues. It is also possible that other queues, like BK_Txop, will drop many packets due to exceeding the maximum queuing delay time. The TraceSource DropBeforeEnqueue fires if a a packet is dropped before it was enqueued, probably because the queue is full. DropAfterDequeue is fired if a packet was dequeued but not because it was sent out for transmission, but rather dropped, which is the case when a packet exceeds its time in the queue. If a packet was dropped because it was dequeued due to exceeding the queuing time, both DropAfterDequeue and Drop will fire.

2.3.3 IP Packet Drops

When you work with IP-packets, packets can be dropped for many reason. The first reason is that a router cannot route a packet, so it drops the packet. When we setup the internet stack, Ipv4 objects (and others) get aggregated to nodes. Take a look at the class TraceSources for Ipv4L3Protocol as it includes TraceSources for Drop, UnicastForward and others. The documentation states that the path to reach this is

```
"/NodeList/[i]/$ns3::Ipv4L3Protocol"
```

Notice that this works because ns3::Ipv4L3Protocol is *aggregated* to nodes, and will match with nodes that have IPv4 capabilities. We will connect to the traces as follows

```
void IpLayerDropTrace (std::string context, const Ipv4Header &header, Ptr<const Packet> packet, DropReason
    reason, Ptr<Ipv4> ipv4, uint32_t interface)
{
  //Your code. You may need to replace 'DropReason' with Ipv4L3Protocol::DropReason
}
void IpUnicastForward (std::string context, const Ipv4Header &header, Ptr< const Packet > packet, uint32_t
    interface)
{
  //Your code
}

int main (int argc, char* argv[])
{
  //... Setup Internet stack first!
  Config::Connect ("/NodeList/*/$ns3::Ipv4L3Protocol/Drop", MakeCallback (&IpLayerDropTrace));
  Config::Connect ("/NodeList/*/$ns3::Ipv4L3Protocol/UnicastForward", MakeCallback (&IpUnicastForward));
  //...
}
```

Another reason why packets might be dropped is due to queuing. If you studied TCP congestion control, then you know that routers do not only drop packets when their queues are full (which is called DropTail queuing), but they may also employ other queuing *disciplines*, where active queuing management like CoDel (Correlated Delay) or RED (Random Early Detection). By default, routers implement FqCoDel queuing discipline in their traffic control layer. To monitor queue drops due to queuing discipline in use:

```
void TrafficControlDrop (std::string context, Ptr<const QueueDiscItem> queue_item)
{
  //your code
}
void NodeStats::TcDropBeforeEnqueue (std::string context, Ptr<const QueueDiscItem> queue_item, const char*
    reason)
{
  std::cout << "Drop Reason: " << reason << std::endl;
}
//...
int main (int argc, char *argv[])
{
  //...
  Config::Connect ("/NodeList/*/$ns3::TrafficControlLayer/RootQueueDiscList/*/Drop", MakeCallback (&
    TrafficControlDrop));
  Config::Connect ("/NodeList/*/$ns3::TrafficControlLayer/RootQueueDiscList/*/DropBeforeEnqueue", MakeCallback
     (&TcDropBeforeEnqueue));
  //...
}
```

Do not get confused with ns-3 naming, QueueDisc is short for Queuing Discipline; it has nothing to do with shape.

2.3.4 Using a different queuing discipline

The queuing discipline for routers is set automatically to FqCoDel when you assign the IP addresses to nodes with the Assign function call. We do not want to modify ns-3 existing code and change this default; we will remove the existing queuing discipline and add a new one to a node as follows. Let us say we want to do this to node 4, and change its queuing discipline to ns3::FifoQueueDisc with a maximum size of 50 packets

```
std::string queue_disc = "ns3::FifoQueueDisc";
uint32_t max_size = 50;
QueueSize queue_size (QueueSizeUnit::PACKETS, max_size);

Ptr <Node> n = NodeList::GetNode (4);
for (uint32_t i=0 ; i<n->GetNDevices(); i++)
{
  Ptr<NetDevice> device = n->GetDevice (i);
  Ptr<TrafficControlLayer> tc = n->GetObject <TrafficControlLayer> ();
  if (DynamicCast<LoopbackNetDevice> (device) == 0)
  {
    tc->DeleteRootQueueDiscOnDevice (device);
    Ptr<NetDeviceQueueInterface> ndqi = device->GetObject<NetDeviceQueueInterface> ();
    if (ndqi)
    {
      TrafficControlHelper tcHelper;
      /*
      If you want to set Queuing Discipline (QueueDisc) attributes, pass a list of attributes names and
    attribute values
      For example,
        tcHelper.SetRootQueueDisc ("ns3::FqCoDelQueueDisc", "UseEcn", BooleanValue(true))
      where "UseEcn" is an attribute of FqCoDelQueueDisc
      */
      tcHelper.SetRootQueueDisc ("ns3::FifoQueueDisc", "MaxSize", QueueSizeValue (queue_size));
      tcHelper.Install (device);
    }
  }
}
```

The call to SetRootQueueDisc can take multiple attributes and attribute values if the corresponding queuing discipline defines multiple attributes. FifoQueueDisc has only one attribute, MaxSize. ns3::RedQueueDisc has over 20 attributes. Here's an example using two attributes of FqCoDelQueueDisc class, but you can use more!

```
tcHelper.SetRootQueueDisc ("ns3::FqCoDelQueueDisc", "MaxSize", QueueSizeValue (queue_size), "UseEcn",
    BooleanValue (true));
```

Adaptive & Gentle RED (ARED) are implemented in the same class RedQueueDisc, where you set the attributes Gentle and ARED to BooleanValue (true) respectively to enable them.

