Group 6
assignment 4 : ns3 simulation

120123027 bhagyashree
120123018 harsh deep
120123028 pranavendra

SINGLE FLOW ANALYSIS :

-dumbbell topology used Sender : H1,H2,H3,R1
Receiver : H4,H5,H6 R2

- H1 is attached with TCP Reno agent.
- H2 is attached with TCP Vegas agent.
- H3 is attached with TCP Fack agent.
- packet size: 1.2KB.

- Number of packets/QUEUE Size decided by Bandwidth delay product:
i.e. #packets /queue size = Bandwidth*Delay(in bits)/packetsize(in bits)

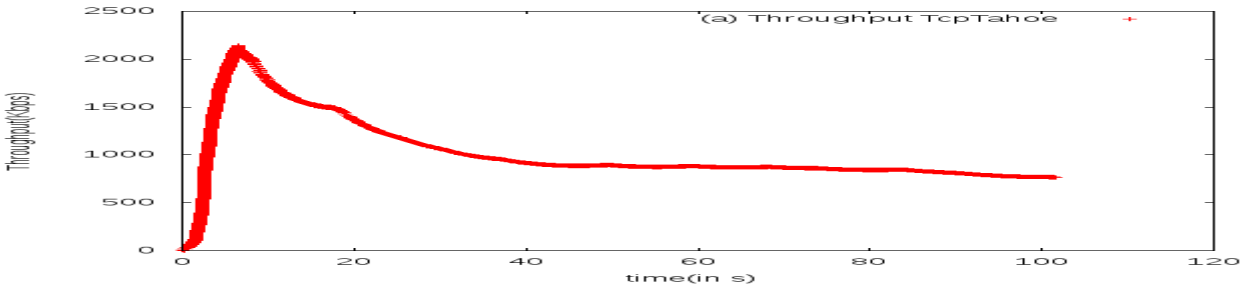
- Congestio n window, throughput, goodput and congestion loss VS Time traced using NS3
- Congestion window, throughput and goodput are measured using TraceCallbacks -
-T=100S SINGKE FLOW
-T=100 S OTHERS START AT T=20S MULTIFLOW

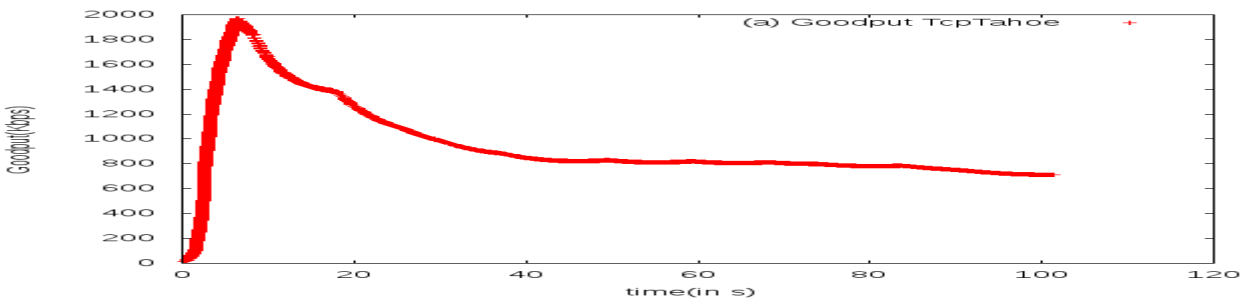
observation : throughput multiflow < single flow

