NS3 Cheatbook

Adil Alsuhaim (aalsuha@clemson.edu)

February 6, 2020

Contents

1 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**.

1.1 Components of a module

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 ns3.

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 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
                  main-attribute-value.cc
                  wscript
          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 ns3 is configured with --enable-examples option.

1.2 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 1: 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 2: 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 3, 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`.

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:

```
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 3: `wscript` file for `clemson` module

```
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`

1.3 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.

1.4 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
{ ...
  static TypeId GetTypeId(void);
  virtual TypeId GetInstanceTypeId (void) const;
  ...
```

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 = Create<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" , PointerValue(tv) );

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

2 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)

2.1 Projects under scratch

It is convienent 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:

```
BandwidthQueue
        bandwidth-with-queue.cc
BareMinimumWsmp
        bare-minimum-wsmp.cc
        wsa_data.txt
DualRadio
        dual-radio-test.cc
IPv6Example
        ipv6_example.cc
```

The file dual-radio-test.cc is in the directory NS3_ROOT_DIR/scratch/DualRadio. 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.

2.2 `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`

2.3 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.

3 Working with ns3 Attributes

Classes of ns3 are written in C++, and can have attributes and trace sources. The root class 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. It provides a way to access the class's attributes, trace sources, constructors, etc.
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.

3.1 Using ns3 Attributes

If a class has .AddAttribute in its GetTypeId(), then it can be accessed using the ns3 method of accessing class information. Consider WifiMacQueue class, for example:

```
TypeId WifiMacQueue::GetTypeId (void)
{
    static TypeId tid = TypeId ("ns3::WifiMacQueue")
        .SetParent<Queue<WifiMacQueueItem> > ()
        .SetGroupName ("Wifi")
        .AddConstructor<WifiMacQueue> ()
        .AddAttribute ("MaxDelay", "If a packet stays longer than this delay in the queue, it is dropped.",
            TimeValue (MilliSeconds (500)),
            MakeTimeAccessor (&WifiMacQueue::SetMaxDelay),
            MakeTimeChecker ())
        .AddAttribute ("DropPolicy", "Upon enqueue with full queue, drop oldest (DropOldest) or newest (DropNewest) packet",
            EnumValue (DROP_NEWEST),
            MakeEnumAccessor (&WifiMacQueue::m_dropPolicy),
            MakeEnumChecker (WifiMacQueue::DROP_OLDEST, "DropOldest",
            WifiMacQueue::DROP_NEWEST, "DropNewest"))
    ;
    return tid;
}
```

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

```
        Ptr<WifiMacQueue> wifi_queue = .....; //however you get it.
        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< nodes.GetN() ; i++){
  Ptr<Node> node  = nodes.Get(0);
  /* We have no idea if device 0 is a WaveNetDevice or not.
  The right way is to loop through the NetDeviceContainer of the Node, and check type
 ↪  */
  for (uint32_t j=0 ; j<node->GetNDevice() ; j++)
  {
    Ptr<NetDevice> device = node->GetDevice(j);
    //check if this device is WaveNetDevice, because nodes can have multiple devices
    if ( device->GetInstanceTypeId() == WaveNetDevice::GetTypeId() )
    {
      Ptr<WaveNetDevice> waveDevice = DynamicCast<WaveNetDevice>(device);
      /* This only changes the queue associated with CCH. */
      Ptr<OcbWifiMac> mac = waveDevice->GetMac(CCH);
      /* The function GetVOQueue() is protected in RegularWifiMac (OcbWifiMac's parent
       ↪  class), so I can't access the queue directly. */
      PointerValue val;
      mac->GetAttribute("VO_Txop", val);
      Ptr<Txop> vo_txop = DynamicCast<Txop>(val.GetObject());
      /* We extract the WifiMacQueue from the Txop object. */
      Ptr<WifiMacQueue> vo_q = vo_txop->GetWifiMacQueue();
      /* Set the MaxDelay to 250 */
      vo_q->SetMaxDelay ( MilliSeconds(250) );
    }
  }
}
```

This looks like a lot of work, and making changes can be tedious. This only changes the value for `VO` access category, so we will need at add that in the code as well. Now, consider using the `Config` method with `Set` to set the `AttributeValue` for all ACs

```
TimeValue t;
t.Set(MilliSeconds(250));
Config::Set("/NodeList/*/DeviceList/*/$ns3::WaveNetDevice/MacEntities/*/$ns3::OcbWifiMac/*/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/" + 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::WaveNetDevice"`** A list of `NetDevice` objects attached to a node. A node can have multiple net devices, for example a `WaveNetDevice` 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 likely we can use the `TypeId` to only match the desired entities.

