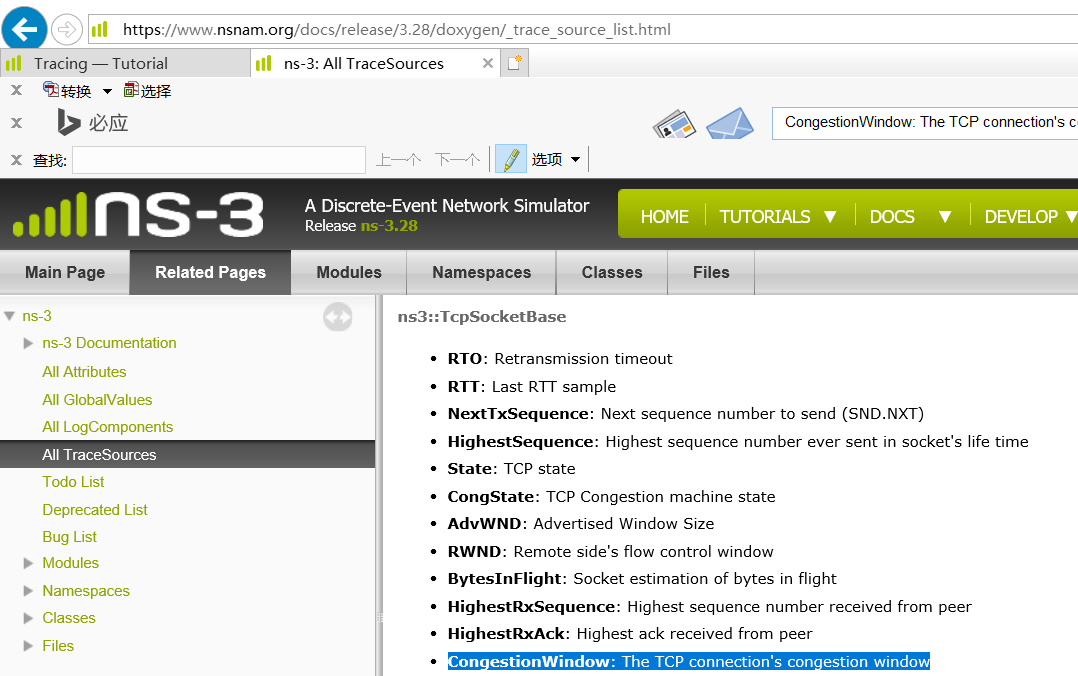
**Lab5 and Lab6: TCP Congestion Window**

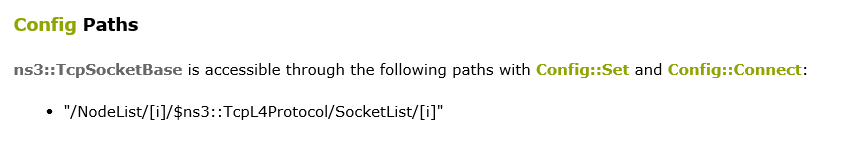
1. In this tutorial, we are going to reproduce the following figure, “Figure 21.10. Value of cwnd and send sequence number while data is being transmitted” from the classic book “TCP/IP Illustrated, Volume 1: The Protocols,” by W. Richard Stevens.



**Step 1: Determine the trace source**

The first thing to think about is how we want to get the data out. Browsing the ns-3 API documentation <https://www.nsnam.org/docs/release/3.28/doxygen/_trace_source_list.html>, we find



Cliking on **ns3::TcpSocketBase** on this page brings up the class documentation for **TcpSocketBase,** scroll down to “Detailed Description”, you will find **Config Paths**

and

**TraceSources**

……

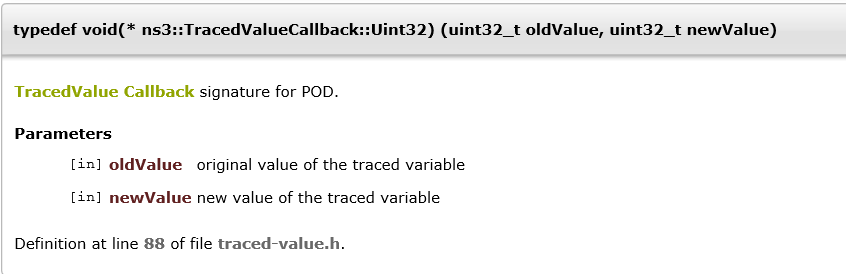
* CongestionWindow: The TCP connection's congestion window