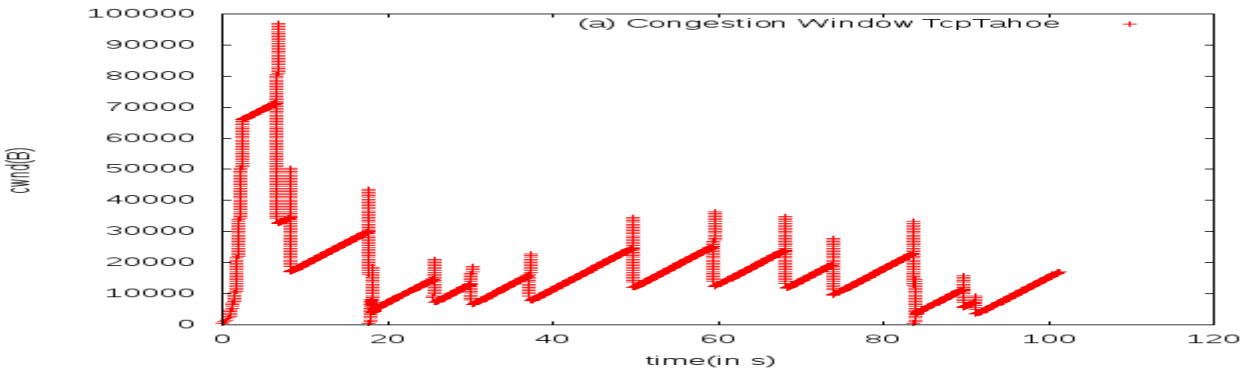
TCP TAHOE

SINGLE FLOW TcpTahoe attached to H2-H5

Total Packet Lost: 13(all congestion)
Max throughput: 2170 Kbps

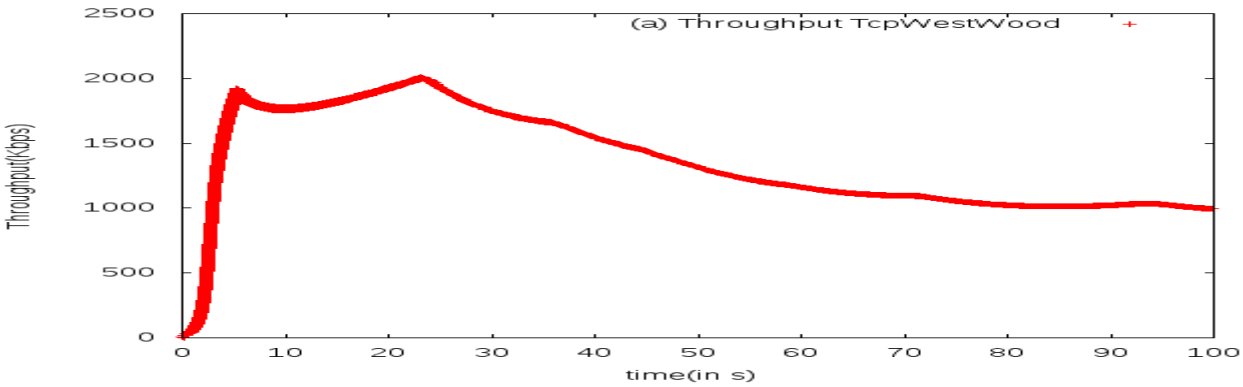


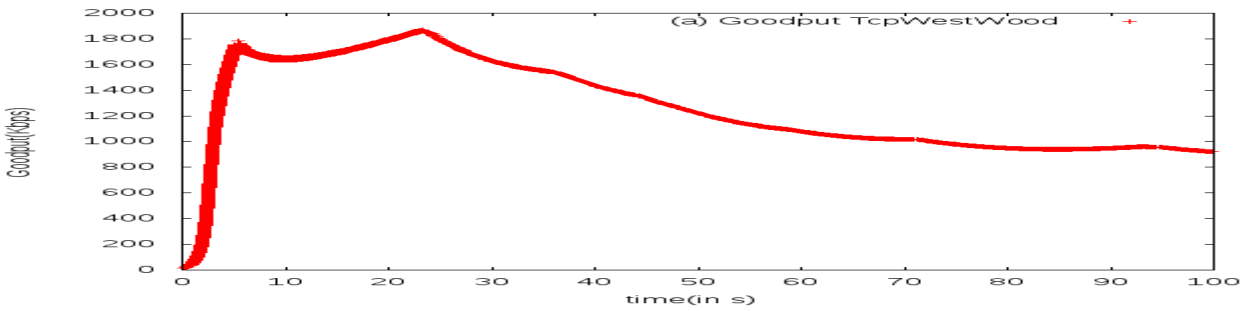


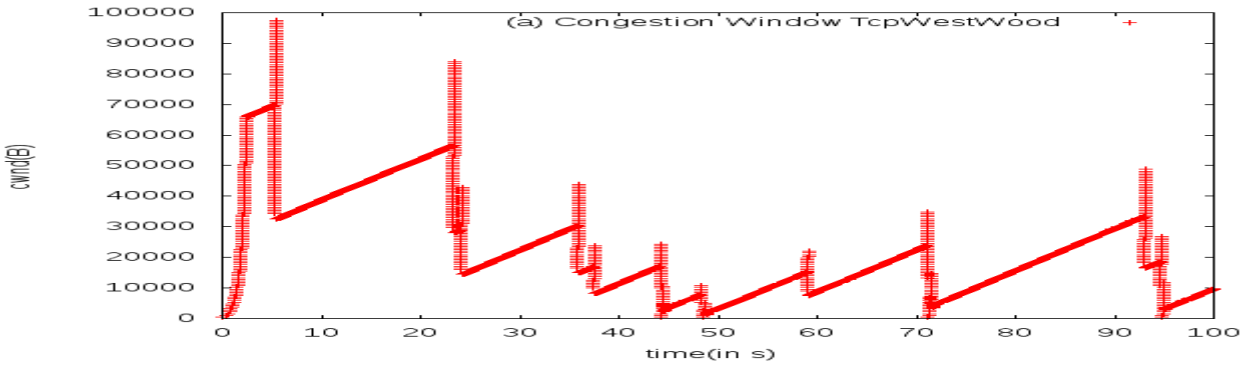


TcpWestWood

SINGLE FLOW TcpWestWood H3-H6
Packet Loss : 9 (Congestion Loss all)
max throughput :2070 kbps



