

Introduction to ns-3 Simulation

Adil Alsuhaime

School of Computing
Clemson University

October 5, 2020

Introduction

- ▶ ns-3 is an open-source network simulation tool
- ▶ Written in C++. Can write scripts in C++ & python.
- ▶ Can simulate simple and complex networking scenarios.

Installation - Requirements

- ▶ Installation instructions can be found on

<https://www.nsnam.org/wiki/Installation>

- ▶ **Installation Prerequisites.** The minimal requirement for ns-3 is to have C++ & python3 is installed in the system.

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

- ▶ ns-3 uses a python script to compile & run your C++ code
 - ▶ It is also possible to use clang++ instead of g++

Installation - Downloading sources

- ▶ ns-3 uses a python tool called bake to download & install ns-3
- ▶ Use git to obtain the bake tool.

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

- ▶ Change into the bake directory with `cd bake`
- ▶ Check if prerequisites are met

```
./bake.py check
```

- ▶ If the python script fails due to missing python libraries, use pip to install them.
 - ▶ pip is a python package-management system.
 - ▶ Install pip with `sudo apt-get install python3-pip`

Installation - Downloading sources

- ▶ To configure bake to download & install ns-3.31 run:

```
./bake.py configure -e ns-3.31
```

- ▶ Run `./bake.py show` to see what bake will install & download.
- ▶ Download ns-3 along with other components

```
./bake.py download
```

- ▶ Build it

```
./bake.py build
```

Installation - Run an example

- ▶ Once ns-3 download & build is finished, we need to run a configure option, then build it.

```
cd source/ns-3.31 #change into ns-3 directory  
make configure # runs './waf configure --enable-examples --enable-tests'  
./waf build
```

- ▶ Run an example.

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

- ▶ All ns-3 simulations must be run using `./waf` from the ns-3 root directory.

Some things to know

- ▶ ns-3 C++ code must be stored in files with `.cc` extension. C++ header files must have `.h` extension.
- ▶ ns-3 uses the `g++` command to compile C++ code. You can configure ns-3 to use another C++ compiler such as `clang++`. If you do that, the entire ns-3 code must be compiled again.

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

- ▶ Simulations run from the ns-3 root directory.
 - ▶ Typically, it would be called `ns-3.30` (for version 3.30) or `ns-3-dev` for the development version.
 - ▶ If your simulation writes to files, they would be stored relative to this directory.

Before you start

- ▶ Choose an IDE to write your code.
 - ▶ I recommend Visual Studio Code or Eclipse CDT (C++ support). Both are free, and available for Windows, macOS and Ubuntu.
- ▶ You can download these IDEs from the internet for free.
- ▶ Those IDEs allows us to have syntax highlighting and auto-completion features if you set them up properly with ns-3.

IDE - Visual Studio Code

- ▶ Make sure your VS Code has the C++ plug-in. Install it from the extensions tool.
- ▶ Whenever you want to code something, open VS Code from the ns-3 root directory.

```
code .
```

- ▶ This creates a hidden directory called `.vscode` in the folder
- ▶ From VS Code, press `Ctrl+Shift+P` and edit `C/C++ Configuration`. You can chose to use the UI, or edit a JSON file.
- ▶ Change `includePath` by adding the path to ns-3 build directory.

```
${workspaceFolder}/build/ns3/**  
/home/adil/ns-3.30.1/build
```

- ▶ This will result in changes to the file `./.vscode/c_cpp_properties.json`, relative to ns-3 root directory.

VS Code - c_cpp_properties.json

```
{  
    "configurations": [  
        {  
            "name": "Linux",  
            "includePath": [  
                "${workspaceFolder}/build/ns3/**",  
                "/home/adil/ns-3.30.1/build"  
            ],  
            "defines": [],  
            "compilerPath": "/usr/bin/clang",  
            "cStandard": "c11",  
            "cppStandard": "c++17",  
            "intelliSenseMode": "clang-x64"  
        }  
    ],  
}
```

Eclipse CDT - Setup for Syntax Highlighting & Auto-Completion

- ▶ Create a new C++ project, and give it a name.
- ▶ Import "File System" and import the `bake/source` directory that was created with the `bake` tool.
- ▶ Navigate to the `ns-3` folder. If you installed `ns-3.31`, then that's the `ns-3` folder.
 - ▶ Open the file `scratch/scratch-simulator.cc`
 - ▶ Check if the C++ is highlighted with colors for keywords and class names.
 - ▶ If you did not get syntax highlighting, try to re-import one directory higher or lower.
- ▶ To run `ns-3`, we typically want to use the terminal, navigate to `ns-3` root folder, and use `./waf`

Eclipse CDT - Setup for running from Eclipse (not needed/recommended)

- ▶ You can run ns-3 scripts from within Eclipse.
- ▶ Right click on the Eclipse project, and click "Properties".
 - ▶ On the left side, choose C/C++ Build
 - ▶ Uncheck "Use default build command" and set the build command to

```
${workspace_loc:/EclipseProjectName}/ns-3.30/waf
```

- ▶ Change build directory to

```
${workspace_loc:/EclipseProjectName}/ns-3.30/build
```

- ▶ In the "Behavior" tab, uncheck "Build on resource save (Auto build)"
 - ▶ For "Build (Incremental build)" leave it checked but without any text.
 - ▶ Leave Clean intact because ns-3 have a ./waf clean option.

- ▶ ns-3 uses the waf tool, which is external to Eclipse. To set it up, go to the “Run” menu, then “External Tools” > “External Tools Configuration”
 - ▶ Right-click on “Program” on the left panel, and create a “New Configuration”
 - ▶ Give it a name, and set the location to the ns-3 waf file, which would look like this

```
${workspace_loc:/EclipseProjectName}/ns-3.30/waf
```

- ▶ Set the “Working Directory” to

```
${workspace_loc:/EclipseProjectName}/ns-3.30/waf
```

- ▶ In “Arguments”, set it with the parameters:

```
--run ${string_prompt}
```

- ▶ Now when you click “Run” with this tool, Eclipse prompt you to enter the ns-3 project name.

Understanding ns-3 directory structure

- ▶ The root ns-3 directory for version 3.30 is called ns-3.30
- ▶ Under the directory, there are several important folders.
 - ▶ **src** this is the ns-3 directory where source code for modules is stores.
 - ▶ **examples** contains simple ns-3 simulation examples.
 - ▶ **contrib** Contributed modules are stored here. You do not need to do anything here unless you are creating libraries for use in ns-3
 - ▶ **build** this is the build directory for ns-3
 - ▶ **scratch** this is where user programs are stored.
- ▶ In most directories, there's a file named **wscript**, which is used by `./waf`.
 - ▶ Examine the file `examples/tutorial/wscript`. You can now see why we used `first` and not `first.cc` when we ran the example.