2.3.5 Using paths to set application parameters

Recall that a node can have a list of application within it, and so we can access applications and set their attributes with a path string. For example, to set the number of packets sent by UdpEchoClient to 5:

```
Config::Set ("/NodeList/*/ApplicationList/*/$ns3::UdpEchoClient/MaxPackets", UintegerValue (5));
```

Which must be done *after* you have created the applications and added them to the nodes. The UdpEchoClient application's default value for MaxPackets is 100, you can see that in the doxygen documentation or if you examine GetTypeId of the source file src/applications/model/udp-echo-client.cc. You can change that default with SetDefault as follows:

```
Config::SetDefault ("ns3::UdpEchoClient::MaxPackets", UintegerValue (5));
```

Which must be done *before* you created the applications

Chapter 3

Useful Tips

3.1 Logging

It is a common habit of programmers to print out message that helps them trace and log what their program does. For example

```
void SomeClass::DoSomething (int x)
{
  std::cout << "Inside DoSomething() with x=" << x;
}
```

This way, we can get some information about what is happening in our program, for example, whether the function `DoSomething` is getting invoked at all. The drawback here is that this message will always get printed, and you need to comment it out if you do not want to display. If you have many of these 'logging' statements, then you will have to comment them out in many places in your code, which is inconvenient. It would be nice if we can easily turn a switch, and chose whether we want to print out these statements, and that's where `ns3` logging can be helpful.

In `ns3`, many C++ classes are designed with a logging functionality, allowing you to trigger them on and off. `ns3` defines multiple types of log messages (or levels), for example 'info', 'warning', 'function' and 'error'. The following is a breakdown of those levels.

Severity	Meaning
LOG_NONE	The default, no logging.
LOG_ERROR	Error messages
LOG_WARN	Warning messages
LOG_DEBUG	For use in debugging
LOG_INFO	Informational.
LOG_FUNCTION	Function tracing.
LOG_LOGIC	Control tracing.

3.1.1 Enabling Logging

Logging can be enabled on `ns3` components that supports it. An `ns3` component supports logging if it has the line

```
  NS_LOG_COMPONENT_DEFINE ("ComponentName");
```

Where "ComponentName" is a name given to the component. This does not need to match the C++ class name, or need to be in a class. In fact, you can do that in your `ns3` programs that you run under `scratch` directory. For example:

```
#include "ns3/core-module.h"
using namespace ns3;
NS_LOG_COMPONENT_DEFINE ("MyThirdProgram");
void SomeFunction(int x){
  NS_LOG_FUNCTION ("x value" << x);
}
int main(int argc, char *argv[]) {
```

```
  LogComponentEnable ("MyThirdProgram", LOG_LEVEL_WARN);
  NS_LOG_INFO ("Program Started!");
  SomeFunction (5);
  NS_LOG_WARN ("Warning message");
  NS_LOG_ERROR ("An Error Happened");
}
```

We defined the component with the name `MyThirdProgram` and enable logging on `LOG_LEVEL_WARN`, meaning we will trigger printing messages with severity `WARN` or higher as shown in the table earlier.

As an example, we can enable logging for `ns3::UdpEchoClientApplication` to print out 'info' messages. The code for that is a simple one line

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

If you are creating your own `ns3` classes, it is recommended that you define logging for them, and make it a habit to add `NS_LOG_FUNCTION(this)` in all functions you create as it would help trace function invocation if needed.

3.1.2 Enabling log from command-line

Another way to enable logging is from the command-line you pass to `ns3`. Let us say we have a program called `DualRadio`, and you want to enable logging on `WifiPhy`. You can achieve that by running

```
NS_LOG="WifiPhy" ./waf --run DualRadio
```

You can enable multiple component at once separated by colon '`:`'

```
NS_LOG="WifiPhy:OcbWifiMac" ./waf --run DualRadio
```

You may also set the log-level from the command line

```
NS_LOG="WifiPhy=level_error:OcbWifiMac=level_debug" ./waf --run DualRadio
```

You can also specify a prefix for logging.

```
NS_LOG="Simulator=level_all:prefix_time" ./waf --run DualRadio
```

For more information, checkout `ns3` documentation `https://www.nsnam.org/docs/manual/html/logging.html`

3.2 Packet Tags

Packet tags do not add to the size of a packet, as they do not contribute anything to the payload of a packet. They are intended for use in simulation to attach certain information to a packet. You can only add a packet tag of a certain type once to a packet.

3.2.1 Creating packet tags

A packet tag is a subclass of `ns3::Tag`. To define a packet tag, you need to create a class as a subclass of `ns3::Tag`, which is an abstract class due to it having the following pure virtual functions:

```
virtual uint32_t GetSerializedSize (void) const = 0;
virtual void Serialize (TagBuffer i) const = 0;
virtual void Deserialize (TagBuffer i) = 0;
virtual void Print (std::ostream &os) const = 0;
```

Listing 1: Pure virtual functions in `tag.h`

In C++, pure virtual funcions are meant to be implemented by subclasses. If we want to create a packet tag class, we will have to implement those functions, but we can also add more functions as we see fit. Here's an example of source header file defining a packet tag class we call `PositionTag`:

```
namespace ns3 {
class PositionTag : public Tag {
public:
  static TypeId GetTypeId(void);
  PositionTag();
  virtual ~PositionTag();
  virtual TypeId GetInstanceTypeId(void) const;
  virtual uint32_t GetSerializedSize(void) const;
  virtual void Serialize (TagBuffer i) const;
  virtual void Deserialize (TagBuffer i);
  virtual void Print (std::ostream & os) const;
  //We added these functions
  void SetComplexValue(Vector v);
  Vector GetComplexValue();
private:
  Vector m_complexValue;
};
}
```

Listing 2: Header file of a user-defined packet tag class

We have added an instance variable of type `Vector`, and functions to set and get it. In `ns3`, the class `Vector` is used to create a 3-dimensional point.