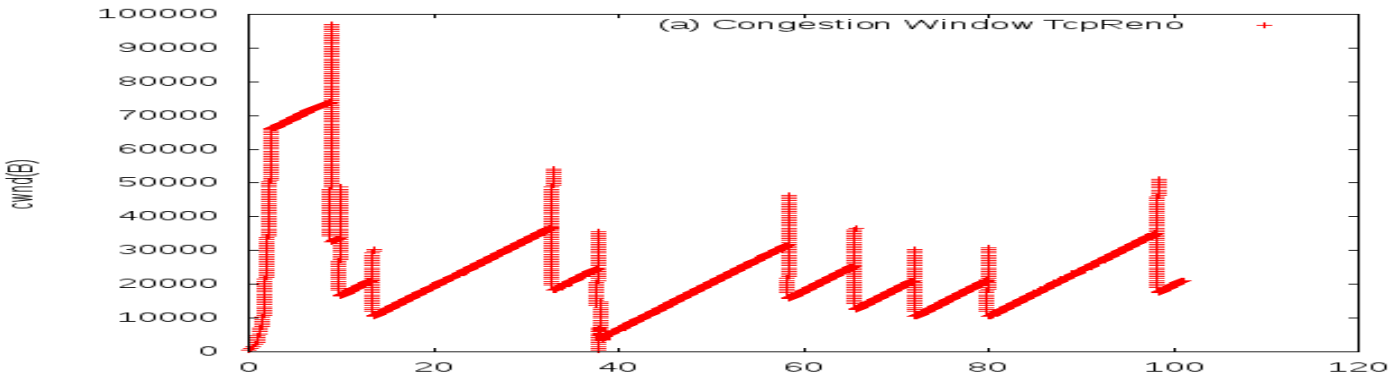


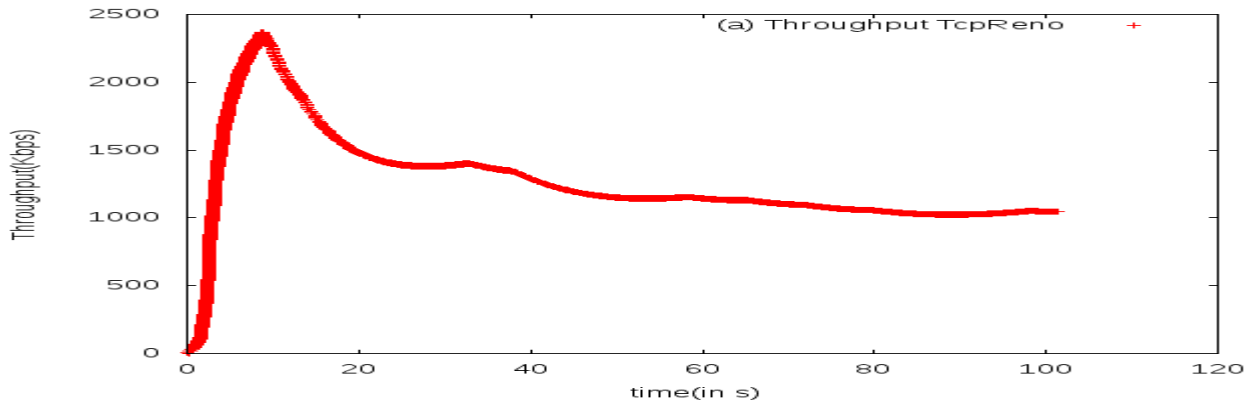


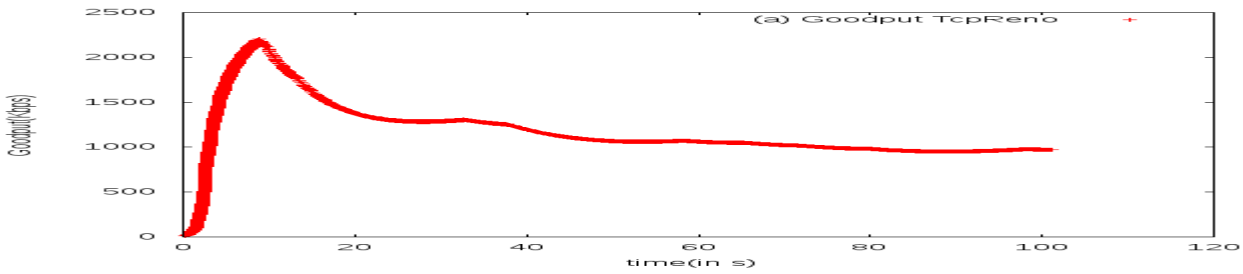
TCP RENO

SINGLE FLOW TcpReno attached to H1-H4

Packet Lost: 10(CONGESTION loss all)
Max throughput: 2414 Kbps





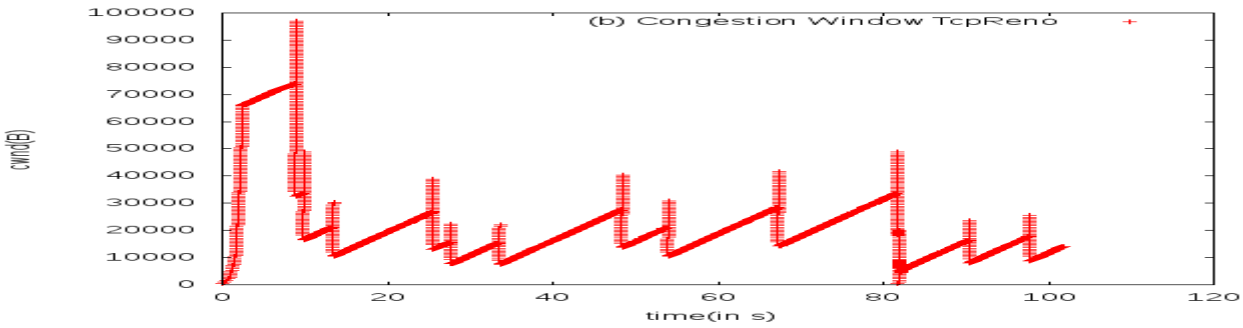


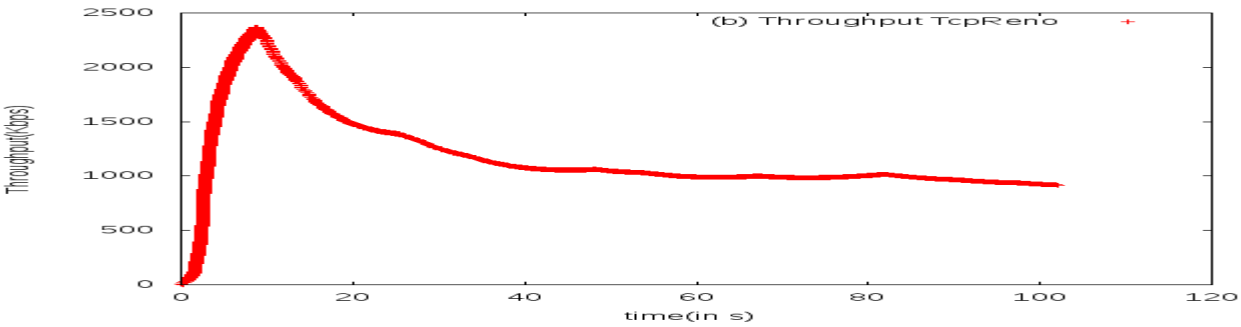
MULTIFLOW ANALYSIS

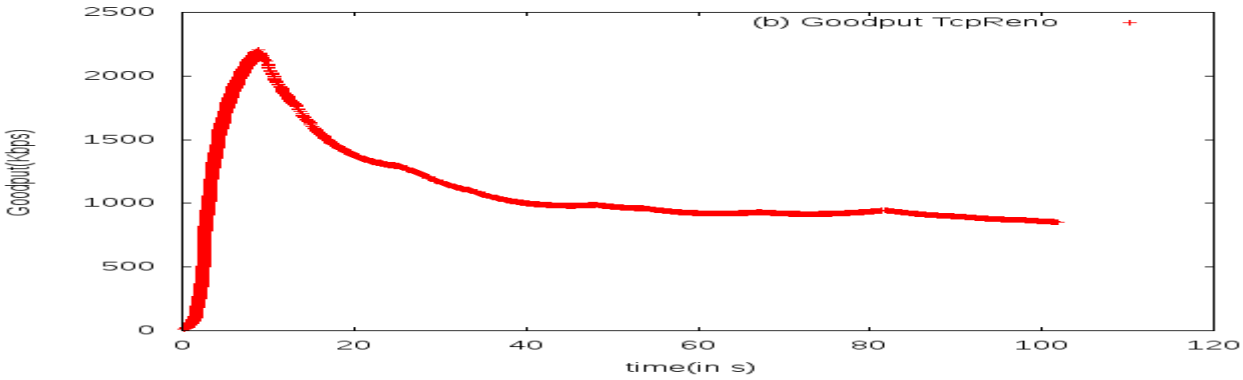
TCP RENO