Understanding ns-3 directory structure

- ▶ Whenever you run `./waf` to build or run programs, all projects under **scratch** are compiled. The command fails if any project has errors.
- ▶ ns-3 comes with many examples. Relative to ns-3 root directory, there are examples under the `examples` directory. In addition, every module under `src` also contains its own `examples` directory.
- ▶ To run the example `./examples/udp/udp-echo.cc` run the following command from ns-3 root directory:

```
./waf --run examples/udp/udp-echo
```

- ▶ The example does not print anything to the terminal, but it will create output files of type `.pcap` and `.tr`
 - ▶ To examine `.pcap` files, open them with a packet analyzer like Wireshark.
 - ▶ the resulting `.tr` file is an ASCII trace of the packets.

ns-3 directory structure - Creating projects under scratch

- ▶ To create an ns-3 project, create a folder under scratch, let's say you called it Project1.
 - ▶ Under Project1, one .cc file must contain a main function
- ▶ To run Project1, you need to be at the ns-3 root directory, and run

```
./waf --run Project1
```

To pass arguments, use quotation marks as follows

```
./waf --run "Project1 --p1=10 --p2=20"
```


Running ns-3 with a debugger

- ▶ If you need to run an ns-3 program with a debugger like gdb

```
./waf --run Project1 --command-template="gdb --args %s"
```

- ▶ If you need to run it with parameters in gdb

```
./waf --run Project1 --command-template="gdb --args %s --p1=10 --p2=5"
```

- ▶ If you're using lldb as a debugger, then run it as follows

```
./waf --run Project1 --command-template="lldb %s"
```

#with parameters

```
./waf --run Project1 --command-template="lldb %s -- --p1=5 --p2=30"
```

Passing Command-Line Arguments

- ▶ ns-3 provides an interface (`ns3::CommandLine`) to handle command-line arguments.
 - ▶ It helps by setting default values if no arguments was passed.
 - ▶ You do not need to pass arguments in order.

```
int main (int argc, char *argv[])
{
    //default values
    bool student = false;
    uint32_t x = 38;
    std::string name = "Adil";
    CommandLine cmd;
    cmd.AddValue ("student", "Whether the person is a student", student);
    cmd.AddValue ("age", "Your age", x);
    cmd.AddValue ("name", "Your name", name);
    cmd.Parse (argc, argv);
    std::cout << name << " is " << x << " years old." << std::endl;
    if (student) std::cout << name << " is a student" << std::endl;
    else std::cout << name << " is NOT a student" << std::endl;
    return 0;
}
```

- ▶ We do not need to follow order of arguments

```
./waf --run "TestProject --name=Stephen --age=22 --student"
```

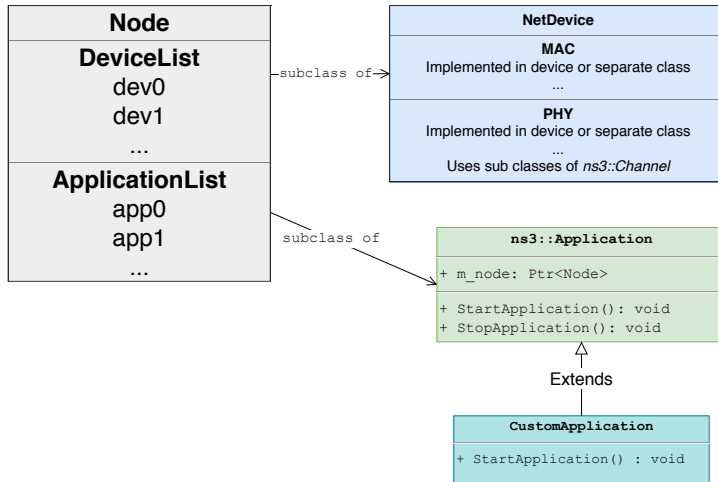
- ▶ We can supply some arguments, the rest will be their default values

```
./waf --run "TestProject --name=Stephen"
```

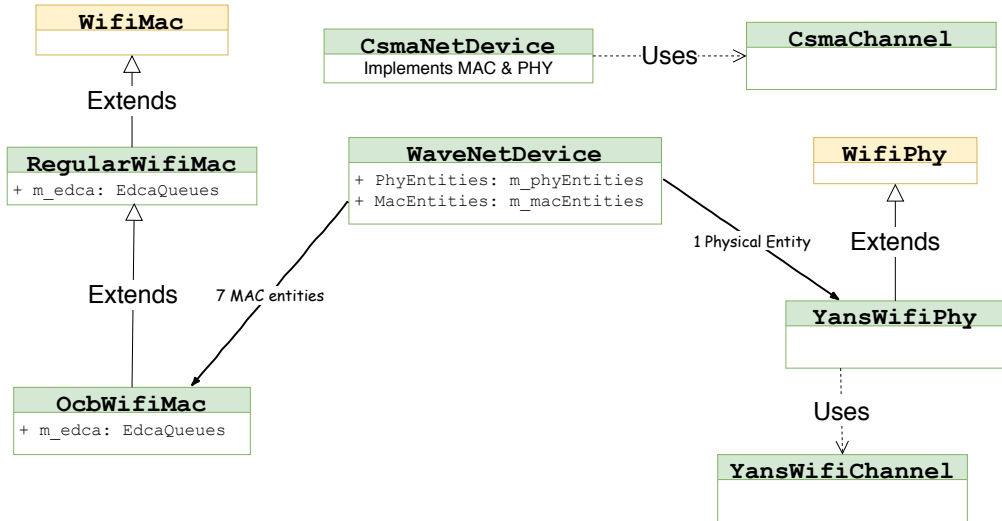
- ▶ We can also run it without any arguments. The default values will be used for all

```
./waf --run TestProject
```

Major ns-3 components



Major ns-3 components



ns-3 components - Nodes

- ▶ **Nodes.** Nodes are the main components of ns-3 simulations.
 - ▶ They represent a network host.
 - ▶ Nodes are implemented in a C++ class `ns3::Node`
 - ▶ We usually create multiple nodes using a `NodeContainer`

```
NodeContainer nodes;  
nodes.Create (10); //create 10 nodes  
//Get a pointer to the first node  
Ptr<Node> n0 = nodes.Get (0);  
//iterate over a NodeContainer  
for (uint32_t i=0 ; i<nodes.GetN() ; i++)  
{  
    Ptr<Node> ni = nodes.Get (i);  
    //...  
}
```

- ▶ Nodes can contain lists of net devices (sub-classes of `ns3::NetDevice`), and applications (sub-classes of `ns3::Application`).
 - ▶ Those can be added according to what we need to implement.

ns-3 components - Net Device

- ▶ **Net devices.** When you set up an ns-3 network simulation, you need to add net devices to nodes.
 - ▶ The class `ns3::NetDevice` is the C++ base class. It's an abstract class.
 - ▶ Sub-classes of `ns3::NetDevice` include `ns3::PointToPointNetDevice`, `ns3::CsmaNetDevice` and `ns3::WifiNetDevice`
 - ▶ ns-3 has a type `NetDeviceContainer` that can contain a list of net device objects.
- ▶ Net device implements functionalities to send & receive data.
 - ▶ We typically do not access these functionalities directly. We use sockets so that MAC & IP headers are added automatically.
- ▶ A net device should have implementation of data-link layer (MAC) and physical layer (PHY)
 - ▶ For simple net devices like `CsmaNetDevice` & `PointToPointNetDevice`, MAC & PHY are implemented within the same class.
 - ▶ For more complex devices like `WifiNetDevice`, MAC is implemented in several classes such as `WifiMac`, `RegularWifiMac`, `ApWifiMac`, and `AdhocWifiMac`. PHY is implemented in `WifiPhy` & `YansWifiPhy`.