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

```
"/NodeList/*/DeviceList/*/MacEntities/*/$ns3::OcbWifiMac/*/Queue/MaxDelay"
```

  Going further from here, we can take a loot at the attributes defined in `WaveNetDevice` to see a few attributes like `Mtu`, `Channel`, `PhyEntities`, `MacEntities`, etc. We picked `MacEntities`!

- **`"MacEntities/*"`** The `WaveNetDevice` defined `MacEntities` as a `std::map` of value pairs corresponding to the `WAVE` channel number, and an objects of type `OcbWifiMac`, the asterix matches the values of the `OcbWifiMac` corresponding to a `WAVE` channel number.

  We used the matcher `$ns3::OcbWifiMac` to specify the type as with the case of `WaveNetDevice`, however we may also use `ns3::RegularWifiMac`, the base class of `OcbWifiMac` were the attribute queue objects are defined in 4 attributes, `VO_Txop`, `VI_Txop`, `BE_Txop`, and `BK_Txop` is defined. We may also remove `$ns3::OcbWifiMac` if we know that all `MacEntities` are of type `OcbWifiMac`, which will make the path string into:

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

  The two consecutive asterix signs match two different things in this case:

  – The channel , the channel number which can be 172, 174, 176, 178, 180, 182, and 184, corresponding to the `WAVE` channel.

  – The queue, which can be either of `VO_Txop`, `VI_Txop`, `BE_Txop`, `BK_Txop`, and also `Txop`, which is the non-QoS queue. The reason why these are matches is because they are all of `TypeId` that defines an attribute named `Queue`. The non-QoS queue is of type `Txop`, while the other four queues are of type `QosTxop` which is a subclass of `Txop`.

    If we want to avoid setting the value for the non-QoS queue, then we should specify that by only matching `$ns3::QosTxop`, the type matching the four queues.

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

- **`"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, since these strings are evaluated at run-time, there is no way to tell if they actually performed the intended purpose. If the path is incorrect, nothing would happen in run-time and it would not through a run-time error. The `Set` function does not return a value indicating success. Hence, you should be careful when you work with paths to attributes, as you may assume your changing a certain value, when you actually dont'.

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.

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

3.2 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.

3.3 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
```

3.4 Accessing Values

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

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

```
    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");
```

4 Working with Callbacks and TraceSources

In ns3, callbacks are pointers to functions. You can declare a callback as a class member and use typedef to give it a more readable name. Let us look at a usage example to illustrate usage.

4.1 Example of usage

Let us say you are running ns3 simulation that uses the wave module. You want to trace all received packets on the physical layer level and print specific information. The WifiPhy class has what we call a TraceSource called MonitorSnifferRx that gets called every time a packet is received in monitor mode. We can connect to it as follows:

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

We also need to create a function called RxPacketInfo that matches the callback defined for MonitorSnifferRx, which is defined in WifiPhy as:

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

and so we need to create RxPacketInfo in our project, with an additional first parameter of type std::string to hold to context (the resolved trace path)

```
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::WaveNetDevice/PhyEntities/0/MonitorSnifferRx"
```

Indicating that the receiver is node 0, along with details of the devices, physical entity and the name of the trace source that fired the event.

4.2 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

4.3 Understanding callbacks

Call back provides a way for user-defined functions to be invoked by other components. In ns3, we can set a Receive callback function for a NetDevice like in WaveNetDevice. For example, if we have a WaveNetDevice object called wave_device in a class called MyClass we can do:

```
...
//Set the receive callback
wave_device->SetReceiveCallback (MakeCallback  (&MyClass::MyReceiveFunction, this));
...
//define the function to be invoked
bool MyClass::MyReceiveFunction (Ptr<NetDevice> dev, Ptr <const Packet> packet, uint16_t mode, const Address & sender)
{
        //Do what you want here...
}
```

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.

In WaveNetDevice, the receive callback represents forwarding a packet up to the application layer, and is declared as

```
NetDevice::ReceiveCallback m_forwardUp;
```

The callback are used like functions, and m_forwardUp is used in WaveNetDevice

```
m_forwardUp (this, packet, llc.GetType (), from);
```

Where `this` is the `NetDevice` object, `packet` is the packet being forwarded to the application layer, `llc.GetType` is the `EtherType` of the packet, and `from` is the sender's address.

4.4 Packet Drops

For packet drops 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:

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

Not that if we were using a `WaveNetDevice` that the path is different. This is because the physical layer implementation is 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";
```

4.5 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` defines a vector of 7 MAC entities corresponding to the 7 DSRC channels, whereas `WifiNetDevice` defines a single MAC entity, 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 do to exceeding the maximum queuing delay time.

5 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.

5.1 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.

5.2 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`

6 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.

6.1 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 4: 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 5: 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.

6.2 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.

7 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]);

        }
}
```

ns3 provides a 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 ns3 program that uses this would look like:

```
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.