SINGLE FLOW TcpReno attached to H1-H4

Packet Lost: 10(CONGESTION loss all)
Max throughput: 2416 Kbps



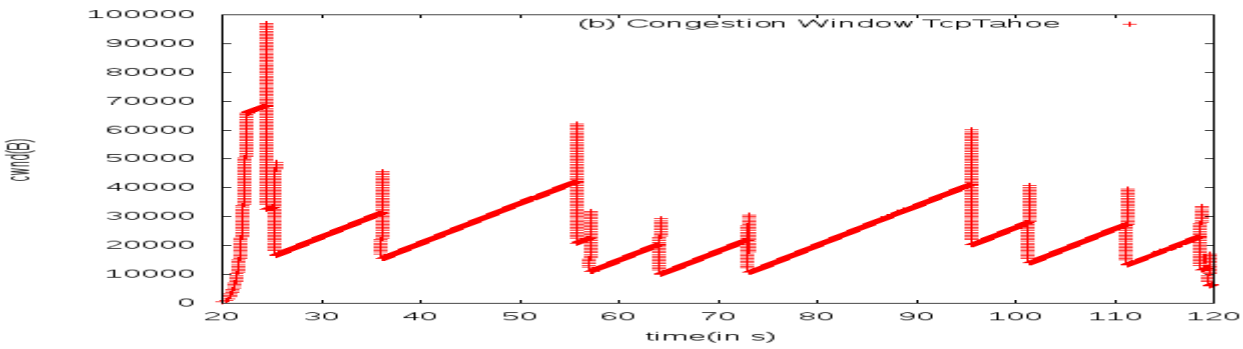


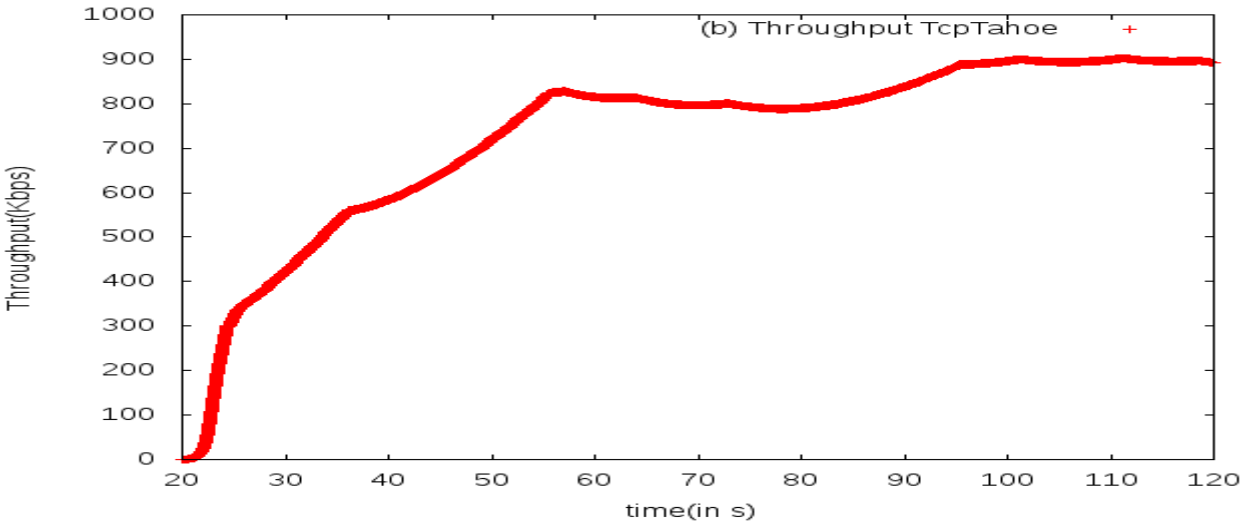


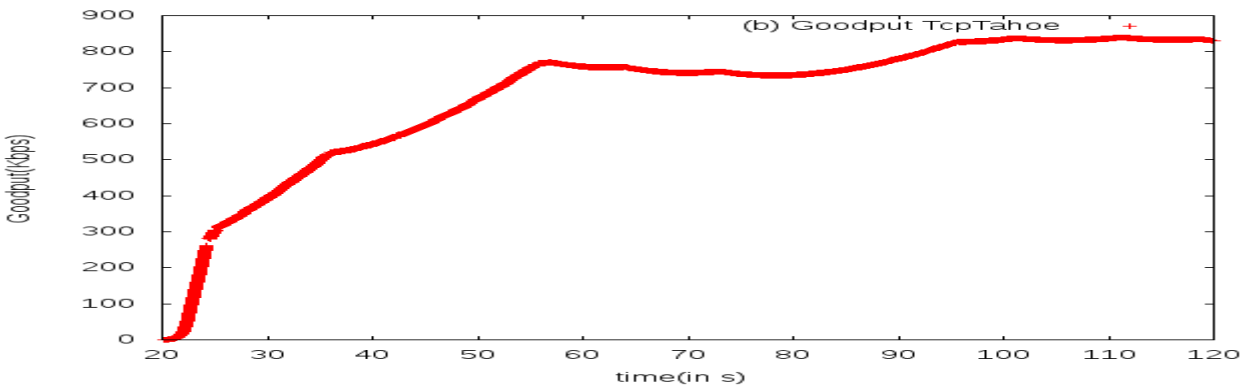
TCP TAHOE

MULTIFLOW FLOW TcpTahoe attached to H2-H5

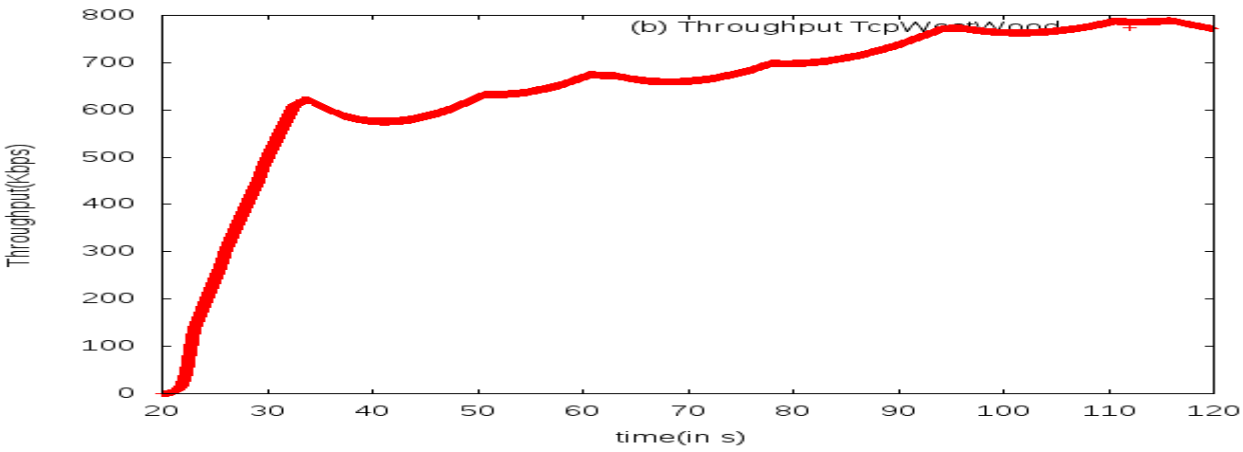
Total Packet Lost: 9(all congestion)
Max throughput: 2090 Kbps