Then we need to impement the class in a `.cc` file as follows:

```
uint32_t PositionTag::GetSerializedSize (void) const
{
  //Because we have three 'double' values
  return sizeof(double) * 3;
}
void PositionTag::Serialize (TagBuffer i) const
{
  i.WriteDouble (m_complexValue.x);
  i.WriteDouble (m_complexValue.y);
  i.WriteDouble (m_complexValue.z);
}
void PositionTag::Deserialize (TagBuffer i)
{
  m_complexValue.x = i.ReadDouble();
```

```
  m_complexValue.y = i.ReadDouble();
  m_complexValue.z = i.ReadDouble();
}
void PositionTag::Print (std::ostream &os) const
{
  os << "PositionTag=" << m_complexValue;
}
double PositionTag::GetSimpleValue (void) const
{
  return m_simpleValue;
}
void PositionTag::SetComplexValue(Vector v)
{
  m_complexValue.x = v.x;
  m_complexValue.y = v.y;
  m_complexValue.z = v.z;
}
```

Notice how the functions `Serialize` and `Deserialize` work. You wan to make sure that use the order for reading and writing. The `Print` is useful if your packet is used in a program where `ASCII` tracing is enabled.

3.2.2 Using packet tags

Suppose we have a packet tag defined in a class named `UserDefinedTag`, the way to add a packet to a packet object would look like this:

```
Ptr<Packet> pkt = Create<Packet>(100);
UserDefinedTag pTag;
pTag.SetSomeValue(some_value);
pkt->AddPacketTag(pTag);
```

Note that `ns3` will give a runtime error if you attempt to add a packet tag of the same type to a packet more than once.

To examine if a packet tag is added to a packet, we will use the `PeekPacketTag` function defined in the `Packet` class:

```
UserDefinedTag pTag;
if(pkt->PeekPacketTag(tmp_tag))
{
    //what to do if such tag exist. Maybe get a certain value
    uint32_t val = pTag.GetSomeValue();
}
```

This is very useful in simulation. In my code, I have defined a `PacketIdTag` that assigns a unique number to every packet created in the simulation. I also defined a `PositionTag` that attached the location of the node that created the packet.

3.3 `CommandLine` Interface for Arguments

Your `ns3` programs likely requires users to pass command-line arguments. These values typically take the value of `argv` where `argc` is the argument count. This is how it's done in typical `C++` programs

```
int main (int argc, char *argv[])
{
  uint32_t node_count;
  double simulation_time;
  if (argc < 3)
  {
    std::cout << "Usage example..." << std::endl;
  }
  if (argc==3)
  {
    number_of_nodes = atoi (argv[1]);
    simulation_time = atof (argv[2]);

  }
}
\\end{lstlisting}
\texttt{ns3} provides a \texttt{CommandLine} utilities that makes handling command line arguments much
    convenient. It allows users to supply default values for argument values that are not supplied by the user
    . An \texttt{ns3} program that uses this would look like:
\begin{lstlisting}[language=C++]
int main (int argc, char *argv[])
{
  CommandLine cmd;
  //We define the defaults
  uint32_t node_count = 2;
  double simulation_time = 10;
  bool use_broadcast = false;
  //for all args, we do: name, description, and variable name
  cmd.AddValue ("t", "Simulation time in seconds", simulation_time);
  cmd.AddValue ("n", "Number of nodes to create", node_count);
  cmd.AddValue ("bcast", "Whether packets should be broadcast", use_broadcast);

  cmd.Parse (argc, argv);
  ...
}
```

Now, we can run the program without arguments, and it will use the defaults defined here, or we can supply only one argument, or all arguments. For example

```
./waf --run ProgramName       #uses default values
./waf --run "ProgramName --n=5"     #sets node count to 5
./waf --run "ProgramName --n=5 --bcast"    #Sets node count to 5, use_broadcast to true
```

To print the list of arguments defined for a program with `cmd.AddValue`, run the program with the `-PrintHelp` argument

```
./waf --run "ProgramName --PrintHelp"
```

Which would output a list of options to use as arguments, with the default value between brackets. For example, this output

```
Program Options:
  --n:  Number of nodes [2]
  --t:  Simulation duration in seconds [8]
  --i:  Packet Interval in seconds [1]
  --l:  Log level [v]
```

indicated that the default number of nodes is 2.

3.4 Common Errors

When connecting to a TraceSource, you get a message that says “got" and “expected".

```
got=CallbackImpl<void,ns3::Ptr<ns3::Packet const>,ns3::Address,ns3::Address,ns3::SeqTsSizeHeader>
expected=CallbackImpl<void,std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >,ns3
    ::Ptr<ns3::Packet const>,ns3::Address,ns3::Address,ns3::SeqTsSizeHeader>", file=./ns3/callback.h, line
    =1584
msg="when connecting to /NodeList/2/ApplicationList/0/$ns3::PacketSink/RxWithSeqTsSize", file=./ns3/traced-
    callback.h, line=146
```

This message indicates that your path string is correct, but the callback function you provided does not match the signature. In this error message, I forgot to provide the context string as a first argument.

Chapter 4

Creating a custom application

As you know, an ns-3 node can have a list of applications added to it. ns-3 provides a list of useful applications such as UdpEchoClient, UdpEchoServer, PacketSink, PacketSocketClient, V4Ping and OnOffApplication to name a few.