Callback signature: ns3::TracedValueCallback::Uint32

……

**Step 2: Determine the return type (always void) and arguments of call back function**

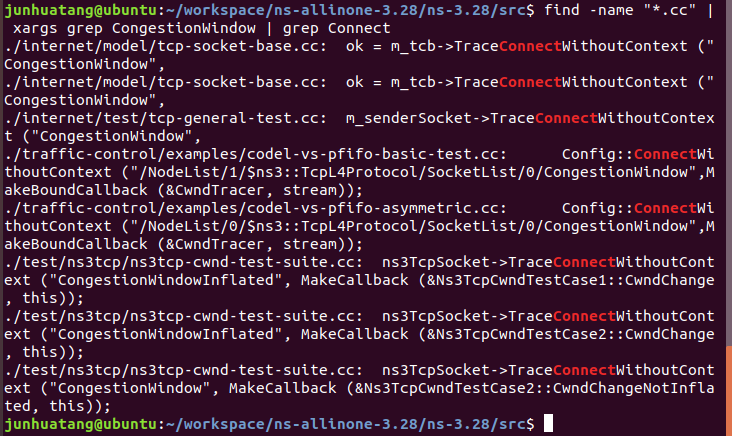
Clicking on the Callback signature above will give us the return type and arguments of the callback function:



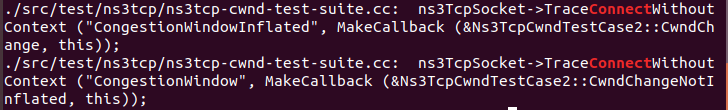
**Step 3: Determine how to connect the trace source to the call back function**

It’s always best to try and find working code laying around that you can modify, rather than starting from scratch. So the first order of business now is to find some code that already hooks the “CongestionWindow” trace source and see if we can modify it. As usual, grep is your friend:

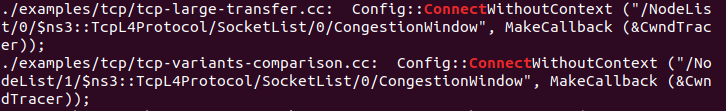
$find -name “\*.cc” | xargs grep CongestionWindow | grep Connect



Method 1: using SocketObject->TraceConnect(……) (We will focus on this method in this tutorial)



Method 2: using Config::Connect(…… ) (This method will be addressed in lab 6)



1. Walking through fifth.cc

In this tutorial we use SocketObject->TraceConnect(……) to connect the trace source and trace sink.

Open src/test/ns3tcp/ns3tcp-cwnd-test-suite.cc in your favorite editor and search for “CongestionWindow”. You will find,



This should look very familiar to you. We mentioned above that if we had a pointer to the TcpSocketBase, we could TraceConnect to the “CongestionWindow” trace source. That’s exactly what we have here; so it turns out that this line of code does exactly what we want.

Let’s go ahead and extract the code we need from this file, and that results in a native ns-3 script – examples/tutorial/fifth.cc..

The fifth.cc example demonstrates an extremely important rule that you must understand before using any kind of trace source: you must ensure that the target of a Config::Connect command exists before trying to use it. This is no different than saying an object must be instantiated before trying to call it. Although this may seem obvious when stated this way, it does trip up many people trying to use the system for the first time.

There are three basic execution phases that exist in any ns-3 script.

* The first phase is sometimes called “Configuration Time” or “Setup Time,” and exists during the period when the main function of your script is running, but before Simulator::Run is called.
* The second phase is sometimes called “Simulation Time” and exists during the time period when Simulator::Run is actively executing its events.
* After it completes executing the simulation, Simulator::Run will return control back to the main function. When this happens, the script enters what can be called the “Teardown Phase,” which is when the structures and objects created during setup are taken apart and released.

Perhaps the most common mistake made in trying to use the tracing system is assuming that entities constructed dynamically during simulation time are available during configuration time. In particular, an ns-3 Socket is a dynamic object often created by Applications to communicate between Nodes. An ns-3 Application always has a “Start Time” and a “Stop Time” associated with it. In the vast majority of cases, an Application will not attempt to create a dynamic object until its StartApplication method is called at some “Start Time”. This is to ensure that the simulation is completely configured before the app tries to do anything (what would happen if it tried to connect to a Node that didn’t exist yet during configuration time?). As a result, during the configuration phase you can’t connect a trace source to a trace sink if one of them is created dynamically during the simulation.

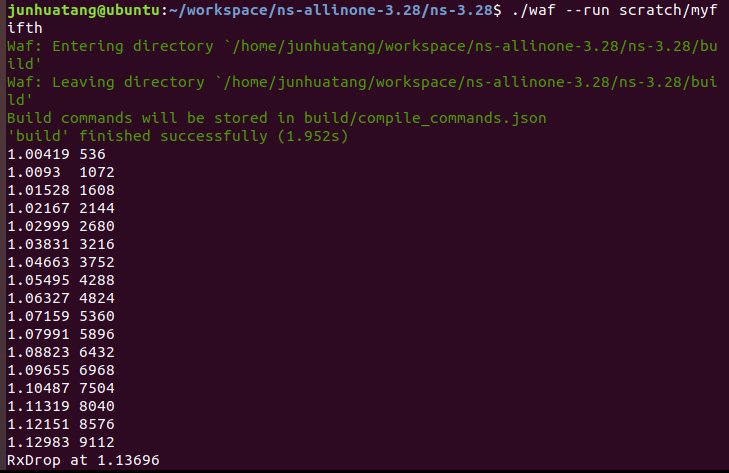
The two solutions to this conundrum are

1. Create a simulator event that is run after the dynamic object is created and hook the trace when that event is executed; or
2. Create the dynamic object at configuration time, hook it then, and give the object to the system to use during simulation time.

