**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
//
```

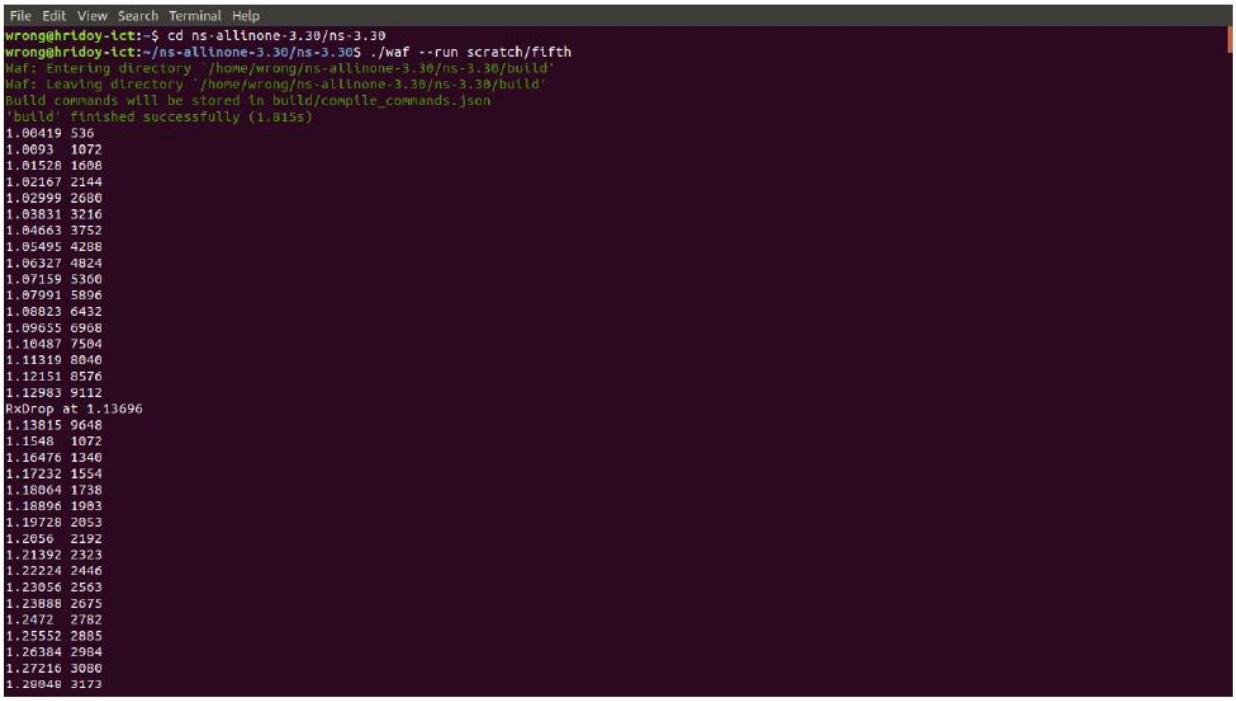
```
//
// We want to look at changes in the ns-3 TCP congestion window. We need
// to crank up a flow and hook the CongestionWindow attribute on the socket
// of the sender. Normally one would use an on-off application to generate a
// flow, but this has a couple of problems. First, the socket of the 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)
```

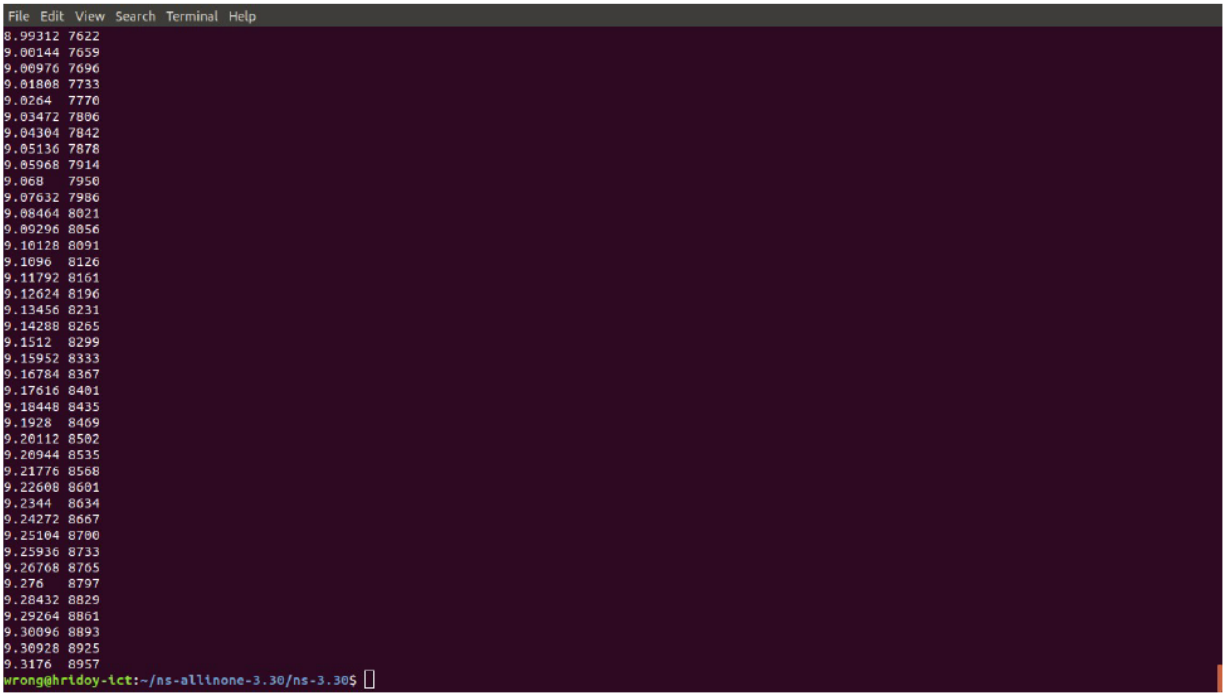
```
{
Ptr<Packet> packet = Create<Packet> (m_packetSize);
m_socket->Send (packet);
if (++m_packetsSent < m_nPackets)
{
ScheduleTx ();
}
}
void
MyApp::ScheduleTx (void)
{
if (m_running)
{
Time tNext (Seconds (m_packetSize * 8 / static_cast<double> (m_dataRate.GetBitRate ())));
m_sendEvent = Simulator::Schedule (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 :**