In order to create a custom user-define application, we need to create a C++ class as a subclass of ns3::Application, where we can override the function StartApplication to provide code that runs when the application starts.

4.1 A Simple UDP socket application

I would like to create a simple UDP socket application. The application will have a function named SendPacket to send a packet that the user specifies to a target IP & port. Let us call our class SimpleUdpApplication. We will start by creating a folder under scratch and call it MyUdpExample and then create the files as follows:

```
scratch
└── MyUdpExample
    ├── simple-udp-application.h
    ├── simple-udp-application.cc
    └── main-program.cc
```

For the header file we will create the class definition. I always use include guards #ifndef when I write C++ classes, and I also would like to make the class part of ns-3, which is optional

The header file:

```
#ifndef NS3_SIMPLE_UDP_APPLICATION_H
#define NS3_SIMPLE_UDP_APPLICATION_H
#include "ns3/socket.h"
#include "ns3/application.h"
using namespace ns3;
namespace ns3
{
  class SimpleUdpApplication : public Application
  {
    public:
      SimpleUdpApplication ();
      virtual ~SimpleUdpApplication ();
      static TypeId GetTypeId ();
      virtual TypeId GetInstanceTypeId () const;
      void HandleRead (Ptr<Socket> socket);
      void SendPacket (Ptr<Packet> packet, Ipv4Address destination, uint16_t port);
    private:
      virtual void StartApplication ();
      Ptr<Socket> m_socket;
  };
}
#endif
```

And the `simple-udp-application.cc` file

```
#include "simple-udp-application.h"
#include "ns3/log.h"
#include "ns3/simulator.h"

namespace ns3
{
NS_LOG_COMPONENT_DEFINE("SimpleUdpApplication");
NS_OBJECT_ENSURE_REGISTERED(SimpleUdpApplication);

TypeId SimpleUdpApplication::GetTypeId()
{
  static TypeId tid = TypeId("ns3::SimpleUdpApplication")
    .AddConstructor<SimpleUdpApplication>()
    .SetParent<Application>();
  return tid;
}

TypeId SimpleUdpApplication::GetInstanceTypeId() const
{
  return SimpleUdpApplication::GetTypeId();
}

SimpleUdpApplication::SimpleUdpApplication()
{
}
SimpleUdpApplication::~SimpleUdpApplication()
{
}
void SimpleUdpApplication::StartApplication()
{
  TypeId tid = TypeId::LookupByName("ns3::UdpSocketFactory");
  m_socket = Socket::CreateSocket(GetNode(), tid);
  //Handles incoming packets on a ephemeral port
  m_socket->SetRecvCallback(MakeCallback(&SimpleUdpApplication::HandleRead, this));
}

void SimpleUdpApplication::HandleRead (Ptr<Socket> socket)
{
  NS_LOG_FUNCTION(this << socket);
  Ptr<Packet> packet;
  Address from;
  Address localAddress;
  while ((packet = socket->RecvFrom(from)))
  {
    NS_LOG_INFO("Received a Packet of size: " << packet->GetSize() << " at time " << Now().GetSeconds()
        << " from " << InetSocketAddress::ConvertFrom(from).GetIpv4());
    NS_LOG_INFO(packet->ToString());
  }
}
void SimpleUdpApplication::SendPacket(Ptr<Packet> packet, Ipv4Address destination, uint16_t port)
{
  NS_LOG_FUNCTION (this << packet << destination << port);
  m_socket->Connect(InetSocketAddress(Ipv4Address::ConvertFrom(destination), port));
  m_socket->Send(packet);
}
} // namespace ns3
```

Notice that `StartApplication` starts when the application start time is reached, which implies that an object of this type has been created and added to a node. We have a call to `GetNode()` within `StartApplication` so the application must be added to the node or this call will fail. This is also why the code is in `StartApplication` and not in the constructor, since if it was in the constructor, the call to `GetNode()` will be NULL, as the application wasn't added to a node yet.

To test this application, let us re-use the two LAN example where we set up `UdpEchoServer`, but now instead of setting up `UdpEchoClient` applications in LAN 1, let us simply install an application in one of the nodes, and have it target different nodes in LAN 2. We need to also enable logging on `SimpleUdpApplication` so that `NS_LOG_INFO` prints to the screen.

We will use the same code from the two LAN example in `main-program.cc` with few changes:

```
//...
#include "simple-udp-application.h"
//...
int main (int argc, char *argv[])
{
  //...
  LogComponentEnable ("SimpleUdpApplication", LOG_LEVEL_INFO);
  //...

  //After we setup UdpEchoServers listening on port 7777 on LAN1
  Ptr<Node> n_i = lan1_nodes.Get (0);
  Ptr<SimpleUdpApplication> udp_i = CreateObject <SimpleUdpApplication> ();
  udp_i->SetStartTime (Seconds (2.0));
  udp_i->SetStopTime (Seconds (10.0));
  n_i->AddApplication (udp_i);

  //Schedule calls to 'SendPacket' function in the application using the object 'udp_i'
  Simulator::Schedule ( Seconds (3), &SimpleUdpApplication::SendPacket, udp_i, Create<Packet> (200),
    lan2_interface.GetAddress (0), 7777);
  Simulator::Schedule ( Seconds (3.2), &SimpleUdpApplication::SendPacket, udp_i, Create<Packet> (1000),
    lan2_interface.GetAddress (1), 7777);
  Simulator::Schedule ( Seconds (3.4), &SimpleUdpApplication::SendPacket, udp_i, Create<Packet> (500),
    lan2_interface.GetAddress (2), 7777);
  //...
}
```