We took the second approach in the fifth.cc example. This decision required us to create the MyApp Application, the entire purpose of which is to take a Socket as a parameter.

$cp examples/tutorial/fifth.cc scratch/myfifth.cc

$./waf --run scratch/myfifth



Let’s redirect that output to a file called cwnd.dat:

$ ./waf --run fifth > cwnd.dat 2>&1

Now edit up “cwnd.dat” in your favorite editor and remove the waf build status and drop lines, leaving only the traced data.

You can now run gnuplot (if you have it installed) and tell it to generate some pretty pictures:

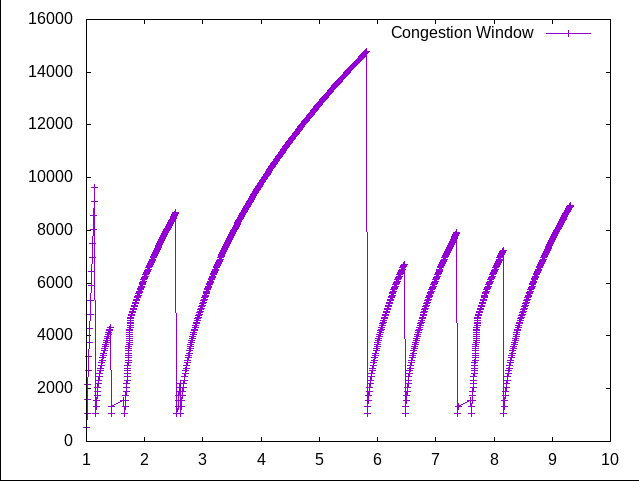
$ gnuplot

gnuplot> set terminal png size 640,480

gnuplot> set output "cwnd.png"

gnuplot> plot "cwnd.dat" using 1:2 title 'Congestion Window' with linespoints

gnuplot> exit



1. Walking through sixth.cc

One of the most important things we want to do is to is to have the ability to easily control the amount of output coming out of the simulation; and we also want to save those data to a file so we can refer back to it later. We can use the mid-level trace helpers provided in *ns-3* to do just that and complete the picture.

sixth.cc is a script that writes the cwnd change and drop events developed in the example fifth.cc to disk in separate files. The cwnd changes are stored as a tab-separated ASCII file and the drop events are stored in a PCAP file.

The changes to make this happen are quite small. Actually this can be done in only 18 lines of code.

static void

CwndChange (Ptr<OutputStreamWrapper> stream, uint32\_t oldCwnd, uint32\_t newCwnd)

{

NS\_LOG\_UNCOND (Simulator::Now ().GetSeconds () << "\t" << newCwnd);

\*stream->GetStream () << Simulator::Now ().GetSeconds () << "\t" << oldCwnd << "\t" << newCwnd << std::endl;

}

...

AsciiTraceHelper asciiTraceHelper;

Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("sixth.cwnd");

ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow", MakeBoundCallback (&CwndChange, stream));

...

static void

RxDrop (Ptr<PcapFileWrapper> file, Ptr<const Packet> p)

{

NS\_LOG\_UNCOND ("RxDrop at " << Simulator::Now ().GetSeconds ());

file->Write(Simulator::Now(), p);

}

...

PcapHelper pcapHelper;

Ptr<PcapFileWrapper> file = pcapHelper.CreateFile ("sixth.pcap", "w", PcapHelper::DLT\_PPP);

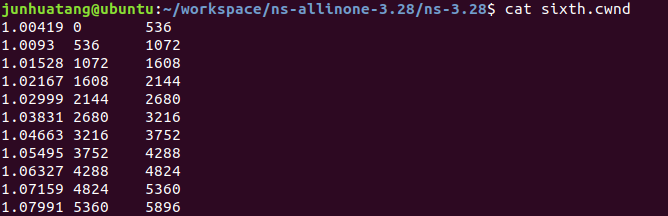
devices.Get (1)->TraceConnectWithoutContext("PhyRxDrop", MakeBoundCallback (&RxDrop, file));

Run sixth.cc and see the results.

$cp examples/tutorial/sixth.cc scratch/mysixth.cc

$ ./waf --run sixth

Since “sixth.cwnd” is an ASCII text file, you can view it with cat or your favorite file viewer.