ns-3 components - Net Device

- ▶ A node can contain multiple devices. You can iterate through them in a loop.

```
//Get all devices from node n0
Ptr<Node> n0 = nodes.Get (0);
for (uint32_t i=0 ; i<GetNDevice(); i++)
{
    Ptr<NetDevice> dev_i = n0->GetDevice (i);
    //perform type cast, other operations
    if (dev_i->GetInstanceTypeId() == CsmaNetDevice::GetInstanceTypeId())
    {
        //perform something...
    }
    else if (dev_i->GetInstanceTypeId() == WifiNetDevice::GetInstanceTypeId())
    {
        //perform something else
    }
}
```


ns-3 components - Channels

- ▶ **Channels.** define a connection between a set of nodes. Nodes that are within the same broadcast domain share one `ns3::Channel` object.
 - ▶ `ns3::Channel` is an abstract class. Concrete sub-classes include `PointToPointChannel`, `CsmaChannel` (for Ethernet), and `YansWifiChannel` for WiFi.
- ▶ For example, all nodes in a LAN share a single `CsmaChannel` object.
 - ▶ The `CsmaChannel` defines the propagation delay, and the data rate of the link.
- ▶ All WiFi nodes in a simulation should have the same `YansWifiChannel`
 - ▶ This is because a broadcast message can be received by nodes within range.

ns-3 components - Packets

- ▶ In ns-3, information are exchanged as packets, implemented in the class `Packet`.
- ▶ A packet will have headers added to it. A header is of sub-class of `ns3::Header`. Headers change a packet object.
 - ▶ Common headers are `Ipv4Header`, `ArpHeader`, `EthernetHeader`
 - ▶ You can define your own headers, but that's not recommended.
- ▶ We can also add tags to a packet. A tag is a of a sub-class of `ns3::Tag`. Adding a tag does not change the packet object. You can only add one tag of a given type to a packet.
 - ▶ Tags are used to exchange information in simulations.
 - ▶ You may need to define your own tags.

ns-3 components - Packets

- ▶ Many applications that you may use in ns-3 create packets hidden from the users.
- ▶ To create a zero payload packet:

```
Ptr<Packet> packet = Create <Packet> ();
```

- ▶ You can create a packet with a payload of zeros (100 bytes)

```
Ptr<Packet> packet = Create <Packet> (100);
```

- ▶ You can create a packet with a specific payload data.

```
std::string word = "Some text as payload";  
char const *cWord = word.c_str();  
uint8_t *data = new uint8_t[word.size()];  
std::memcpy( data , cWord, word.size());  
Ptr<Packet> packet = Create<Packet> (data, word.size());
```

ns-3 components - Packets

- ▶ When you run your simulation, you can obtain a string representation of a packet. To do this, you need to first enable printing on packets

```
Packet::EnablePrinting (); //Must be done before you start your simulation, or you'll get an error
```

- ▶ Then you can use `ToString()`, which only works if you enabled metadata. To get a string representation of a `Packet` object named `p`

```
std::cout << p->ToString () << std::endl;
```

- ▶ You can examine & remove packet headers using `PeekHeader` and `RemoveHeader`. This can be tricky if you want to get the expected results, as you need to examine/remove headers in order.

ns-3 components - Packets

- ▶ If packet *p* is received, and you are sure that it begins with an `EthernetHeader`, you can examine it as follows:

```
EthernetHeader ethernet;  
if (p->PeekHeader (ethernet)) //works if there's enough data to cover an EthernetHeader (14 bytes or more)  
{  
    std::cout << "Src MAC: " << ethernet.GetSource () << " Dest MAC: " << ethernet.GetDestination () << std::endl;  
}
```

- ▶ This is a packet without header. `PeekHeader` works, but doesn't give the expected results.

```
Ptr<Packet> p = Create <Packet> (100); //packet of size 100 bytes, no headers.  
EthernetHeader hdr;  
if (p->PeekHeader (hdr))  
{  
    std::cout << "Peeking worked, but it gives wrong results" << std::endl;  
}
```

ns-3 components - Packet Tags

- ▶ Tags are a convenient way to carry information in packets, without changing packets' data or headers.
- ▶ A tag that can be added to a packet has to be a sub-class of `ns3::Tag`.
- ▶ Let us say we have create a tag in a class named `MyDataTag` that carries a single real value, and two public function `SetValue()` and `GetValue()`. We will add a tag to a packet `p` as follows:

```
MyDataTag t;  
t.SetValue (22.5);  
p->AddPacketTag (t);
```

- ▶ We can extract that information as follows:

```
MyDataTag t;  
if (p->PeekPacketTag (t)){  
    std::cout << "Packet Info: " << t.GetValue () << std::endl;  
}
```

ns-3 components - Applications

- ▶ **Applications.** We can add applications to a node. An application must have a start & stop time.
- ▶ Applications added are sub-classes of `ns3::Application`
 - ▶ You can create your own application as a subclass of `ns3::Application` written in C++.
- ▶ An application has a pointer to the node's pointer.
 - ▶ This implies that it also has a pointer to the list of net device.
- ▶ A list of applications can be contained in a `ApplicationContainer`.

ns-3 components - Mobility

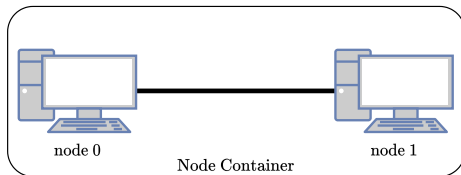
- ▶ **Mobility.** specifies the mobility of a node. A node can be stationary, or moving following different kinds of models.
- ▶ When we install mobility to a node, we aggregate a `ns3::MobilityModel` object to it.
 - ▶ `MobilityModel` is an abstract class. Some sub-classes of it are `ConstantPositionMobilityModel` (for stationary nodes), `ConstantVelocityMobilityModel` and `RandomWayPointMobilityModel`
- ▶ Nodes that use wireless communication must have a mobility model installed to them, even if the node is stationary.
 - ▶ This is because the wireless signal is influenced by the distance between sender & receiver.

Object Factory & Helpers

- ▶ In Software Engineering, there is a design pattern known as *factory* or *object factory* design pattern.
 - ▶ It helps construct objects with initial and/or default values for the object's attributes.
- ▶ ns-3 uses a class called `ns3::ObjectFactory` to construct objects. Usually the code is not used directly by users, as users would use *Helper* classes to construct objects.
- ▶ Helpers are a convenient way to configure nodes. However, you can still configure nodes manually if you want more control.

Example - Using PointToPointHelper

- ▶ To create a point-to-point link between two nodes, we use `PointToPointHelper`
 - ▶ This installs a `PointToPointNetDevice` to nodes. It sets the data rate to 5 Mbps.
 - ▶ It also creates a `PointToPointChannel` object. It sets the propagation delay to 2ms.



```
NodeContainer nodes;  
nodes.Create (2);  
  
PointToPointHelper pointToPoint;  
pointToPoint.SetDeviceAttribute ("DataRate", StringValue  
↳ ("5Mbps"));  
pointToPoint.SetChannelAttribute ("Delay", StringValue  
↳ ("2ms"));  
  
NetDeviceContainer devices;  
devices = pointToPoint.Install (nodes);
```

Example - Setup IP Networking

- ▶ We use `InternetStackHelper` to setup IP networking for a set of nodes.
- ▶ We use `Ipv4AddressHelper` to set up IPv4 addressing.
- ▶ We store the IP interfaces in a `Ipv4InterfaceContainer`

```
InternetStackHelper stack;  
stack.Install (nodes);  
  
Ipv4AddressHelper address;  
address.SetBase ("10.1.1.0", "255.255.255.0");  
  
Ipv4InterfaceContainer interfaces = address.Assign  
↳ (devices);
```

Example - Using UdpEchoServerHelper

- ▶ The UdpEchoServerHelper is used to install a UdpEchoServer, which is a C++ sub-class of ns3::Application, to node 1
 - ▶ it listens on port number 9999.
 - ▶ The application starts after 1 second from the start of simulation.
 - ▶ It stops after 10 seconds of start of simulation.