The `UdpEchoServer` application simply echos back the same packet to the sender, the output would be:

```
At time +3.01133s server received 200 bytes from 10.1.1.1 port 49153
At time +3.01133s server sent 200 bytes to 10.1.1.1 port 49153
Received a Packet of size: 200 at time 3.01965 from 192.168.1.1
Payload (size=200)
At time +3.20928s server received 1000 bytes from 10.1.1.1 port 49153
At time +3.20928s server sent 1000 bytes to 10.1.1.1 port 49153
Received a Packet of size: 1000 at time 3.22156 from 192.168.1.2
Payload (size=1000)
At time +3.41167s server received 500 bytes from 10.1.1.1 port 49153
At time +3.41167s server sent 500 bytes to 10.1.1.1 port 49153
Received a Packet of size: 500 at time 3.42635 from 192.168.1.3
Payload (size=500)
```

Ephemeral Port

In socket programming, we create a socket, say a UDP socket targeting an IP address and port number. When the networking stack in the operating system creates the packet with a UDP header, the header will have a destination port number that was specified by the user. It will also create a temporary port number as a source port number in the UDP header. This is called ephemeral (short lived) port number, and it can be used by recipients to reply to an incoming packet.

I talk about this on my YouTube video `https://www.youtube.com/watch?v=7bc0rtznq5I`

4.2 Periodic broadcast VANET application

Vehicular Ad-hoc NETworks (VANETs) are supported in `ns-3` in the `wave` module, and use the `WaveNetDevice` device. `WaveNetDevice` has a function named `SendX` by which users can specify per-packet transmission parameters.

Check it out on my YouTube

The example is on my Github `https://github.com/addola/NS3-HelperScripts/tree/master/examples/CustomApplicationExample` and I have a detailed YouTube video on it `https://www.youtube.com/watch?v=aJxM1P9XLCQ`

As you know, an application has access to the node's pointer, and therefore, access to the net devices installed in the node. The design of this application assumes that the node has a `WaveNetDevice` installed in the node. Let us call this class `CustomApplication` and create `custom-application.h` with a function called `BroadcastInformation`, a function to receive a packet `ReceivePacket` and an instance variable to point to the net device

```
public:
  void BroadcastInformation ();
  bool ReceivePacket (Ptr<NetDevice> device, Ptr<const Packet> packet,uint16_t protocol, const Address &sender
    );
  ...
private:
  void StartApplication ();
  Ptr <WaveNetDevice> m_wave_device;
  ...
```

In `StartApplication` we will make sure there is a `WaveNetDevice` in the node, or we'll through an error

```
void CustomApplication::StartApplication ()
{
  Ptr <Node> n = GetNode ();
  for (uint32_t i=0 ; i < n->GetNDevices (); i++)
  {
    m_wave_device = DynamicCast <WaveNetDevice> (n->GetDevice (i));
    //if the casting succeeds
    if (m_wave_device)
    {
      m_wave_device->SetReceiveCallback (MakeCallback (&CustomApplication::ReceivePacket, this));
      break;
    }
  }
  if (!m_wave_device)
    NS_FATAL_ERROR ("There's no WaveNetDevice in node " << n->GetId ());

  //All good, so schedule the first broadcast
  Ptr<UniformRandomVariable> rand = CreateObject<UniformRandomVariable> ();
  Time random_offset = MicroSeconds (rand->GetValue(50,200));
  Simulator::Schedule (m_broadcast_time+random_offset, &CustomApplication::BroadcastInformation, this);
}
```

The function `BroadcastInformation` will broadcast packet with 400 bytes payload every 100 milliseconds:

```
void CustomApplication::BroadcastInformation ()
{
  TxInfo tx;
  tx.channelNumber = CCH;
  tx.txPowerLevel = 7;
  tx.priority = 7
  tx.dataRate = WifiMode("OfdmRate6MbpsBW10MHz"); //Using 6 Mbps
  tx.preamble = WIFI_PREAMBLE_LONG;    //always set this to WIFI_PREAMBLE_LONG
  Ptr<Packet> packet = Create <Packet> (400); //Create a packet with 400 bytes payload
  m_wave_device->SendX (packet, Mac48Address::GetBroadcast(), 0x88dc, tx);
  //Schedule next broadcast
  Simulator::Schedule (MilliSeconds (100), &CustomApplication::BroadcastInformation, this);
}
```

The WAVE standard allows per-packet transmission parameters set in `TxInfo`. By default, `CCH` channel is enabled and for multi-channel operation we need to start service channel with `StartSch` on one of the other six service channels. When we setup `WaveNetDevice`, we can specify a lower and upper bounds on the transmission power with `TxPowerStart` and `TxPowerEnd` and it will be divided into seven levels with 7 by the highest. The `priority` determines the outgoing MAC queue as specified by IEEE 802.11e. We can specify `dataRate` to one of the supported data rates per-packet. Of course, your class design can define multiple instance variables that change according to your need so that different packets will be sent with different `TxInfo` parameters.