Since “sixth.pcap” is a PCAP file, you can fiew it with tcpdump.

junhuatang@ubuntu:~/workspace/ns-allinone-3.28/ns-3.28$ tcpdump -nn -tt -r sixth.pcap

reading from file sixth.pcap, link-type PPP (PPP)

1.136956 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 17177:17681, ack 1, win 32768, options [TS val 1133 ecr 1127,eol], length 504: HTTP

1.403196 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 33280:33784, ack 1, win 32768, options [TS val 1399 ecr 1394,eol], length 504: HTTP

1.436476 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 37440:37944, ack 1, win 32768, options [TS val 1432 ecr 1428,eol], length 504: HTTP

2.533823 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 174184:174720, ack 1, win 32768, options [TS val 2530 ecr 2525,eol], length 536: HTTP

2.543036 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 175760:176264, ack 1, win 32768, options [TS val 2539 ecr 2534,eol], length 504: HTTP

2.608703 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 183544:184080, ack 1, win 32768, options [TS val 2605 ecr 2600,eol], length 536: HTTP

5.804476 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 583440:583944, ack 1, win 32768, options [TS val 5800 ecr 5796,eol], length 504: HTTP

6.453436 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 664560:665064, ack 1, win 32768, options [TS val 6449 ecr 6445,eol], length 504: HTTP

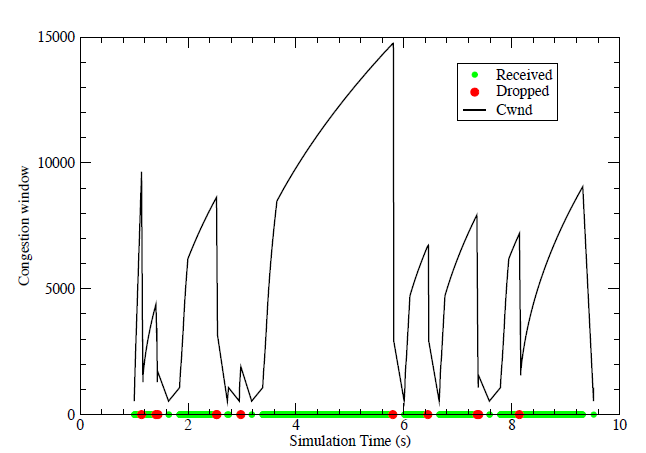
7.367743 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 778424:778960, ack 1, win 32768, options [TS val 7364 ecr 7360,eol], length 536: HTTP

7.393596 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 782080:782584, ack 1, win 32768, options [TS val 7389 ecr 7386,eol], length 504: HTTP

8.158143 IP 10.1.1.1.49153 > 10.1.1.2.8080: Flags [.], seq 877224:877760, ack 1, win 32768, options [TS val 8155 ecr 8149,eol], length 536: HTTP

junhuatang@ubuntu:~/workspace/ns-allinone-3.28/ns-3.28$

1. Lab5.cc :Produce a figure showing TCP congestion window size, dropped packets and received packets, like in the following figure. Use the same network configuration as in mysixth.cc.



Hints:

1. Modify mysixth.cc
2. Use trace source

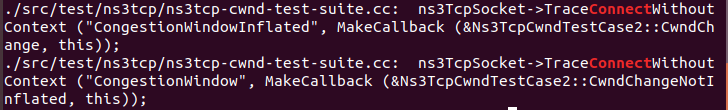
[ns3::PointToPointNetDevice](https://www.nsnam.org/docs/release/3.28/doxygen/classns3_1_1_point_to_point_net_device.html)

* + **PhyRxEnd**: Trace source indicating a packet has been completely received by the device

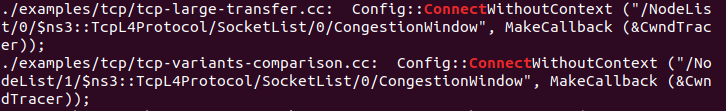
1. Lab6.cc

Remember there are two approaches to connect your trace sink to the CongestionWindow trace source.

Method 1: using SocketObject->TraceConnect(……)



Method 2: using Config::Connect(…… )



We have used method 1 in lab5, use method 2 in lab6.cc to get the value of TCP congestion window.

Hint:

1. Use a two-node point-to-pint topology. Use BulkSendApplication on 1 node, and PacketSink on the other node. (refer to examples/tcp/tcp-bulk-send.cc)
2. Use the link error model as in mysixth.cc
3. To use Config::Connect (……) , you need to create a simulator event that is run after the dynamic object is created and hook the trace when that event is executed. Refer to /examples/tcp/tcp-variants-comparison.cc for more details.

Turn in：

1. lab5.cc, lab6.cc
2. A short report lab5-6-report
   1. showing your simulation results
   2. any thoughts on the exercises