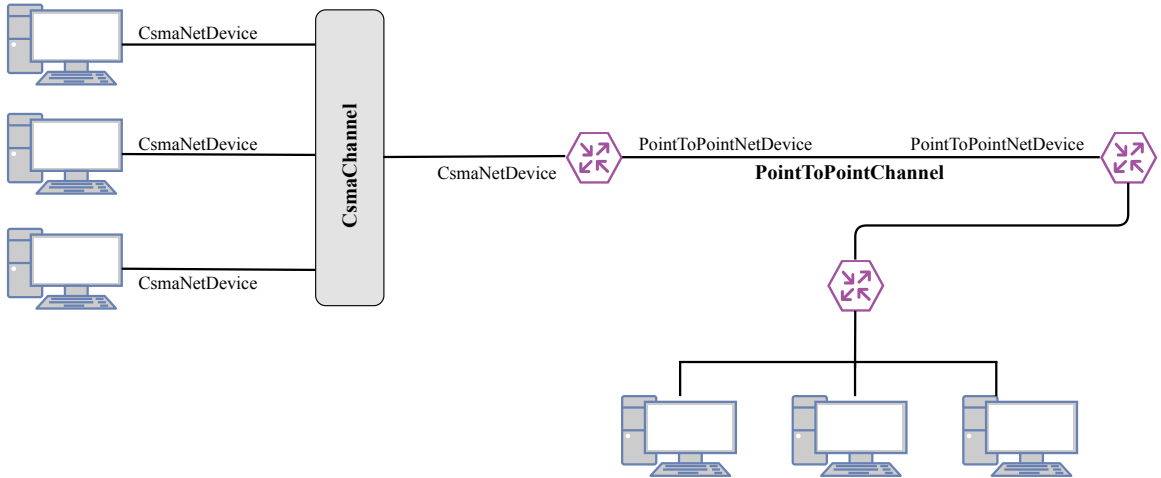
```
UdpEchoServerHelper echoServer (9999);  
  
ApplicationContainer serverApps = echoServer.Install  
↳ (nodes.Get (1));  
serverApps.Start (Seconds (1.0));  
serverApps.Stop (Seconds (10.0));
```

Example - Using UdpEchoClientHelper

- ▶ The UdpEchoServerHelper is used to install a UdpEchoClient to node 0.
- ▶ Notice how we set certain attributes, such as MaxPackets, Interval and PacketSize
 - ▶ These are used internally by an ObjectFactory.
- ▶ The echo client uses the IP address of node 1, and targets port number 9999

```
UdpEchoClientHelper echoClient (interfaces.GetAddress (1),  
    ↪ 9999);  
echoClient.SetAttribute ("MaxPackets", UIntegerValue (1));  
echoClient.SetAttribute ("Interval", TimeValue  
    ↪ (MicroSeconds(100)));  
echoClient.SetAttribute ("PacketSize", UIntegerValue  
    ↪ (1024));  
  
ApplicationContainer clientApps = echoClient.Install  
    ↪ (nodes.Get (0));  
clientApps.Start (Seconds (2.0));  
clientApps.Stop (Seconds (10.0));
```

Example - Lan



Code

Get the code <https://github.com/addola/NS3-HelperScripts/tree/master/examples/ThreeRouters>

```
//For the first network
NodeContainer lan1_nodes;
//For the second network
NodeContainer lan2_nodes;
//for the nodes in the middle.
NodeContainer router_nodes;

lan1_nodes.Create (n1);
lan2_nodes.Create (n2);
router_nodes.Create (3);
//Add router 1 to LAN 1
lan1_nodes.Add (router_nodes.Get (0));

CsmaHelper csma1; //LAN 1 attributes
csma1.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
csma1.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));
//Actually attaching CsmaNetDevice (Ethernet) to all LAN 1 nodes, including Router 1
NetDeviceContainer lan1Devices;
lan1Devices = csma1.Install (lan1_nodes);
```

Code

```
//Doing the same for LAN 2: Router 3 is on LAN 2, so we add it to the node container
lan2_nodes.Add (router_nodes.Get (1));
CsmaHelper csma2;
csma2.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
csma2.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));
//Actually install CsmaNetDevice to LAN 2, including Router 3
NetDeviceContainer lan2Devices;
lan2Devices = csma2.Install (lan2_nodes);

//Let's connect R1<-->R2 and R2<-->R3
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("10Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
//Install PointToPointNetDevice for R1 & R2
NetDeviceContainer firstHopLinkDevs;
firstHopLinkDevs = pointToPoint.Install (router_nodes.Get(0), router_nodes.Get(2));
//Install PointToPointNetDevice for R2 & R3
NetDeviceContainer secondHopLinkDevs;
secondHopLinkDevs = pointToPoint.Install (router_nodes.Get(2), router_nodes.Get(1));
```


Code - Internet stack, and IP addresses

```
//Setting IP addresses.
InternetStackHelper stack;
stack.Install (lan1_nodes);
stack.Install (lan2_nodes);
//R1 & R3 are in LAN1 & LAN2, so they already have internet stack. R3 doesn't yet, so we install it to it.
stack.Install (router_nodes.Get(2));
//Setting IPv4 For LAN 1
address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer lan1interfaces;
lan1interfaces = address.Assign (lan1Devices);
//Setting IPv4 For LAN 2
address.SetBase ("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer lan2interfaces;
lan2interfaces = address.Assign (lan2Devices);
//For PointToPoint
address.SetBase ("10.1.100.0", "255.255.255.0");
Ipv4InterfaceContainer routerInterfaces;
routerInterfaces = address.Assign (firstHopLinkDevs);

address.SetBase ("10.1.200.0", "255.255.255.0");
Ipv4InterfaceContainer routerInterfaces2;
routerInterfaces = address.Assign (secondHopLinkDevs);
```

Code - Install applications, Populate Routing Tables

```
UdpEchoServerHelper echoServer (9);
ApplicationContainer serverApps = echoServer.Install (lan2_nodes);
serverApps.Start (Seconds (0));
serverApps.Stop (Seconds (10));

//Let's create UdpEchoClients in all LAN1 nodes.
UdpEchoClientHelper echoClient (lan2interfaces.GetAddress (0), 9);
echoClient.SetAttribute ("MaxPackets", UIntegerValue (100));
echoClient.SetAttribute ("Interval", TimeValue (Milliseconds (200)));
echoClient.SetAttribute ("PacketSize", UIntegerValue (1024));

//We'll install UdpEchoClient on two nodes in lan1 nodes
NodeContainer clientNodes (lan1_nodes.Get(0), lan1_nodes.Get(1));

ApplicationContainer clientApps = echoClient.Install (clientNodes);
clientApps.Start (Seconds (1));
clientApps.Stop (Seconds (10));

//For routers to be able to forward packets, they need to have routing rules.
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
```