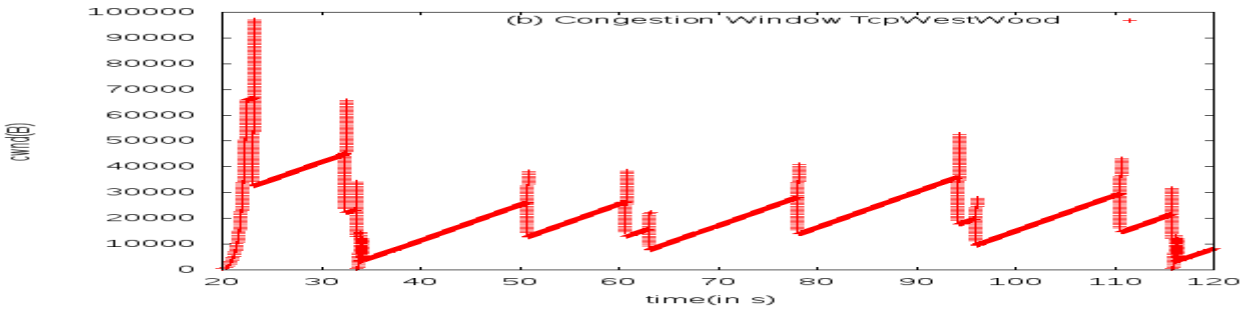


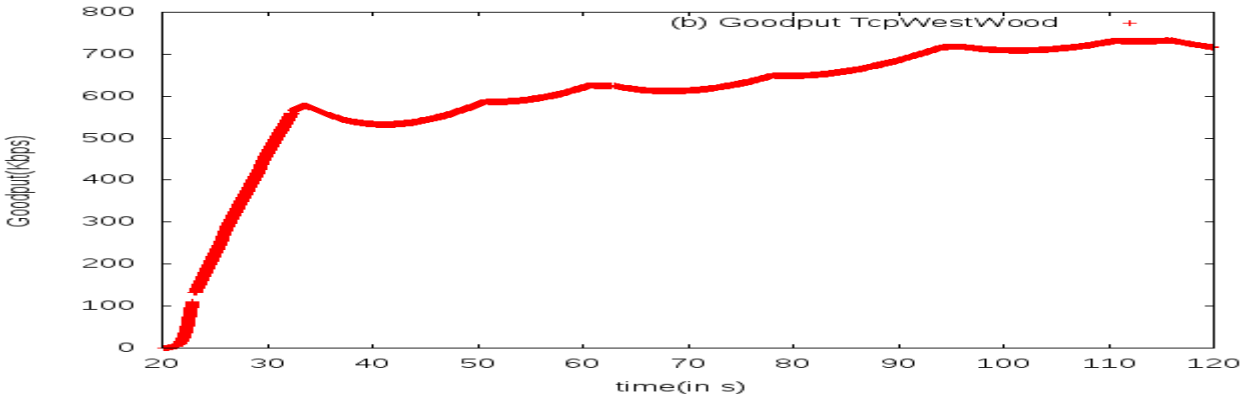




TCP WESTWOOD
MULTIFLOW H3-H6
PACKET LOSS 10(CONGESTION LOSS)
THROUGHPUT 810 Kbps







code :

```
/*
      LAB Assignment #4

    Question Specifications

 Network topology

      H1 (n0)  --->   +           + ----->  H4 (n3)
                      |           |
      H2 (n1) -->  (n6) R1  -> (n7) R2 -> H5 (n4)
                      |           |
      H3 (n2) -->  +           +-->       H6 (n5)
```

```
 - Links b/2 R1 and R2 : 10Mbps and 50ms
 - Links b/2 Hi and Rj : 100Mbps and 20ms (i = {1,2,3,4,5,6} and j = {1,2}

 - packet size : 1.2KB
 - Number of packets to be decided by bandwidth delay product
 - implies No. of packets b/w hosts and routers := 100Mbps * 20ms = 2000000
 - implies No. of packets b/w routers := 10Mbps * 50ms = 500000


*/

#include <fstream>
#include <string>
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/internet-module.h"
#include "ns3/flow-monitor-module.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/netanim-module.h"
#include <iostream>
#include <map>

using namespace ns3;
using namespace std;

NS_LOG_COMPONENT_DEFINE ("Lab4");

class MyApp : public Application
{
    public:

    MyApp ();
    virtual ~MyApp();

    void Setup (Ptr<Socket> socket, Address address, uint32_t packetSize,
uint32_t nPackets, DataRate dataRate);
    void ChangeRate(DataRate newrate);

    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);
    }
}

void MyApp::ChangeRate(DataRate newrate)
{
    m_dataRate = newrate;
    return;
}

//we o/p to a stream i.e. a file as o/p is very long so can't go to stdout.
static void CwndChange(Ptr<OutputStreamWrapper> stream, double startTime,
uint32_t oldCwnd, uint32_t newCwnd)
{
      *stream->GetStream() << Simulator::Now ().GetSeconds () - startTime <<
"\t" << newCwnd << std::endl;
}
```

```
void IncRate (Ptr<MyApp> app, DataRate rate)
{
    app->ChangeRate(rate);
    return;
}

std::map<uint32_t, uint32_t> mapDrop;
static void packetDrop(Ptr<OutputStreamWrapper> stream, double startTime,
uint32_t myId)
{
      *stream->GetStream() << Simulator::Now ().GetSeconds () - startTime <<
"\t" << std::endl;

      if(mapDrop.find(myId) == mapDrop.end())
            mapDrop[myId] = 0;

      mapDrop[myId]++;
}


map<Address, double> mapBytesReceived;
map<string, double> mapBytesReceivedIPV4, mapMaxThroughput;
static double lastTimePrint = 0, lastTimePrintIPV4 = 0;
double printGap = 0;

void ReceivedPacket(Ptr<OutputStreamWrapper> stream, double startTime, string
context, Ptr<const Packet> p, const Address& addr)
{
    double timeNow = Simulator::Now().GetSeconds(); //current time
      if(mapBytesReceived.find(addr) == mapBytesReceived.end())   //if address
at end implies 0
            mapBytesReceived[addr] = 0;

      mapBytesReceived[addr] += p->GetSize();

      double kbps = (((mapBytesReceived[addr] * 8.0) / 1024)/(timeNow-
startTime));
      if(timeNow - lastTimePrint >= printGap)
      {
            lastTimePrint = timeNow;
            *stream->GetStream() << timeNow-startTime << "\t" <<  kbps << endl;
      }
}


void ReceivedPacketIPV4(Ptr<OutputStreamWrapper> stream, double startTime,
std::string context, Ptr<const Packet> p, Ptr<Ipv4> ipv4, uint32_t interface)
{
      double timeNow = Simulator::Now().GetSeconds();

      if(mapBytesReceivedIPV4.find(context) == mapBytesReceivedIPV4.end())
            mapBytesReceivedIPV4[context] = 0;

      if(mapMaxThroughput.find(context) == mapMaxThroughput.end())
            mapMaxThroughput[context] = 0;

      mapBytesReceivedIPV4[context] += p->GetSize();
      double kbps_ = (((mapBytesReceivedIPV4[context] * 8.0) / 1024)/(timeNow-
startTime));

      if(timeNow - lastTimePrintIPV4 >= printGap)
      {
```

```
            lastTimePrintIPV4 = timeNow;
            *stream->GetStream() << timeNow-startTime << "\t" <<  kbps_ <<
std::endl;
            if(mapMaxThroughput[context] < kbps_)
                  mapMaxThroughput[context] = kbps_;
      }
}

Ptr<Socket> uniFlow(Address sinkAddress,
                              uint32_t sinkPort,
                              string tcpVariant,
                              Ptr<Node> hostNode,
                              Ptr<Node> sinkNode,
                              double startTime,
                              double stopTime,
                              uint32_t packetSize,
                              uint32_t numPackets,
                              std::string dataRate,
                              double appStartTime,
                              double appStopTime)
{
    //if tcp socket is Reno set it to default and so on for Tahoe and Westwood
      if(tcpVariant.compare("TcpReno") == 0)
            Config::SetDefault("ns3::TcpL4Protocol::SocketType",
TypeIdValue(TcpReno::GetTypeId()));

      else if(tcpVariant.compare("TcpTahoe") == 0)
            Config::SetDefault("ns3::TcpL4Protocol::SocketType",
TypeIdValue(TcpTahoe::GetTypeId()));

      else if(tcpVariant.compare("TcpWestwood") == 0)
            Config::SetDefault("ns3::TcpL4Protocol::SocketType",
TypeIdValue(TcpWestwood::GetTypeId()));

      else
      {
            fprintf(stderr, "Invalid TCP version\n");
            exit(EXIT_FAILURE);
      }

      //setup the sink
      PacketSinkHelper packetSinkHelper("ns3::TcpSocketFactory",
InetSocketAddress(Ipv4Address::GetAny(), sinkPort));
      ApplicationContainer sinkApps = packetSinkHelper.Install(sinkNode);
      sinkApps.Start(Seconds(startTime));
      sinkApps.Stop(Seconds(stopTime));

      Ptr<Socket> ns3TcpSocket = Socket::CreateSocket(hostNode,
TcpSocketFactory::GetTypeId());

    //setup the source
      Ptr<MyApp> app = CreateObject<MyApp>();
      app->Setup(ns3TcpSocket, sinkAddress, packetSize, numPackets,
DataRate(dataRate));
      hostNode->AddApplication(app);
      app->SetStartTime(Seconds(appStartTime));
      app->SetStopTime(Seconds(appStopTime));

      return ns3TcpSocket;
}

/*void animation(string s, NodeContainer routers, NodeContainer senders,
NodeContainer recievers)
{
```

```
    //NetAnim
    AnimationInterface anim (s);

    //left
    anim.SetConstantPosition (senders.Get(0), 1.0, 2.0);
    anim.SetConstantPosition (senders.Get(1), 1.0, 5.0);
    anim.SetConstantPosition (senders.Get(2), 1.0, 8.0);

    //right
    anim.SetConstantPosition (recievers.Get(0), 7.0, 2.0);
    anim.SetConstantPosition (recievers.Get(1), 7.0, 5.0);
    anim.SetConstantPosition (recievers.Get(2), 7.0, 8.0);

    //routers
    anim.SetConstantPosition (routers.Get(0), 3.0, 5.0);
    anim.SetConstantPosition (routers.Get(1), 5.0, 5.0);
}*/


void part1(void)
{

    cout << "Setting up Part 1" << endl;