File Edit View Search Terminal Help
2.91952 4787
2.92784 4768
2.93616 4828
2.94448 4887
2.9528 4945
2.96112 5003
2.96944 5060
2.97776 5116
2.98608 5172
2.9944 5227
3.00272 5281
3.01104 5335
3.01936 5388
3.02768 5441
3.036 5493
3.04432 5545
3.05264 5596
3.06096 5647
3.06928 5697
3.0776 5747
3.08592 5796
3.09424 5845
3.10256 5894
3.11088 5942
3.1192 5990
3.12752 6037
3.13584 6084
3.14416 6131
3.15248 6177
3.1608 6223
3.16912 6269
3.17744 6314
3.18576 6359
3.19408 6404
3.2024 6448
3.21072 6492
3.21904 6536
3.22736 6579
3.23568 6622
3.244 6665
3.25232 6708


File Edit View Search Terminal Help
8.99312 7622
9.00144 7659
9.00976 7696
9.01808 7733
9.0264 7770
9.03472 7806
9.04304 7842
9.05136 7878
9.05968 7914
9.068 7950
9.07632 7986
9.08464 8021
9.09296 8056
9.10128 8091
9.1096 8126
9.11792 8161
9.12624 8196
9.13456 8231
9.14288 8265
9.1512 8299
9.15952 8333
9.16784 8367
9.17616 8401
9.18448 8435
9.1928 8469
9.20112 8502
9.20944 8535
9.21776 8568
9.22608 8601
9.2344 8634
9.24272 8667
9.25104 8700
9.25936 8733
9.26768 8765
9.276 8797
9.28432 8829
9.29264 8861
9.30096 8893
9.30928 8925
9.3176 8957
wrong@hridoy-ict:~/ns-allinone-3.30/ns-3.30$

**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.