Code - Using helpers to enable tracing

```
//Enable packet capture (*.pcap) files
//use lan1 as a prefix
csma1.EnablePcap("lan1", lan1Devices);
//use lan2 as a prefix
csma2.EnablePcap("lan2", lan2Devices);
//use routers as prefix
pointToPoint.EnablePcapAll("routers");
//Enable ASCII-based tracing
pointToPoint.EnableAscii("ascii-p2p", router_nodes);
```

ns-3 Smart Pointers

- ▶ ns-3 define numerous C++ classes, with `ns3::Object` & `ns3::SimpleRefCount` being the base class of most of them.
 - ▶ This allows ns-3 to implement smart pointer functionalities, using `ns3::Ptr<T>`
- ▶ For a class that is a sub-class of `ns3::Object`, we can create objects using `CreateObject<T>` function.
 - ▶ For example, to create an object of type `ns3::Node` using its default (parameterless) constructor the code will be:

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

- ▶ The `Packet` class does not inherit from `ns3::Object`. To create a `Packet` with a payload of size 200 bytes (all zeros), we will use `Create<T>` function as follows.

```
Ptr<Packet> packet = Create <Packet> (200);
```

ns-3 Type cast & Copy operations

- ▶ ns-3 provides a way to perform type casting as well. For example, to cast from NetDevice named devObj to WifiNetDevice, we do:

```
Ptr <NetDevice> dev = node_ptr->GetDevice (0);  
Ptr <WifiNetDevice> wifi_dev = DynamicCast <WifiNetDevice> (dev);
```

- ▶ To copy an object, we will use the functions Copy & CopyObject depending on whether the object we want to copy is of a type that is a subclass of ns3::Object

```
//for a class that is a sub-class of ns3::Object  
Ptr <Node> copy_node = CopyObject (original_node);  
  
//for a class that is a sub-class of ns3::SimpleRefCount  
Ptr <Packet> pkt = Create <Packet> (200);  
Ptr <Packet> pkt_copy = pkt->Copy ();
```

Scheduling an Event

```
#include "ns3/core-module.h"
using namespace ns3;
void MyFunction ()
{
    std::cout << "Time: " << Now () << std::endl;
}
void ParameterFunction (int x, int y)
{
    std::cout << "Time: " << Now() << " Result: " << (x*y) << std::endl;
}
int main(int argc, char *argv[])
{
    CommandLine cmd;
    cmd.Parse(argc, argv);
    Simulator::Schedule(MilliSeconds(200), &MyFunction);
    Simulator::Schedule(Seconds (3.5), &ParameterFunction, 2, 13);
    Simulator::Stop (Seconds (20));
    Simulator::Run();
    Simulator::Destroy();
    return 0;
}
```

Scheduling an Event - In another class

If you have a C++ class called MyClass

```
class MyClass : public ns3::Object
{
public:
    void Fun1 ();
    void Fun2 (double value);
};

void MyClass::Fun1 ()
{
    std::cout << Now () << " Fun1" << std::endl;
}

void MyClass::Fun2 (double value)
{
    std::cout << Now () << " Fun2: value=" << value <<
    ↵ std::endl;
}
```

You schedule events in main by using an object to the class.

```
int main (int argc, char **argv)
{
    //...
    Ptr <MyClass> obj = Create<MyClass> ();
    Simulator::Schedule (Seconds (1.5), &MyClass::Fun1,
    ↵ obj);
    Simulator::Schedule (Seconds (3.5), &MyClass::Fun2, obj,
    ↵ 33.1);
    //...
}
```

ns-3 Implementation of reflections

- ▶ Most ns-3 C++ classes implement the functions `GetTypeId` & `GetInstanceTypeId`
- ▶ In `GetTypeId`, the type is registered as `ns3::TypeId`, along with:
 - ▶ *Attributes*. which are attributes of a given class. Attributes are inherited by subclasses.
 - ▶ *TraceSources*. which represent an event that is triggered during the simulation. You can connect them to a callback function to handle what happens when the event occurs.
- ▶ `GetInstanceTypeId` is used to inspect the run-time type of an object.
(*Polymorphism in Object-Oriented Programming*)
- ▶ We can use the *Attributes* and *TraceSources* using the `Config` namespace

Config namespace

- ▶ The helper constructs a `UdpEchoClient` application, sets its attributes, and install it at a node.
- ▶ Notice that we used string values `MaxPackets`, `Interval` and `PacketSize`.
- ▶ These string values corresponds to *attributes* of the `UdpEchoClient` class.

```
UdpEchoClientHelper echoClient (interfaces.GetAddress (1),  
    ↪ 9999);  
echoClient.SetAttribute ("MaxPackets", UIntegerValue (1));  
echoClient.SetAttribute ("Interval", TimeValue  
    ↪ (MicroSeconds(100)));  
echoClient.SetAttribute ("PacketSize", UIntegerValue  
    ↪ (1024));  
  
ApplicationContainer clientApps = echoClient.Install  
    ↪ (nodes.Get (0));  
clientApps.Start (Seconds (2.0));  
clientApps.Stop (Seconds (10.0));
```

Config namespace - Attributes

- ▶ Let us say we want to change the packet size after the UdpEchoClient application was created. We do this, as one would expect

```
Ptr<Node> n0 = nodes.Get (0);  
//iterate through all applications in node 0  
for (uint32_t i=0 ; i < n0->GetNApplications(); i++)  
{  
    Ptr<Application> app_i = n0->GetApplication (i);  
    //Check if it's safe to perform type cast  
    if (app_i->GetInstanceTypeId () == UdpEchoClient::GetTypeId())  
    {  
        Ptr<UdpEchoClient> udp_cli = DynamicCast <UdpEchoClient> (app_i); //type cast  
        udp_cli->SetDataSize (512); //actually set the value.  
    }  
}
```

Config namespace - Change Attributes

- ▶ Instead of using a loop, we may use the Config namespace provided by ns-3 to set attributes in fewer lines of code

```
Config::Set("/NodeList/*/ApplicationList*/$ns3::UdpEchoClient/PacketSize", UIntegerValue(512));
```

- ▶ The string is called *path* to an attribute.
 - ▶ The first asterisk means it will match all nodes in the global list of nodes, NodeList.
- ▶ If you look at the documentation for ns3::Node and the code node.cc, you will notice that the function GetTypeId() defines some attributes
 - ▶ **ApplicationList** a vector of applications installed in a node.
 - ▶ **DeviceList** a vector of all devices installed in a node.
- ▶ The second asterisk means match all applications in a node, and the following \$ns3::UdpEchoClient specify the matching type.
- ▶ Finally, PacketSize is an attribute of \$ns3::UdpEchoClient

Config namespace - More Examples


- ▶ Change the delay of all `PointToPointChannel` objects to three seconds.

```
Config::Set("/ChannelList/*/ns3::PointToPointChannel/Delay", TimeValue (Seconds (3)));
```