      string rateHR = "100Mbps";  //data rate b/w hosts and routers
      string latencyHR = "20ms";  //latency b/w hosts and routers
      string rateRR = "10Mbps";  //data rate b/w routers
      string latencyRR = "50ms"; //latenc b/w routers

      uint32_t packetSize = 1.2*1024;          //1.2KB
      uint32_t queueSizeHR = (100000*20)/packetSize;  //100Mbps
      uint32_t queueSizeRR = (10000*50)/packetSize;   //10Mbps

      uint32_t numSender = 3;

      double errorP = 0.000001;

      Config::SetDefault("ns3::DropTailQueue::Mode",
StringValue("QUEUE_MODE_PACKETS"));

      //Creating channel without IP address
      PointToPointHelper p2pHR, p2pRR;

      p2pHR.SetDeviceAttribute("DataRate", StringValue(rateHR));
      p2pHR.SetChannelAttribute("Delay", StringValue(latencyHR));
      p2pHR.SetQueue("ns3::DropTailQueue", "MaxPackets",
UintegerValue(queueSizeHR));

      p2pRR.SetDeviceAttribute("DataRate", StringValue(rateRR));
      p2pRR.SetChannelAttribute("Delay", StringValue(latencyRR));
      p2pRR.SetQueue("ns3::DropTailQueue", "MaxPackets",
UintegerValue(queueSizeRR));

      //Adding some errorrate
      Ptr<RateErrorModel> em = CreateObjectWithAttributes<RateErrorModel>
("ErrorRate", DoubleValue (errorP));

    //create nodes(routers, senders and recievers)
      NodeContainer routers, senders, recievers;
      routers.Create(2);
      senders.Create(numSender);
      recievers.Create(numSender);

      NetDeviceContainer routerDevices = p2pRR.Install(routers);
```

```
      NetDeviceContainer leftRouterDevices, rightRouterDevices, senderDevices,
recieverDevices;

      //Adding links
      for(uint32_t i = 0; i < numSender; ++i)
      {
            NetDeviceContainer cleft = p2pHR.Install(routers.Get(0),
senders.Get(i));
            leftRouterDevices.Add(cleft.Get(0));
            senderDevices.Add(cleft.Get(1));
            cleft.Get(0)->SetAttribute("ReceiveErrorModel", PointerValue(em));

            NetDeviceContainer cright = p2pHR.Install(routers.Get(1),
recievers.Get(i));
            rightRouterDevices.Add(cright.Get(0));
            recieverDevices.Add(cright.Get(1));
            cright.Get(0)->SetAttribute("ReceiveErrorModel", PointerValue(em));
      }

      //Install Internet Stack
      InternetStackHelper stack;
      stack.Install(routers);
      stack.Install(senders);
      stack.Install(recievers);


      //Adding IP addresses
      Ipv4AddressHelper routerIP = Ipv4AddressHelper("10.3.0.0",
"255.255.255.0");
      Ipv4AddressHelper senderIP = Ipv4AddressHelper("10.1.0.0",
"255.255.255.0");
      Ipv4AddressHelper recieverIP = Ipv4AddressHelper("10.2.0.0",
"255.255.255.0");


      Ipv4InterfaceContainer routerIFC, senderIFCs, recieverIFCs,
leftRouterIFCs, rightRouterIFCs;

      routerIFC = routerIP.Assign(routerDevices);

      for(uint32_t i = 0; i < numSender; ++i)
      {
            NetDeviceContainer senderDevice;
            senderDevice.Add(senderDevices.Get(i));
            senderDevice.Add(leftRouterDevices.Get(i));
            Ipv4InterfaceContainer senderIFC = senderIP.Assign(senderDevice);
            senderIFCs.Add(senderIFC.Get(0));
            leftRouterIFCs.Add(senderIFC.Get(1));
            senderIP.NewNetwork();

            NetDeviceContainer recieverDevice;
            recieverDevice.Add(recieverDevices.Get(i));
            recieverDevice.Add(rightRouterDevices.Get(i));
            Ipv4InterfaceContainer recieverIFC =
recieverIP.Assign(recieverDevice);
            recieverIFCs.Add(recieverIFC.Get(0));
            rightRouterIFCs.Add(recieverIFC.Get(1));
            recieverIP.NewNetwork();
      }


      /*****************************************************************
      PART (1) :
```

```
      one flow for each tcp variant

      -> throughput
      -> evolution of congestion window*/

      cout << "Part A starting (integrated with Part 3 ; i.e. goodput
inclusive)" << endl;

      double durationGap = 100;
      double netDuration = 0;
      uint32_t port = 9000;
      uint32_t numPackets = 10000000;
      string transferSpeed = "400Mbps";

      //TCP Reno from H1 to H4
      cout << "Flow from H1 -> H4 : TcpReno" << endl;
      cout << "Writing to app6_h1_h4_a.cwnd (congestion window) and
app6_h1_h4_a.tp (throughput)" << endl;
      AsciiTraceHelper asciiTraceHelper;
      Ptr<OutputStreamWrapper> stream1CWND =
asciiTraceHelper.CreateFileStream("app6_h1_h4_a.cwnd");
      Ptr<OutputStreamWrapper> stream1TP =
asciiTraceHelper.CreateFileStream("app6_h1_h4_a.tp");
      Ptr<OutputStreamWrapper> stream1PD =
asciiTraceHelper.CreateFileStream("app6_h1_h4_a.congestion_loss");
      Ptr<OutputStreamWrapper> stream1GP =
asciiTraceHelper.CreateFileStream("application_6_h1_h4_a.gp");
      Ptr<Socket> ns3TcpSocket1 =
uniFlow(InetSocketAddress(recieverIFCs.GetAddress(0), port), port, "TcpReno",
senders.Get(0), recievers.Get(0), netDuration, netDuration+durationGap,
packetSize, numPackets, transferSpeed, netDuration, netDuration+durationGap);
      ns3TcpSocket1->TraceConnectWithoutContext("CongestionWindow",
MakeBoundCallback (&CwndChange, stream1CWND, netDuration));
      ns3TcpSocket1->TraceConnectWithoutContext("Drop", MakeBoundCallback
(&packetDrop, stream1PD, netDuration, 1));

      // Measure PacketSinks
      std::string sink = "/NodeList/5/ApplicationList/0/$ns3::PacketSink/Rx";
      Config::Connect(sink, MakeBoundCallback(&ReceivedPacket, stream1GP,
netDuration));

      std::string sink_ = "/NodeList/5/ApplicationList/0/$ns3::PacketSink/Rx";
      Config::Connect(sink_, MakeBoundCallback(&ReceivedPacket, stream1TP,
netDuration));

      netDuration += durationGap;


      //TCP Tahoe from H2 to H5
      cout << "Flow from H2 -> H5 : TcpTahoe" << endl;
      cout << "Writing to app6_h2_h5_a.cwnd (congestion window) and
app6_h2_h5_a.tp (throughput)" << endl;
      Ptr<OutputStreamWrapper> stream2CWND =
asciiTraceHelper.CreateFileStream("app6_h2_h5_a.cwnd");
      Ptr<OutputStreamWrapper> stream2PD =
asciiTraceHelper.CreateFileStream("app6_h2_h5_a.congestion_loss");
      Ptr<OutputStreamWrapper> stream2TP =
asciiTraceHelper.CreateFileStream("app6_h2_h5_a.tp");
      Ptr<OutputStreamWrapper> stream2GP =
asciiTraceHelper.CreateFileStream("app6_h2_h5_a.gp");
      Ptr<Socket> ns3TcpSocket2 =
uniFlow(InetSocketAddress(recieverIFCs.GetAddress(1), port), port, "TcpTahoe",
senders.Get(1), recievers.Get(1), netDuration, netDuration+durationGap,
packetSize, numPackets, transferSpeed, netDuration, netDuration+durationGap);
```

```
      ns3TcpSocket2->TraceConnectWithoutContext("CongestionWindow",
MakeBoundCallback (&CwndChange, stream2CWND, netDuration));
      ns3TcpSocket2->TraceConnectWithoutContext("Drop", MakeBoundCallback
