**Amrita kamkar**

**ID:IT-14060**

**Experiment No:02**

**Experiment Name: TCP variants**

**Objectives:** In this experiment, the TCP variants have been applied based on the ns-3 installation.

**Theoretical Explanation:**

TCP is a transport layer protocol, part of the TCP/IP suite which defines how to establish and  
maintain a connection in a network. It is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which supports a variety network applications.  
TCP’s design philosophy has evolved considerably from where the goal was to develop an effective packet switching protocol to a protocol which is fair, robust and reliable. As the internet traffic has increased substantially over the past few decades it was important for TCP to be fair and take into consideration the congestion in a network.

The TCP variants were supported by the NS-3.25 and 3.26 versions. Now these are user-friendly with the newer versions(example ns-3.29,ns-3.30,ns-3.31 etc.)

To host TCP Socket attributes common to all implementations as follow :­

– Send Buffer  
– Receive Buffer  
– Segment Size  
– Slow Start Threshold  
– Initial Congestion Window  
– and few more

ns3.25 version support following TCP Variants :­  
– Tahoe  
– Reno  
– New Reno (default)  
– Westwood  
– Westwood+  
– Hybla  
– High Speed

**TCP Reno**

It was introduced in 1990 by Van Jacobson. It has the same features like TCP Tahoe.  
We can also represent it as follows:-  
TCP Tahoe + Fast Recovery = TCP Reno

In TCP Reno when three duplicate packets are received, then it is the sign of  
congestion. If congestion occurs, then TCP Reno retransmits the packets and enters a  
new mechanism that is fast recovery. The following shows the algorithm for TCP  
Reno:-

if (cwnd<ssthresh)  
cwnd = cwnd + 1 # slow start  
else if (cwnd>= ssthresh)  
cwnd = cwnd + 1/cwnd# congestion avoidance  
if (duplicate ACK)  
If (duplicate ACK = = (1 || 2))  
cwnd = ssthresh #packet delayed/ out-of-packet received  
1718 *Harjinder Kaur and Dr. Gurpreet Singh*ssthresh = cwnd/2  
else (duplicate ACK > 2)  
cwnd = cwnd + Number (ACK) # packet loss due to congestion  
ssthresh = cwnd/2

The algorithm shows if cwnd (congestion window) is less than the threshold value  
(that is represented using variable ssthresh) then congestion window is increment by  
one otherwise it enters the slow start. As in algorithm shows if one or two  
acknowledgments are received, then threshold value is set half of the congestion  
window, but if more than two acknowledgments are received then it indicates the  
congestion. For each duplicate acknowledgment received increase congestion window  
by 1. TCP Reno has a limitation that, it can detect only single packet loss .

**TCP New Reno**

TCP New Reno is the extension of TCP Reno. It has some advantages over TCP Reno that can detect the multiple packet loss and it does not leave the fast recovery until it receives acknowledgment of all packets, present in the window . The fast recovery phase proceeds as in TCP Reno, when a fresh acknowledgment is received then there are two cases:- (i) If it acknowledges all the packets which are outstanding when entered fast recovery, then it exits fast recovery and set cwnd to ssthresh and still continues congestion avoidance. (ii) If the acknowledgment is an incomplete acknowledgement, then it deduces that the next packet in line was lost and it retransmits that packet and sets the number of duplicate acknowledgment received on 0 . We have some advantages of TCP New Reno these advantages are given below to measure the retransmit the packet :

It can detect multiple packet loss. Its congestion avoidance mechanism is very efficient and utilizes network resources much more efficiently. TCP New Reno has few retransmits because of its modified congestion avoidance and slow start.

**TCP SACK**

TCP SACK or selective acknowledgement requires that packets should acknowledge  
selectively. It is an option enabling a receiver to tell the sender the range of noncontiguous packets received. Without SACK, the receiver

In TCP Reno when three duplicate packets are received, then it is the sign of congestion. If congestion occurs, then TCP Reno retransmits the packets and enters a new mechanism that is fast recovery. The following shows the algorithm for TCP Reno:-

if (cwnd<ssthresh)

cwnd = cwnd + 1 # slow start

else if (cwnd>= ssthresh) cwnd = cwnd + 1/cwnd# congestion avoidance if (duplicate ACK)

If (duplicate ACK = = (1 || 2))

cwnd = ssthresh #packet delayed/ out-of-packet received

ssthresh = cwnd/2 else (duplicate ACK> 2)

cwnd = cwnd + Number (ACK) # packet loss due to congestion ssthresh = cwnd/2

The algorithm shows if cwnd (congestion window) is less than the threshold value (that is represented using variable ssthresh) then congestion window is increment by one otherwise it enters the slow start. As in algorithm shows if one or two acknowledgments are received, then threshold value is set half of the congestion window, but if more than two acknowledgments are received then it indicates the congestion. For each duplicate acknowledgment received increase congestion window by 1. TCP Reno has a limitation that, it can detect only single packet loss .

**Pseudo Code for the TCP variants:**

#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");

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;

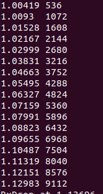
EventId m\_sendEvent;

bool m\_running;

uint32\_t m\_packetsSent;

};

The generated output:



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)

{

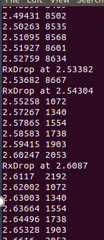
ScheduleTx ();

}

}

void

Output measured :



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);

}

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));

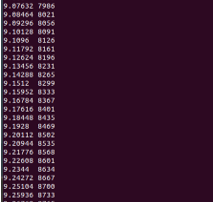
Simulator::Run ();

Simulator::Destroy ();

return 0;

}

The obtained output:



**Discussion:** In this experiment the TCP(Transfer control protocol) variants were evaluated. The ns-3.30(upgraded version) were used. The code was written in C++ language. First Installing a TCP socket on sNode1 that connects to next Node. Then Installed a UDP socket instance on the former which will connect to another Node. When this connection is established the TCP application at time 1sec was started. Again the UDP application at time 20s at rate Rate1 such that it clogs half of the bridge's link capacity. Increase the UDP application's rate at time 30s to rate Rate2 such that it clogs the whole of the dumbbell bridge's capacity. Although TCP has six important variations the experiment were done for Tahoe, old Reno and new Reno.