Now, the `ReceivePacket` function will simply receive packets without their `WifiMacHeader` information.

```
bool CustomApplication::ReceivePacket (Ptr<NetDevice> device, Ptr<const Packet> packet,uint16_t protocol,
    const Address &sender)
{
  NS_LOG_FUNCTION (device << packet << protocol << sender);
  NS_LOG_INFO ("ReceivePacket() : Node " << GetNode()->GetId() << " : Received a packet from " << sender << "
    Size:" << packet->GetSize());
  return true;
}
```

If you want to receive the MAC information as well, you can modify `StartApplication` and add connect to trace sources like `MacRx` of the `MacEntities` of the device. If you want to get received signal and noise values then connect to `MonitorSnifferRx` of the `PhyEntities` of the device.

```
std::string node_id = std::to_string (GetNode ()->GetId ());
Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WaveNetDevice/MacEntities/*/MacRx", MakeCallback (&
    CustomApplication::MacRxTrace, this));
Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WaveNetDevice/PhyEntities/*/MonitorSnifferRx", MakeCallback
    (&CustomApplication::SniffRx, this));
```

and create the callback function signature in the header and in your `.cc` file

```
void CustomApplication::MacRxTrace (std::string context, Ptr< const Packet > packet)
{
  //your code
}
void CustomApplication::SniffRx (std::string context, Ptr<const Packet> packet, uint16_t channelFreq,
    WifiTxVector tx, MpduInfo mpdu, SignalNoiseDbm sn)
{
  //your code
}
```

You can also connect without a context if you get the corresponding object

```
Ptr <OcbWifiMac> mac = m_wave_device->GetMac (CCH);
Ptr <WifiPhy> phy = m_wave_device->GetPhy (0); //assuming only one entity
mac->TraceConnectWithoutContext ("MaxRx", MakeCallback(&CustomApplication::MacRxTrace, this));
phy->TraceConnectWithoutContext ("MonitorSnifferRx", MakeCallback(&CustomApplication::SniffRx, this));
```

and remove the context string from the callback function.

Chapter 5

Extensions & Add-ons

5.1 Sharing ns3 work

The code for ns3's built-in modules changes with new releases. Custom built modules may not work if they are tested with different ns3 version. As of June 2019, the latest ns3 version is ns-3.29. However, the ns-3-dev portion of the ns3 repository may contain work-in-progress that is not part of the latest ns3 release. Therefore, it is important to note which version of ns3 was used with user-built modules.

In addition, when we first start working on ns3, we first need to perform ./waf configure to configure ns3. ns3 can be configured with different compilers and different compile and link flags. For example, to configure ns3 with clang++ instead of g++ compiler, we perform:

```
CXX="clang++" ./waf configure
```

Of course, we may also configure more parameters. It is advised that you modify the file **$NS3_ROOT_DIR/Makefile** and change the configure rule with the desired settings so that instead of ./waf configure, one can simply perform make configure to configure ns3 with the intended parameters. For example, to configure ns3 to use clang++ as a compiler and to disable warning errors, we can change the configure rule in the Makefile to:

```
configure:
  CXX="clang++" ./waf configure --disable-werror --enable--tests
```

When sharing code with others, it is advised that you specify some information on the parameters of your development environment, this includes but not limited to:

- **C++ compiler and version.** different users use different compiler versions, as there might be differences between newer and older versions of the same compiler (say, gcc-5 vs gcc-6)
- **Compiler flags.** Some users may use sepcific compile and link flags. A user publishing their work should specify if certain flags were used. I suggest that this would be specified in th Makefile.
- **OS version.** it can be helpful to determine which operating system was used for development.

When sharing ns3 code you are going to share either:

1. **ns3 projects** that is typicall made under scratch directory, or
2. **user-built modules.** which contains sources, headers and even examples accompanies with wscript file(s).

Either way, you can share your code either in a compressed folder (for example, tar archive) or a code repository (for example, git or svn)

5.1.1 Projects under scratch

It is convenient to have your ns3 projects that are under scratch in separate folders with its related files. One of the .cc must have a main function in each directory. This way, you can run a particular project by using its folder name.

For example, my `scratch` directory listing shows four directories:

```
scratch
├── BandwidthQueue
│   └── bandwidth-with-queue-main.cc
├── BareMinimumWsmp
│   ├── wsmp-example.cc
│   └── some-text.txt
├── DualRadio
│   └── dual-radio-test.cc
└── IPv6Example
    ├── main.cc
    ├── MyApplication.h
    └── MyApplication.cc
```

The file `dual-radio-test.cc` is in the directory `NS3_ROOT_DIR/scratch/DualRadio` and must have a `main` function. To run it, I simply run:

```
./waf --run DualRadio
```

So if I wanted to share the `DualRadio` project with others, I can simply compress the folder and share it. A person that want to use it can unpack the file into a directory under `scratch` and run it.

5.1.2 ns3 modules

As we mentioned earlier, `ns3` modules are stored in a directory with a `wscript` file associated with them. To use a shared module named `vanet` for example, you need to:

- Copy the module's directory to your `$NS3_ROOT_DIR/contrib`. We need to copy its files & folders to `contrib/vanet`.
- Run `./waf configure` or `make configure` to scan for the changes, and add the newly added module to the `ns3` build list.
- Run `./waf build` or just `./waf`.
- To use it in projects under `scratch`, include the module named `vanet` in your code using `ns3/vanet-module.h`

5.1.3 What if the obtained code does not work?

There are several reasons why a code you obtained from someone fail not work on your machine. Here is a list to help your with the troubleshooting:

- Obtained code was created to work with a different `ns3` version. It is common that the `ns3` development group refactored certain function, and re-worked certain classes.
- `C++` errors due to differnt compilers. This include older/newer compilers and different compilers like `clang++`
- `C++` errors because contributor configured their C++ with different flags
  - a different standard (e.g. `-std=c++11` vs `-std=c++17`)
  - Not treating warnnings as errore (e.g. `-disable-werror`)
- Missing `C++` libraries that contributors have installed on their machines (e.g. Boost). You may need to install additional libraries and/or re-configure `ns3` with different compile and link flags.

