**Experiment N0: 03**

**Name of Experiments: TCP Variants**

**Objective**:

1. Create a simple dumbbell topology, two client Node1 and Node2 on the left side of the

dumbbell and server nodes Node3 and Node4 on the right side of the dumbbell. Let Node5

and Node6 form the bridge of the dumbbell. Use point to point links.

2. Install a TCP socket instance on Node1 that will connect to Node3.

3. Install a UDP socket instance on Node2 that will connect to Node4.

4. Start the TCP application at time 1s.

5. Start the UDP application at time 20s at rate Rate1 such that it clogs half the dumbbell

bridge's link capacity.

6. Increase the UDP application's rate at time 30s to rate Rate2 such that it clogs the whole of

the dumbbell bridge's capacity.

7. Use the ns-3 tracing mechanism to record changes in congestion window size of the TCP

instance over time. Use gnuplot/matplotlib to visualise plots of cwnd vs time.

8. Mark points of fast recovery and slow start in the graphs.

9. Perform the above experiment for TCP variants Tahoe, Reno and New Reno, all of which are

available with ns-3.

**Source Code:**

#include <fstream>

#include "ns3/core-module.h"

#include "ns3/network-module.h"

#include "ns3/internet-module.h"

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

#include "ns3/applications-module.h"

using namespace ns3;

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

//

===========================================================================

//

// node 0 node 1

// +----------------+ +----------------+

// | ns-3 TCP | | ns-3 TCP |

// +----------------+ +----------------+

// | 10.1.1.1 | | 10.1.1.2 |

// +----------------+ +----------------+

// | point-to-point | | point-to-point |

// +----------------+ +----------------+

// | |

// +---------------------+

// 5 Mbps, 2 ms

//

//

// We want to look at changes in the ns-3 TCP congestion window. We need

// to crank up a flow and hook the CongestionWindow attribute on the socket

// of the sender. Normally one would use an on-off application to generate a

// flow, but this has a couple of problems. First, the socket of the on-off

// application is not created until Application Start time, so we wouldn't be

// able to hook the socket (now) at configuration time. Second, even if we

// could arrange a call after start time, the socket is not public so we

// couldn't get at it.

//

// So, we can cook up a simple version of the on-off application that does what

// we want. On the plus side we don't need all of the complexity of the on-off

// application. On the minus side, we don't have a helper, so we have to get

// a little more involved in the details, but this is trivial.

//

// So first, we create a socket and do the trace connect on it; then we pass

// this socket into the constructor of our simple application which we then

// install in the source node.

//

===========================================================================

//

class MyApp : public Application

{

public:

MyApp ();

virtual ~MyApp();

void Setup (Ptr<Socket> socket, Address address, uint32\_t packetSize, uint32\_t nPackets,

DataRate dataRate);

private:

virtual void StartApplication (void);

virtual void StopApplication (void);

void ScheduleTx (void);

void SendPacket (void);

Ptr<Socket> m\_socket;

Address m\_peer;

uint32\_t m\_packetSize;

uint32\_t m\_nPackets;

DataRate m\_dataRate;

EventId m\_sendEvent;

bool m\_running;

uint32\_t m\_packetsSent;

};

MyApp::MyApp ()

: m\_socket (0),

m\_peer (),

m\_packetSize (0),

m\_nPackets (0),

m\_dataRate (0),

m\_sendEvent (),

m\_running (false),

m\_packetsSent (0)

{

}

MyApp::~MyApp()

{

m\_socket = 0;

}

void

MyApp::Setup (Ptr<Socket> socket, Address address, uint32\_t packetSize, uint32\_t nPackets,

DataRate dataRate)

{

m\_socket = socket;

m\_peer = address;

m\_packetSize = packetSize;

m\_nPackets = nPackets;

m\_dataRate = dataRate;

}

void

MyApp::StartApplication (void)

{

m\_running = true;

m\_packetsSent = 0;

m\_socket->Bind ();

m\_socket->Connect (m\_peer);

SendPacket ();

}

void

MyApp::StopApplication (void)

{

m\_running = false;

if (m\_sendEvent.IsRunning ())

{

Simulator::Cancel (m\_sendEvent);

}

if (m\_socket)

{

m\_socket->Close ();

}

}

void

MyApp::SendPacket (void)

{

Ptr<Packet> packet = Create<Packet> (m\_packetSize);

m\_socket->Send (packet);

if (++m\_packetsSent < m\_nPackets)

{

ScheduleTx ();

}

}

void

MyApp::ScheduleTx (void)

{

if (m\_running)

{

Time tNext (Seconds (m\_packetSize \* 8 / static\_cast<double> (m\_dataRate.GetBitRate ())));

m\_sendEvent = Simulator::Schedule (tNext, &MyApp::SendPacket, this);

}

}

static void

CwndChange (uint32\_t oldCwnd, uint32\_t newCwnd)

{

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

}

static void

RxDrop (Ptr<const Packet> p)

{

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

}

int

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

{

CommandLine cmd;

cmd.Parse (argc, argv);

NodeContainer nodes;

nodes.Create (2);

PointToPointHelper pointToPoint;

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

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

NetDeviceContainer devices;

devices = pointToPoint.Install (nodes);

Ptr<RateErrorModel> em = CreateObject<RateErrorModel> ();

em->SetAttribute ("ErrorRate", DoubleValue (0.00001));

devices.Get (1)->SetAttribute ("ReceiveErrorModel", PointerValue (em));