- ▶ ns-3 maintains a global list of all `Channel` objects in the simulation.
- ▶ **Delay** is an attribute in `PointToPointChannel`
- ▶ Another example to `GetAttribute` from an object pointer (without full path):

```
Ptr<UdpEchoClient> udp_client = DyanmicCast<UdpEchoClient> (clientApps.Get(0));  
UIntegerValue val;  
udp_client->GetAttribute ("PacketSize", val);  
std::cout << val.Get() << std::endl;
```

List of attributes - Doxygen



A Discrete-Event Network Simulator
ns-3-dev @ 4d1387c(+)

HOME | TUTORIALS ▼ | DOCS ▼ | DEVELOP ▼

Main Page | Related Pages | Modules | Namespaces ▼ | Classes ▼ | Files ▼

Q Search

ns-3

- ns-3 Documentation
- All ns3::TypeId's
- All Attributes
- All GlobalValues
- All LogComponents
- All TraceSources
- Todo List
- Deprecated List
- Bug List
- Modules
- Namespaces
- Classes
- Files

ns3::UdpClient

- MaxPackets:** The maximum number of packets the application will send
- Interval:** The time to wait between packets
- RemoteAddress:** The destination Address of the outbound packets
- RemotePort:** The destination port of the outbound packets
- PacketSize:** Size of packets generated. The minimum packet size is 12 bytes which is the size of the header carrying the sequence number and the time stamp.

ns3::UdpEchoClient

- MaxPackets:** The maximum number of packets the application will send
- Interval:** The time to wait between packets
- RemoteAddress:** The destination Address of the outbound packets
- RemotePort:** The destination port of the outbound packets
- PacketSize:** Size of echo data in outbound packets

ns3::UdpEchoServer

- Port:** Port on which we listen for incoming packets.

Attribute definition in GetTypeId()

```
TypeId
UdpClient::GetTypeId (void)
{
    static TypeId tid = TypeId ("ns3::UdpClient")
        .SetParent<Application> ()
        .SetGroupName ("Applications")
        .AddConstructor<UdpClient> ()
        .AddAttribute ("MaxPackets",
            "The maximum number of packets the application will send",
            UIntegerValue (100),
            MakeUIntegerAccessor (&UdpClient::m_count),
            MakeUIntegerChecker<uint32_t> ())
        .AddAttribute ("Interval",
            "The time to wait between packets", TimeValue (Seconds (1.0)),
            MakeTimeAccessor (&UdpClient::m_interval),
            MakeTimeChecker ())
    ...
}
```

Config namespace - TraceSources

- ▶ TraceSources causes a function with a specific signature to be called. The user can write code to handle this event.
- ▶ They are accessible with a *path* string, however, we need to create a callback function to handle the event.
 - ▶ To figure out the signature of this callback, look at the source code in which the TraceSource is define

```
void ClientTx (std::string context, Ptr<const Packet> packet)
{
    //what happens when UdpEchoClient start Tx
}


int main (int argc, char *argv[])
{
    //...
    Config::Connect ("/NodeList/*/ApplicationList/*/$ns3::UdpEchoClient/Tx", MakeCallback(&ClientTx));
    //...
}
```

More examples

- ▶ When we use `Connect`, we are not specifying a *context*, so it must be the first argument of the callback.
 - ▶ `context` string is the match for the regular expression (i.e. the path string).
 - ▶ `/NodeList/3/ApplicationList/0/$ns3::UdpEchoClient/Tx` means that application id 0 in node id 3 matched the call.
- ▶ If you have a pointer to an object of a given type, you can connect without context and without the full path.

```
void ClientTx (Ptr<const Packet> packet) //notice no context string
{
    //code to handle the event
}
int main (int argc, char *argv[])
{
    //...
    Ptr<UdpEchoClient> udp_client = DynamicCast <UdpEchoClient> (clientApps.Get(0));
    udp_client->TraceConnectWithoutContext ("Tx", MakeCallback(&ClientTx));
    //...
}
```


Config namespace - TraceSources

 A Discrete-Event Network Simulator
ns-3-dev @ 4d1387c(+)

HOME | TUTORIALS ▼ | DOCS ▼ | DEVELOP ▼

API

Main Page | Related Pages | Modules | Namespaces ▼ | Classes ▼ | Files ▼

Q Search

ns-3

ns-3 Documentation

All ns3::TypeId's

All Attributes

All GlobalValues

All LogComponents

All TraceSources

Todo List

Deprecated List

Bug List

Modules

Namespaces

Classes

Files

ns3::PointToPointNetDevice

- **MacTx**: Trace source indicating a packet has arrived for transmission by this device
- **MacTxDrop**: Trace source indicating a packet has been dropped by the device before transmission
- **MacPromiscRx**: A packet has been received by this device, has been passed up from the physical layer and is being forwarded up the local protocol stack. This is a promiscuous trace,
- **MacRx**: A packet has been received by this device, has been passed up from the physical layer and is being forwarded up the local protocol stack. This is a non-promiscuous trace,
- **PhyTxBegin**: Trace source indicating a packet has begun transmitting over the channel
- **PhyTxEnd**: Trace source indicating a packet has been completely transmitted over the channel
- **PhyTxDrop**: Trace source indicating a packet has been dropped by the device during transmission
- **PhyRxEnd**: Trace source indicating a packet has been completely received by the device
- **PhyRxDrop**: Trace source indicating a packet has been dropped by the device during reception
- **Sniffer**: Trace source simulating a non-promiscuous packet sniffer attached to the device
- **PromiscSniffer**: Trace source simulating a promiscuous packet sniffer attached to the device

Config namespace - TraceSources

- ▶ Here's a snippet of `GetTypeId()` function's code in `point-to-point-net-device.cc`.
- ▶ `MacTx` is a `TraceSource` we would like to connect to.

```
...  
.AddAttribute ("TxQueue",  
    "A queue to use as the transmit queue in the device.",  
    PointerValue (),  
    MakePointerAccessor (&PointToPointNetDevice::m_queue),  
    MakePointerChecker<Queue<Packet> > ())  
.AddTraceSource ("MacTx",  
    "Trace source indicating a packet has arrived "  
    "for transmission by this device",  
    MakeTraceSourceAccessor (&PointToPointNetDevice::m_macTxTrace),  
    "ns3::Packet::TracedCallback")  
...
```

- ▶ The signature of the callback function is specified by
`PointToPointNetDevice::m_macTxTrace`

TraceSource - Callback signature

- For PointToPointNetDevice, check the file point-to-point-net-device.h, and find the line:

```
TracedCallback<Ptr<const Packet> > m_macTxTrace;
```

- We can see that the required signature is Ptr<const Packet>.

```
void MyTrace (Ptr<const Packet> p) { /*Config::ConnectWithoutContext*/}  
void MyTrace (std::string context, Ptr<const Packet> p) { /*Config::Connect*/}
```

Another Example

- ▶ Let's say we want to access the RxWithAddresses TraceSource in UdpEchoClient. Check the code in udp-echo-client.cc

```
.AddTraceSource ("RxWithAddresses", "A packet has been received",  
    MakeTraceSourceAccessor (&UdpEchoClient::m_rxTraceWithAddresses),  
    "ns3::Packet::TwoAddressTracedCallback")
```

- ▶ Check the declaration for **m_rxTraceWithAddresses** in udp-echo.client.h

```
TracedCallback<Ptr<const Packet>, const Address &, const Address &> m_rxTraceWithAddresses;
```