5.2 Using external C++ libraries

Most of the simulations that one would create will use ns-3 source code. However, you may need to use some C/C++ libraries external to your program. It is possible to include C++ header files from your C++ ns-3 scripts if they are in the system's include path. For example, if you would like to write some output to text files, you can include fstream as follows:

```
...
#include <fstream>
...
std::ofstream out_file;
...
out_file << "Some text output" << std::endl;
out_file.close();
```

5.2.1 Using GNU Scientific Library gsl

The GNU Scientific Library (gsl) can be used easily in ns-3. You need to first install the libraries, then reconfigure ns-3. To install gsl on Ubuntu:

```
apt-get install gsl-bin libgsl-dev libgsl23 libgslcblas0
```

Then run ./waf configure and make sure that under the summary of ns-3 optional features, that GNU Scientific library is enabled. The ./waf build tool will pass the flags -lgsl, -lgslcblas and -lm for you.

5.2.2 C++ libraries requiring flags

If you want to use a C++ libraries that require passing compilation & link flags, you will have to reconfigure ns-3 with those flags. For example, if the code needs the -frouding-math compilation flags, and -lgmp -lCGAL link flags, then you can configure ns-3 as follows:

```
CXXFLAGS_EXTRA="-frounding-math" LDFLAGS="-lgmp -lCGAL" ./waf configure
```

5.3 Using an external ns-3 module

ns-3 built-in modules are in the src folder. There is an empty folder named contrib in the root directory of ns-3 which is intended for "*contributed*" modules. A module would be a single folder containing a python script in a file named wscript, and possibly folders named model, and examples. The examples folder contains its own wscript file.

ns-3 is in active development, so it is possible that you obtain an old module that would not work with the version of ns-3 that you have with modification, and the modification can be challenging. The best case scenario is that the contributed module was written for the version of ns-3 that you have. In this case, simply copy the module's folder to contrib, and then run ./waf configure to scan for the added modules.

Some popular external ns-3 modules are designed with a modified version of ns-3, requiring you to download an entire ns-3 source tree to avoid compatibility issues. A good example of this is mmWave, designed by New York University, which supports Millimeter wave 5G communication. In this case, you are likely going to download a compressed file composed of ns-3 with the additional modules in either the src or contrib directories. The mmwave module developed by NYU can be downloaded from https://github.com/nyuwireless-unipd/ns3-mmwave/releases/tag/v3.0

5.3.1 The ns-3 App Store

You heard it right! ns-3 has its own tiny app store. It contains additional modules that you can download and use, and most (if not all) of them are free. Every add-on is listed with the version of ns-3 that it was designed to work with. In addition, developers of those modules list instructions on how to install & use them.

Some of those modules are:

- **DOCSIS models for ns-3** designed to work with ns-3.31

- **ndnSIM** Named-Data Networking (NDN) Simulator : designed to work with `ns-3.29`
- **5G-LENA** simulates 3GPP 5G networks. It requires you to contact the developers to get the code.
- **QUIC** a native implementation of the IETF QUIC for `ns-3`. Works with `ns-3.32`
- **mmWave Cellular Network Simulator** use to evaluate cross-layer and end-to-end performance of 5G mmWave networks. Based on `ns-3.31`

You can also search the internet for `ns-3` modules as well, but keep in mind that you should use modules that are well-documented and supported by the developer. You also want to make sure that it is a reliable module that was used to create projects.

5.4 Creating **ns3** Module

If you write C++ code that you expect to reuse very often, it is more convienent to bundle it in a module that you can call from your programs under `scratch` directory. I will assume that root `ns3` directory in which `ns3` scripts are run is defined by an environment variable named **$NS3_ROOT_DIR**

Built-in `ns3` modules sources are stored under **$NS3_ROOT_DIR/src**. For example the source for the `wave` module are under **src/wave**.

5.4.1 Creating modules and **wscript** file

You may create a module by creating a directory under **$NS3_ROOT_DIR/src**, or under **$NS3_ROOT_DIR/contrib**. The file **$NS3_ROOT_DIR/src/wscript** is executed by the `./waf` script, and it lists all directories under `src`, executing their respective `wscript` scripts. Therefore, you will not need to change the `wscript` in file.

Let us say we want to create a module and call it **clemson**, under **$NS3_ROOT_DIR/contrib**

```
adil@Darwin:~$ cd /Users/adil/ns-3-allinone/bake/source/ns-3.29/contrib
adil@Darwin:contrib$ mkdir clemson
adil@Darwin:contrib$ cd clemson
```

Now let us create a C++ source & header pair. Let's call them `clemson-test.cc` and `clemson-test.h` respectively. You can place there files directly under **src/contrib/clemson**, or under a subdirectory of it.

```
#ifndef CLEMSON_TEST_H
#define CLEMSON_TEST_H
class ClemsonTest
{
  public:
    ClemsonTest(int value);
    void PrintValue();
  private:
    int m_x;
}
#endif
```

Listing 3: clemson-test.h

```
#include "clemson-test.h"
#include <iostream>
ClemsonTest::ClemsonTest(int x)
{
  m_x = x;
}
void ClemsonTest::PrintValue()
{
  std::cout << m_x << std::endl;
}
```

Listing 4: clemson-test.cc

Now, we need to create a `wscript` under **contrib/clemson**, the file has to be named `wscript` and written in `Python`. The `wscript` file format is showing in Listing 5, with a function named `build`. It defines the source and header files that will be used by the `waf` build tool. Everytime you add new source files to your module, you have to include them in the `wscript` file of your module.

In addition, the `wscript` file should specify the modules that it depends on, and assign a name to the module. You may specify whether `Python` bindings should be built as shown in the code.