InternetStackHelper stack;

stack.Install (nodes);

Ipv4AddressHelper address;

address.SetBase ("10.1.1.0", "255.255.255.252");

Ipv4InterfaceContainer interfaces = address.Assign (devices);

uint16\_t sinkPort = 8080;

Address sinkAddress (InetSocketAddress (interfaces.GetAddress (1), sinkPort));

PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory", InetSocketAddress

(Ipv4Address::GetAny (), sinkPort));

ApplicationContainer sinkApps = packetSinkHelper.Install (nodes.Get (1));

sinkApps.Start (Seconds (0.));

sinkApps.Stop (Seconds (20.));

Ptr<Socket> ns3TcpSocket = Socket::CreateSocket (nodes.Get (0), TcpSocketFactory::GetTypeId

());

ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow", MakeCallback

(&CwndChange));

Ptr<MyApp> app = CreateObject<MyApp> ();

app->Setup (ns3TcpSocket, sinkAddress, 1040, 1000, DataRate ("1Mbps"));

nodes.Get (0)->AddApplication (app);

app->SetStartTime (Seconds (1.));

app->SetStopTime (Seconds (20.));

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

Simulator::Stop (Seconds (20));

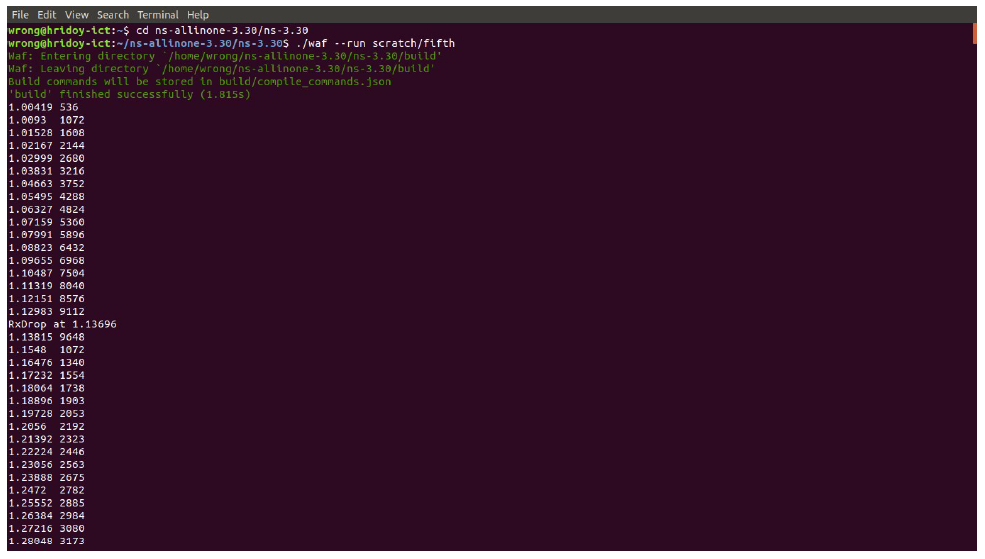
Simulator::Run ();

Simulator::Destroy ();

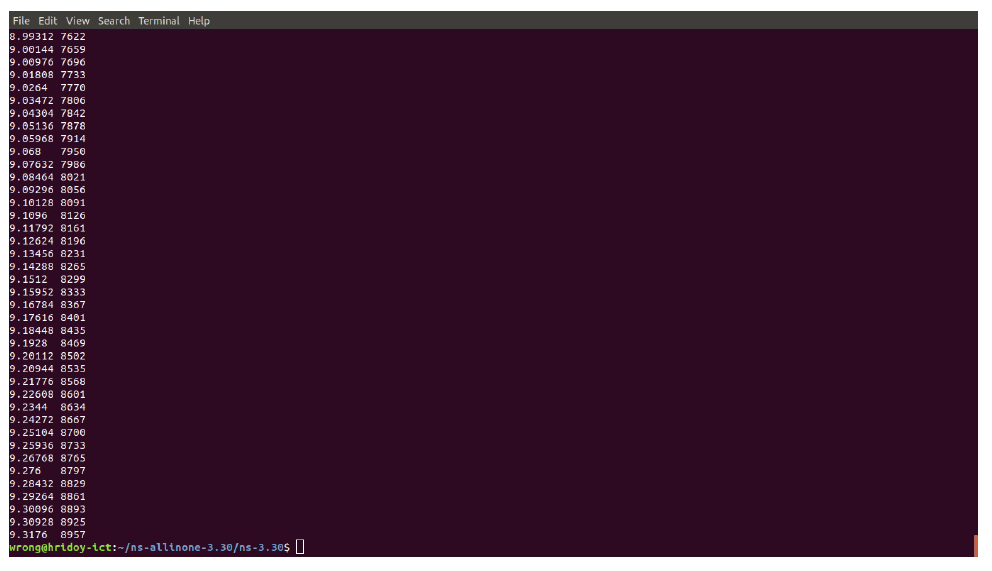
return 0;

}

**OUTPUT :**







**Conclusion:**

From this experience, we have learnt to create dumbbell topology, installing TCP and UDP socket. We use the ns-3 tracing mechanism to record changes in congestion window size of the TCP instance over time and gnuplot/matplotlib to visualise plots of cwnd vs time. Congestion and packet loss will interact with TCP’s connectivity and timing issues, and ultimately the overall performance of the network.