- ▶ Create a callback function with matching signature, adding std::string context if connecting with context.

```
void MyTrace (Ptr<const Packet> p, const Address& a, const Address& b) { /*Config::ConnectWithoutContext*/}  
void MyTrace (std::string context, Ptr<const Packet> p, const Address& a, const Address& b) {  
↳ /*Config::Connect*/}
```

TraceSource - MacTx in PointToPointNetDevice

```
// Trace source to handle MacTx event
void MacTxTrace(std::string context, Ptr<const Packet> pkt)
{
    std::cout << context << std::endl;
    std::cout << "\tMacTX Size=" << pkt->GetSize() << "\t" << Simulator::Now() << std::endl;
}

int main (int argc, char **argv)
{
    //... After creating the devices, connect to the trace source
    Config::Connect ("/NodeList/*/DeviceList/*/ns3::PointToPointNetDevice/MacTx", MakeCallback(&MacTxTrace))
    //...
}
```

► **context** is the matching the regular expression used in `Config::Connect`

Failsafe function

- ▶ When using `Config` to use *Attributes* and *TraceSources*, a run-time error may occur if it was improper.
 - ▶ Usually, when the signature of the callback function does not match the required signature.
- ▶ You can use `SetAttributeFailSafe` & `GetAttributeFailSafe` functions, which return `false` if it failed to set/get the attribute.
- ▶ If you have `ns-3.31`, you can use `ConnectFailSafe` to connect to a `TraceSource`. It will return `false` if it fails.

Logging

- ▶ You might have developed the habit of printing out to the terminal to inspect how your code is working.
 - ▶ It helps investigate whether certain code is reached.
 - ▶ It can be used to print intermediary values.
- ▶ However, if you want stop printing you either have to remove the printing statements, or commenting them out.
 - ▶ It takes time!
- ▶ ns-3 provides a way to enable/disable print statement with its Logging functionality.
- ▶ ns-3 components can be defined as a log component for which printing can be enabled & disabled on multiple levels.

Example

- ▶ Check out the code in `./src/examples/tutorial/first.cc`, notice the two lines:

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

which enables logging at the INFO level on these two components.

- ▶ Try to change `LOG_LEVEL_INFO` to `LOG_LEVEL_ALL`, then run it again. You will notice that you are logging function calls in the two components.
- ▶ Enabling `LOG_LEVEL_INFO` also enables levels of higher severity such as `LOG_LEVEL_WARN` and `LOG_LEVEL_ERROR`

Log level severity

Level	Meaning
LOG_LEVEL_ERROR	Only log LOG_ERROR severity
LOG_LEVEL_WARN	Only log LOG_WARN severity and above
LOG_LEVEL_DEBUG	Only log LOG_DEBUG severity and above
LOG_LEVEL_INFO	Only log LOG_INFO severity and above
LOG_LEVEL_FUNCTION	Only log LOG_FUNCTION severity and above
LOG_LEVEL_LOGIC	Only log LOG_LOGIC severity and above
LOG_LEVEL_ALL	All severity class.

Logging example

- ▶ Another way to enable logging is through the command-line. The following command enables all logging levels for `UdpEchoClientApplication`

```
NS_LOG=UdpEchoClientApplication ./waf --run examples/tutorials/first
```

- ▶ We can enable logging a single log-level from the command line.
 - ▶ Let's enable ALL on `UdpEchoClientApplication`.
 - ▶ Let's also enable only INFO on `UdpEchoServerApplication`

```
NS_LOG="UdpEchoClientApplication:UdpEchoServerApplication=info" ./waf  
↪ --run examples/tutorial/first
```

- ▶ When you enable logging from the command-line, you only enable the specified level.
- ▶ When no level is specified, the default level is `all`

Logging in your programs

- ▶ Save this file under
./scratch/TestLogging/test.cc
- ▶ Run it **./waf -run TestLog**
 - ▶ Notice that you get WARN & ERROR, but not INFO
- ▶ Now comment out the statement that enable logging, and instead, use the command-line

```
NS_LOG=MyBeautifulProgram=warn ./waf --run  
↪ TestLog
```

- ▶ notice how this way only enabled logging on the WARN level and not the higher level.

```
#include "ns3/core-module.h"  
using namespace ns3;  
//define this file as a logging component.  
NS_LOG_COMPONENT_DEFINE ("MyBeautifulProgram");  
void PrintMe (int x, int y)  
{  
    NS_LOG_FUNCTION (x << y);  
}  
int main (int argc, char *argv[])  
{  
    CommandLine cmd;  
    cmd.Parse (argc, argv);  
    LogComponentEnable ("MyBeautifulProgram",  
        ↪ LOG_LEVEL_WARN);  
    NS_LOG_UNCOND ("Will always be printed");  
    NS_LOG_INFO ("This is an info message!");  
    NS_LOG_WARN ("This is a Warning message");  
    NS_LOG_ERROR ("This is an error messages");  
    PrintMe (10,22);  
    return 0;  
}
```

Log prefix

- ▶ ns-3 can add prefixes to log messages. For example, adding the function name before the log message. Let us say we want to enable all levels for our components, but we want the node name as a prefix.

```
NS_LOG="MyBeautifulProgram=all|func" ./waf --run TestProject
```

- ▶ Let's try level INFO on our ./src/examples/tutorial/first.cc from the command line with some prefixes.

```
NS_LOG="UdpEchoClientApplication=info|func:UdpEchoServerApplication=info|time" ./waf --run  
↩ examples/tutorial/first
```

- ▶ The prefix can be node, time, level or all. The all prefix is the default.
 - ▶ all prints out a prefix of current simulation time (in seconds), followed by the node id (if possible, or -1), then the function in which the log statement is, then the log level followed by the log message

Log prefix

- ▶ Finally, to appreciate the work done by ns-3 try enabling logging on all components with the asterisk.

```
NS_LOG="*=info|all" ./waf --run examples/tutorial/first
```

- ▶ And even more details

```
NS_LOG="*=debug|all" ./waf --run examples/tutorial/first
```

- ▶ or every single log message:

```
NS_LOG="*=all|all" ./waf --run examples/tutorial/first
```

Visualization

- ▶ There are two types of visualization in ns-3.
 - ▶ **netanim** which works offline on an XML file generated from your simulation. netanim is a stand-alone application.
 - ▶ **Pyviz** which works online, and uses python.
 - ▶ You need to make sure that when you run `./waf configure` that PyViz visualizer is enabled. Otherwise, install the required dependencies.
- ▶ To make your simulation generate an XML file for netanim, add the lines

```
int main (int argc, char *argv[]) {  
    //...  
    Packet::EnableMetadata();  
    AnimationInterface anim ("animation_output.xml");  
    //...  
}
```

Visualization

- ▶ To run your simulation with PyViz, make sure your main function uses CommandLine because running in PyViz involves passing an argument:

```
int main (int argc, char *argv[]) {  
    //define default command-line arguments  
    CommandLine cmd;  
    //add some command-line arguments with cmd.AddValue  
    cmd.Parse (argc, argv);  
}
```

- ▶ To run in PyViz, we pass `-vis`

```
./waf --run "SomeProgram" --vis
```

- ▶ If you need to pass arguments, you run it as

```
./waf --run "SomeProgram --n1=3 --n2=10" --vis
```