With newly created modules, you need to run `./waf configure` to scan for module addition, then build it using `./waf`.

```
def build(bld):
  module_dependencies = ['wave', 'netanim']
  module = bld.create_ns3_module('clemson', module_dependencies)
  #Source files
  module.source = [
    'ClemsonTest.cc'
    ]
  #Header files
  headers = bld(features='ns3header')
  headers.module = 'clemson'
  headers.source = [
    'ClemsonTest.h'
    ]
  #If ns3 is configured with --enable-examples
  if (bld.env['ENABLE_EXAMPLES']):
    bld.recurse('examples')

  #Uncomment this to attempt to make Python bindings (optional, and may not work)
  #bld.ns3_python_bindings()
```

Listing 5: `wscript` file for `clemson` module

It is elegant to break your sources under your modules into directories with meaningful names like `model`, `helper` and `examples`. You will need to specify the relative path in your `wscript`, for example:

```
module.source = [ 'model/custom-channel-scheduler.cc',
        'helper/channel-scheduler-helper.cc',
        'example/scheduler-example.cc'
        ]
#Header files
headers = bld(features='ns3header')
headers.module = 'clemson'
headers.source = ['model/custom-channel-scheduler.h',
        'helper/channel-scheduler-helper.h'
        ]
```

`ns3` provide a `create-module.py` script under **`$NS3_ROOT_DIR/src`** that can be used to create a module skeleton under the same directory. `ns3` documentation provides help in how to use it. I personally prefer using **`$NS3_ROOT_DIR/contrib`** directory since it does not get mixed up with built-in `ns3` modules. I also think that modules built under `contrib` take less time to build than those built under `src`

5.4.2 Including your module in your `ns3` programs

You should notice that since we have created a module named `clemson` that a header file named **`clemson-module.h`** is generated. This header file includes all header files of the module, and is intended for use in programs under `scratch` directory. In fact, you do not need to include many files when you write an `ns3` program under scratch directory. You can simply include the top-level module headers, i.e. *`name`*`-module.h`

```
#include "ns3/wave-module.h"
#include "ns3/clemson-module.h"
#include "ns3/core-module.h"
```

This type of inclusion is meant to be used in your `ns3` programs under `scratch` directory. Do not use it within the modules you write, as the `ns3` documentation discourage it. You should instead include headers file that are in your module with with the relative path name, and include header files from modules that your module depends on with `ns3/file-name.h` if `file-name` is in a module that your module depends on, which would look like this:

```
#include "some-file.h" //some-file.h is in the same directory
#include "../model/another-file.h" // another-file.h is in another directory
#include "ns3/object.h" //object.h is in the 'core', which our module depend on.
```

When `ns3` builds, it makes a copy of all header files and place them in the directory `$NS3_ROOT_DIR/build/ns3`, this is why we must make sure that the module's `wscript` defines the module dependencies properly. Notice that `object.h`

is in core module, which we didn't specify in our wscript, however, our module depends on modules that depend on core and so the inclusion ns3/object.h works.

You may find that, since our module is not part of the ns3 namespace, that you will need to either call ns3 classes with the ns3:: prefix or using namespace ns3. Alternatively, you can specify that your code is part of ns3 namespace by enclosing your classes within namespace ns3 {...} in both the header and source files.

5.4.3 Using ns3 features

ns3 provides several classes that are helpful for C++ programming such as using smart pointers (ns3::Ptr).

To take advantage of ns3's smart pointer implementation (Ptr<T>), make sure that your C++ classes are a subclass of ns3::Object or its descendants. Then implement the GetTypeId and GetInstanceTypeId. Our clemson-test.h header becomes:

```
#include "ns3/nstime.h" //for ns3::Time
#include "ns3/object.h" //for ns3::Object
namespace ns3 {
class ClemsonTest : public Object
{
  public:
    static TypeId GetTypeId(void);
    virtual TypeId GetInstanceTypeId (void) const;
    ...
  private:
    Time m_interval;
    ...
}
```

In GetTypeId, a variable of type TypeId is created, and to it, we can set important parameter to it. But for starter, we need to worry about setting is:

- **ns3::ClassName** this is useful to check object's type in run-time.
- **SetParent<T>().** which is useful when we use inheritance-based function such as DynamicCast and IsChildOf.
- **AddConstructor<T>().** This is needed for when we want to use ObjectFactory to create objects in run-time.
- **AddAttribute.** this is useful when we want to use Config, but also useful when we use ObjectFactory to set certain values of the object.

So, with that the implementation source file clemson-test.cc should look like this:

```
#include "ns3/log.h" //To use ns3 logging
NS_LOG_COMPONENT_DEFINE ("ClemsonTest");
namespace ns3{
TypeId ClemsonTest::GetTypeId(void) {
  static TypeId tid = TypeId ("ns3::ClemsonTest")
    .SetParent<Object>()
    .SetGroupName ("clemson") //optional for documentation grouping
    .AddConstructor<ClemsonTest>()
    .AddAttribute ("Interval",
      "The time to wait between packets",
      TimeValue (Seconds (1.0)),
      MakeTimeAccessor (&ClemsonTest::m_interval),
      MakeTimeChecker ())
    ;
  return tid;
}
TypeId ClemsonTest::GetInstanceTypeId() const
{
  return ClemsonTest::GetTypeId();
}
...
```

Now we can use ns3's features like Create, DynamicCast, etc.

```
//create a Smart pointer object of type ClemsonTest
Ptr<ClemsonTest> ct1 = CreateObject<ClemsonTest>();

//Cast someObject to ClemsonTest
Ptr<ClemsonTest> ct2 = DynamicCast<ClemsonTest> (someObject);

//Using object factory
ObjectFactory fact;
fact.SetTypeId ( ns3::ClemsonTest::GetTypeId() );

TimeValue tv ( Seconds(2.0) );
fact.Set ( "Interval" , tv );

Ptr<ClemsonTest> ct_obh = fact.Create<ClemsonTest>();
```