(&packetDrop, stream2PD, netDuration, 2));

      sink_ = "/NodeList/6/ApplicationList/0/$ns3::PacketSink/Rx";
      Config::Connect(sink_, MakeBoundCallback(&ReceivedPacket, stream2TP,
netDuration));
      sink = "/NodeList/6/ApplicationList/0/$ns3::PacketSink/Rx";
      Config::Connect(sink, MakeBoundCallback(&ReceivedPacket, stream2GP,
netDuration));
      netDuration += durationGap;

      //TCP WestWood from H3 to H6
      cout << "Flow from H3 -> H6 : TcpWestwood" << endl;
      cout << "Writing to app6_h3_h6_a.cwnd (congestion window) and
app6_h3_h6_a.tp (throughput)" << endl;
      Ptr<OutputStreamWrapper> stream3CWND =
asciiTraceHelper.CreateFileStream("app6_h3_h6_a.cwnd");
      Ptr<OutputStreamWrapper> stream3PD =
asciiTraceHelper.CreateFileStream("app6_h3_h6_a.congestion_loss");
      Ptr<OutputStreamWrapper> stream3TP =
asciiTraceHelper.CreateFileStream("app6_h3_h6_a.tp");
      Ptr<OutputStreamWrapper> stream3GP =
asciiTraceHelper.CreateFileStream("app6_h3_h6_a.gp");
      Ptr<Socket> ns3TcpSocket3 =
uniFlow(InetSocketAddress(recieverIFCs.GetAddress(2), port), port,
"TcpWestwood", senders.Get(2), recievers.Get(2), netDuration,
netDuration+durationGap, packetSize, numPackets, transferSpeed, netDuration,
netDuration+durationGap);
      ns3TcpSocket3->TraceConnectWithoutContext("CongestionWindow",
MakeBoundCallback (&CwndChange, stream3CWND, netDuration));

      sink_ = "/NodeList/7/ApplicationList/0/$ns3::PacketSink/Rx";
      Config::Connect(sink_, MakeBoundCallback(&ReceivedPacket, stream3TP,
netDuration));
      sink = "/NodeList/7/ApplicationList/0/$ns3::PacketSink/Rx";
      Config::Connect(sink, MakeBoundCallback(&ReceivedPacket, stream3GP,
netDuration));
      netDuration += durationGap;

      //Turning on Static Global Routing
      Ipv4GlobalRoutingHelper::PopulateRoutingTables();

      Ptr<FlowMonitor> flowmon;
      FlowMonitorHelper flowmonHelper;
      flowmon = flowmonHelper.InstallAll();
      Simulator::Stop(Seconds(netDuration));

    //animation("anim3.xml", routers, senders, recievers);    //run the
animation for netanim

      Simulator::Run();

      flowmon->CheckForLostPackets();

    //throughput calculation
    cout << "Part b throughput into app6_a.tp" << endl;
      Ptr<OutputStreamWrapper> streamTP =
asciiTraceHelper.CreateFileStream("app6_a.tp");
      Ptr<Ipv4FlowClassifier> classifier =
DynamicCast<Ipv4FlowClassifier>(flowmonHelper.GetClassifier());
      std::map<FlowId, FlowMonitor::FlowStats> stats = flowmon->GetFlowStats();
      for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i =
```

```
stats.begin(); i != stats.end(); ++i)
      {
            Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);

            if(t.sourceAddress == "10.1.0.1")
            {
                  if(mapDrop.find(1)==mapDrop.end())
                        mapDrop[1] = 0;

                  *stream1PD->GetStream() << "TcpReno Flow " << i->first  << "
(" << t.sourceAddress << " -> " << t.destinationAddress << ")\n";
                  *stream1PD->GetStream()  << "Net Packet Lost: " << i-
>second.lostPackets << "\n";
                  *stream1PD->GetStream()  << "Packet Lost due to buffer
overflow: " << mapDrop[1] << "\n";
                  *stream1PD->GetStream()  << "Packet Lost due to Congestion: "
<< i->second.lostPackets - mapDrop[1] << "\n";
                  *stream1PD->GetStream() << "Max throughput: " <<
mapMaxThroughput["/NodeList/5/$ns3::Ipv4L3Protocol/Rx"] << std::endl;
            }
            else if(t.sourceAddress == "10.1.1.1")
            {
                  if(mapDrop.find(2)==mapDrop.end())
                        mapDrop[2] = 0;

                  *stream2PD->GetStream() << "Tcp Tahoe Flow " << i->first  << "
(" << t.sourceAddress << " -> " << t.destinationAddress << ")\n";
                  *stream2PD->GetStream()  << "Net Packet Lost: " << i-
>second.lostPackets << "\n";
                  *stream2PD->GetStream()  << "Packet Lost due to buffer
overflow: " << mapDrop[2] << "\n";
                  *stream2PD->GetStream()  << "Packet Lost due to Congestion: "
<< i->second.lostPackets - mapDrop[2] << "\n";
                  *stream2PD->GetStream() << "Max throughput: " <<
mapMaxThroughput["/NodeList/6/$ns3::Ipv4L3Protocol/Rx"] << std::endl;
            }
            else if(t.sourceAddress == "10.1.2.1")
            {
                  if(mapDrop.find(3)==mapDrop.end())
                        mapDrop[3] = 0;

                  *stream3PD->GetStream() << "Tcp WestWood Flow " << i->first
<< " (" << t.sourceAddress << " -> " << t.destinationAddress << ")\n";
                  *stream3PD->GetStream()  << "Net Packet Lost: " << i-
>second.lostPackets << "\n";
                  *stream3PD->GetStream()  << "Packet Lost due to buffer
overflow: " << mapDrop[3] << "\n";
                  *stream3PD->GetStream()  << "Packet Lost due to Congestion: "
<< i->second.lostPackets - mapDrop[3] << "\n";
                  *stream3PD->GetStream() << "Max throughput: " <<
mapMaxThroughput["/NodeList/7/$ns3::Ipv4L3Protocol/Rx"] << std::endl;
            }
      }

      Simulator::Destroy();
}


void part2()
{
    cout << "Seting up Part B with C" << endl;

      string rateHR = "100Mbps";  //data rate b/w hosts and routers
      string latencyHR = "20ms";  //latency b/w hosts and routers
```

```
      string rateRR = "10Mbps";  //data rate b/w routers
      string latencyRR = "50ms"; //latenc b/w routers

      uint32_t packetSize = 1.2*1024;          //1.2KB
      uint32_t queueSizeHR = (100000*20)/packetSize;  //100Mbps
      uint32_t queueSizeRR = (10000*50)/packetSize;   //10Mbps

      uint32_t numSender = 3;

      double errorP = 0.000001;

      Config::SetDefault("ns3::DropTailQueue::Mode",
StringValue("QUEUE_MODE_PACKETS"));

      //Creating channel without IP address
      PointToPointHelper p2pHR, p2pRR;

      p2pHR.SetDeviceAttribute("DataRate", StringValue(rateHR));
      p2pHR.SetChannelAttribute("Delay", StringValue(latencyHR));
      p2pHR.SetQueue("ns3::DropTailQueue", "MaxPackets",
UintegerValue(queueSizeHR));

      p2pRR.SetDeviceAttribute("DataRate", StringValue(rateRR));
      p2pRR.SetChannelAttribute("Delay", StringValue(latencyRR));
      p2pRR.SetQueue("ns3::DropTailQueue", "MaxPackets",
UintegerValue(queueSizeRR));

      //Adding some errorrate
      Ptr<RateErrorModel> em = CreateObjectWithAttributes<RateErrorModel>
("ErrorRate", DoubleValue (errorP));

    //create nodes(routers, senders and recievers)
      NodeContainer routers, senders, recievers;
      routers.Create(2);
      senders.Create(numSender);
      recievers.Create(numSender);

      NetDeviceContainer routerDevices = p2pRR.Install(routers);
      NetDeviceContainer leftRouterDevices, rightRouterDevices, senderDevices,