Example : UDP Socket

```
void ReceiveHandler(Ptr<Socket> socket) {
    Ptr<Packet> packet;
    Address from;
    while ((packet = socket->RecvFrom(from)))
        NS_LOG_INFO(Now () << "Socket Recv: Content: " << packet->ToString());
}

int main (int args, char **argv) {
    //...
    TypeId tid = TypeId::LookupByName("ns3::UdpSocketFactory");
    Ptr<Socket> send_socket = Socket::CreateSocket(nodes.Get(0), tid);
    Ptr<Socket> recv_socket = Socket::CreateSocket(nodes.Get(1), tid);
    InetSocketAddress local = InetSocketAddress(Ipv4Address::GetAny(), port);
    if (recv_socket->Bind(local) == -1)
        NS_FATAL_ERROR("Failed to bind socket");
    Ipv4Address destination = interfaces.GetAddress(1); //IP of node 1.
    send_socket->Connect(InetSocketAddress(Ipv4Address::ConvertFrom(destination), port));
    //Handle reception at socket-level (packet will have no headers)
    recv_socket->SetRecvCallback(MakeCallback(&ReceiveHandler));
    //...
}
```


UDP Socket

```
void SendPacket (Ptr<Socket> socket, Ptr<Packet> packet) {
    socket->Send(packet);
    //send another packet after 1 second.
    Simulator::Schedule (Seconds (1), &SendPacket, socket, packet);
}

void MacRxTrace(std::string context, Ptr<const Packet> packet) {
    NS_LOG_INFO("MacRxTrace : " << packet->ToString());
}

int main (int args, char **argv) {
    //...
    //Enables packet->ToString() to give details about content
    Packet::EnablePrinting ();
    //Create and send a packet
    Ptr<Packet> packet = Create <Packet> (512);
    Simulator::Schedule (Seconds (2.0), &SendPacket, send_socket, packet);
    //Handle reception at MAC level using Config
    Config::Connect ("/NodeList/*/DeviceList*/$ns3::PointToPointNetDevice/MacRx", MakeCallback (&MacRxTrace));
    Simulator::Stop (Seconds(10));
    Simulator::Run ();
    Simulator::Destroy ();
    return 0;
}
```

Application Example

- ▶ Create a C++ class as a sub-class of ns3::Application
- ▶ Override the StartApplication () function. Provide code to run when an application starts.
- ▶ From within the class, you have access to the node with GetNode () function

```
#ifndef MY_UDP_APPLICATION_H
#define MY_UDP_APPLICATION_H
#include "ns3/socket.h"
#include "ns3/application.h"
using namespace ns3;
class SimpleUdpApplication : public ns3::Application
{
public:
    SimpleUdpApplication ();
    virtual ~SimpleUdpApplication ();

    void SendPacket (Ptr<Packet> packet, Ipv4Address
        ↪ destination, uint16_t port);
    void ReceivePacket (Ptr<Socket> socket);
private:
    virtual void StartApplication ();

    Ptr<Socket> m_socket;
    uint16_t m_port;
};
#endif
```

Adding application to node

```
int main (int argc, char **argv) {  
    // ...  
    Ptr <SimpleUdpApplication> udp0 = CreateObject <SimpleUdpApplication> ();  
    Ptr <SimpleUdpApplication> udp1 = CreateObject <SimpleUdpApplication> ();  
    //Set the start & stop times  
    udp0->SetStartTime (Seconds(0));  
    udp0->SetStopTime (Seconds (10));  
    udp1->SetStartTime (Seconds(0));  
    udp1->SetStopTime (Seconds (10));  
    //install one application at node 0, and the other at node 1  
    nodes.Get(0)->AddApplication (udp0);  
    nodes.Get(1)->AddApplication (udp1);  
    //This is the IP address of node 1  
    Ipv4Address dest_ip ("10.1.1.2");  
    //Schedule an event to send a packet of size 400 using udp0 targeting IP of node 0, and port 7777  
    Ptr <Packet> packet1 = Create <Packet> (400);  
    Simulator::Schedule (Seconds (1), &SimpleUdpApplication::SendPacket, udp0, packet1, dest_ip, 7777);  
    //...  
}
```

Application example

- ▶ You can find the complete example code on my github

`https://github.com/addola/NS3-HelperScripts/tree/master/
examples/SimpleUdpAppExample`

Creating custom packet tags

- ▶ Let us create a packet tag that carries the following information (*attributes*)
 - ▶ **node id.** (**uint32_t**) the id of the node from which this tag originates.
 - ▶ **Time.** (**ns3::Time**) the timestamp when this tag was created.
 - ▶ **Position.** (**ns3::Vector3D**) The x, y, z coordinates of the node when this tag was created.
- ▶ When you create a sub-class of **ns3::Tag** you need to create the following functions:
 - ▶ **uint32_t GetSerializedSize (void) const** which should return the serialized size of your tag. For this example, the serialized size will be the number of bytes needed to represent node id, timestamp and position, or:

```
sizeof (uint32_t) + sizeof(ns3::Time) + sizeof (Vector3D)
```

- ▶ **void Serialize (TagBuffer i) const** convert the attributes to binary.
- ▶ **void Deserialize (TagBuffer i)** convert from binary and set the value of the attributes.
- ▶ **void Print (std::ostream &os) const** output a string representation to output stream. Used in ASCII traces.

Creating custom packet tags

```
class CustomDataTag : public Tag {
public:
    CustomDataTag();
    static TypeId GetTypeId(void);
    virtual TypeId GetInstanceTypeId(void) const;
    //Functions inherited from ns3::Tag that you have to implement.
    virtual uint32_t GetSerializedSize(void) const;
    virtual void Serialize (TagBuffer i) const;
    virtual void Deserialize (TagBuffer i);
    virtual void Print (std::ostream & os) const;
    //These are custom accessor & mutator functions
    Vector GetPosition(void);
    uint32_t GetNodeId();
    Time GetTimestamp ();
    void SetPosition (Vector pos);
    void SetNodeId (uint32_t node_id);
    void SetTimestamp (Time t);
private:
    uint32_t m_nodeId;
    Vector m_currentPosition;
    Time m_timestamp;
};
```

Creating custom packet tags

```
uint32_t CustomDataTag::GetSerializedSize (void) const {
    return sizeof(Vector) + sizeof (ns3::Time) + sizeof(uint32_t);
}

/** The order of how you do Serialize() should match the order of Deserialize() */
void CustomDataTag::Serialize (TagBuffer i) const {
    i.WriteDouble(m_timestamp.GetDouble());
    i.WriteDouble (m_currentPosition.x);
    i.WriteDouble (m_currentPosition.y);
    i.WriteDouble (m_currentPosition.z);
    i.WriteU32 (m_nodeId);
}

/** This function reads data from a buffer and store it in class's instance variables. */
void CustomDataTag::Deserialize (TagBuffer i){
    m_timestamp = Time::FromDouble (i.ReadDouble(), Time::NS);
    m_currentPosition.x = i.ReadDouble();
    m_currentPosition.y = i.ReadDouble();
    m_currentPosition.z = i.ReadDouble();
    m_nodeId = i.ReadU32();
}

void CustomDataTag::Print (std::ostream &os) const {
    os << "Custom Data --- Node : " << m_nodeId << "\t(" << m_timestamp << ")" << " Pos (" << m_currentPosition << "));"
}
```