recieverDevices;

      //Adding links
      for(uint32_t i = 0; i < numSender; ++i)
      {
            NetDeviceContainer cleft = p2pHR.Install(routers.Get(0),
senders.Get(i));
            leftRouterDevices.Add(cleft.Get(0));
            senderDevices.Add(cleft.Get(1));
            cleft.Get(0)->SetAttribute("ReceiveErrorModel", PointerValue(em));

            NetDeviceContainer cright = p2pHR.Install(routers.Get(1),
recievers.Get(i));
            rightRouterDevices.Add(cright.Get(0));
            recieverDevices.Add(cright.Get(1));
            cright.Get(0)->SetAttribute("ReceiveErrorModel", PointerValue(em));
      }

      //Install Internet Stack
      InternetStackHelper stack;
      stack.Install(routers);
      stack.Install(senders);
      stack.Install(recievers);
```

```
      //Adding IP addresses
      Ipv4AddressHelper routerIP = Ipv4AddressHelper("10.3.0.0",
"255.255.255.0");
      Ipv4AddressHelper senderIP = Ipv4AddressHelper("10.1.0.0",
"255.255.255.0");
      Ipv4AddressHelper recieverIP = Ipv4AddressHelper("10.2.0.0",
"255.255.255.0");


      Ipv4InterfaceContainer routerIFC, senderIFCs, recieverIFCs,
leftRouterIFCs, rightRouterIFCs;

      routerIFC = routerIP.Assign(routerDevices);

      for(uint32_t i = 0; i < numSender; ++i)
      {
            NetDeviceContainer senderDevice;
            senderDevice.Add(senderDevices.Get(i));
            senderDevice.Add(leftRouterDevices.Get(i));
            Ipv4InterfaceContainer senderIFC = senderIP.Assign(senderDevice);
            senderIFCs.Add(senderIFC.Get(0));
            leftRouterIFCs.Add(senderIFC.Get(1));
            senderIP.NewNetwork();

            NetDeviceContainer recieverDevice;
            recieverDevice.Add(recieverDevices.Get(i));
            recieverDevice.Add(rightRouterDevices.Get(i));
            Ipv4InterfaceContainer recieverIFC =
recieverIP.Assign(recieverDevice);
            recieverIFCs.Add(recieverIFC.Get(0));
            rightRouterIFCs.Add(recieverIFC.Get(1));
            recieverIP.NewNetwork();
      }


      /*********************************************************************
      PART (2)

      start 2 other flows while one is progress

      ->    measure throughput and CWND of each flow at steady state
      ->  Also find the max throuhput per flow

      *********************************************************************/

      cout << "Part B Starting" << endl;

      double durationGap = 100;
      double oneFlowStart = 0;
      double otherFlowStart = 20;
      uint32_t port = 9000;
      uint32_t numPackets = 10000000;
      string transferSpeed = "400Mbps";


      //TCP Reno from H1 to H4
      cout << "Flow from H1 -> H4 : TcpReno" << endl;
      cout << "Writing to app6_h1_h4_b.cwnd (congestion window) and
app6_h1_h4_b.tp (throughput)" << endl;
      AsciiTraceHelper asciiTraceHelper;
      Ptr<OutputStreamWrapper> stream1CWND =
asciiTraceHelper.CreateFileStream("app6_h1_h4_b.cwnd");
      Ptr<OutputStreamWrapper> stream1PD =
asciiTraceHelper.CreateFileStream("app6_h1_h4_b.congestion_loss");
```

```
      Ptr<OutputStreamWrapper> stream1TP =
asciiTraceHelper.CreateFileStream("app6_h1_h4_b.tp");
      Ptr<OutputStreamWrapper> stream1GP =
asciiTraceHelper.CreateFileStream("app6_h1_h4_b.gp");
      Ptr<Socket> ns3TcpSocket1 =
uniFlow(InetSocketAddress(recieverIFCs.GetAddress(0), port), port, "TcpReno",
senders.Get(0), recievers.Get(0), oneFlowStart, oneFlowStart+durationGap,
packetSize, numPackets, transferSpeed, oneFlowStart, oneFlowStart+durationGap);
      ns3TcpSocket1->TraceConnectWithoutContext("CongestionWindow",
MakeBoundCallback (&CwndChange, stream1CWND, 0));
      ns3TcpSocket1->TraceConnectWithoutContext("Drop", MakeBoundCallback
(&packetDrop, stream1PD, 0, 1));

      std::string sink_ = "/NodeList/5/ApplicationList/0/$ns3::PacketSink/Rx";
      Config::Connect(sink_, MakeBoundCallback(&ReceivedPacket, stream1TP,
oneFlowStart));
      std::string sink = "/NodeList/5/ApplicationList/0/$ns3::PacketSink/Rx";
      Config::Connect(sink, MakeBoundCallback(&ReceivedPacket, stream1GP, 0));

      //TCP Tahoe from H2 to H5
      cout << "Flow from H2 -> H5 : TcpTahoe" << endl;
      cout << "Writing to app6_h2_h5_b.cwnd (congestion window) and
app6_h2_h5_b.tp (throughput)" << endl;
      Ptr<OutputStreamWrapper> stream2CWND =
asciiTraceHelper.CreateFileStream("app6_h2_h5_b.cwnd");
      Ptr<OutputStreamWrapper> stream2PD =
asciiTraceHelper.CreateFileStream("app6_h2_h5_b.congestion_loss");
      Ptr<OutputStreamWrapper> stream2TP =
asciiTraceHelper.CreateFileStream("app6_h2_h5_b.tp");
      Ptr<OutputStreamWrapper> stream2GP =
asciiTraceHelper.CreateFileStream("app6_h2_h5_b.gp");
      Ptr<Socket> ns3TcpSocket2 =
uniFlow(InetSocketAddress(recieverIFCs.GetAddress(1), port), port, "TcpTahoe",
senders.Get(1), recievers.Get(1), otherFlowStart, otherFlowStart+durationGap,
packetSize, numPackets, transferSpeed, otherFlowStart,
otherFlowStart+durationGap);
      ns3TcpSocket2->TraceConnectWithoutContext("CongestionWindow",
MakeBoundCallback (&CwndChange, stream2CWND, 0));
      ns3TcpSocket2->TraceConnectWithoutContext("Drop", MakeBoundCallback
(&packetDrop, stream2PD, 0, 2));

      sink_ = "/NodeList/6/ApplicationList/0/$ns3::PacketSink/Rx";
      Config::Connect(sink_, MakeBoundCallback(&ReceivedPacket, stream2TP,
otherFlowStart));
      sink = "/NodeList/6/ApplicationList/0/$ns3::PacketSink/Rx";
      Config::Connect(sink, MakeBoundCallback(&ReceivedPacket, stream2GP, 0));

      //TCP WestWood from H3 to H6
      cout << "Flow from H3 -> H6 : TcpWestwood" << endl;
      cout << "Writing to app6_h3_h6_b.cwnd (congestion window) and
app6_h3_h6_b.tp (throughput)" << endl;
      Ptr<OutputStreamWrapper> stream3CWND =
asciiTraceHelper.CreateFileStream("app6_h3_h6_b.cwnd");
      Ptr<OutputStreamWrapper> stream3PD =
asciiTraceHelper.CreateFileStream("app6_h3_h6_b.congestion_loss");
      Ptr<OutputStreamWrapper> stream3TP =
asciiTraceHelper.CreateFileStream("app6_h3_h6_b.tp");
      Ptr<OutputStreamWrapper> stream3GP =
asciiTraceHelper.CreateFileStream("app6_h3_h6_b.gp");
      Ptr<Socket> ns3TcpSocket3 =
uniFlow(InetSocketAddress(recieverIFCs.GetAddress(2), port), port,
"TcpWestwood", senders.Get(2), recievers.Get(2), otherFlowStart,
otherFlowStart+durationGap, packetSize, numPackets, transferSpeed,
otherFlowStart, otherFlowStart+durationGap);
```

```
      ns3TcpSocket3->TraceConnectWithoutContext("CongestionWindow",
MakeBoundCallback (&CwndChange, stream3CWND, 0));
      ns3TcpSocket3->TraceConnectWithoutContext("Drop", MakeBoundCallback
(&packetDrop, stream3PD, 0, 3));

      sink = "/NodeList/7/ApplicationList/0/$ns3::PacketSink/Rx";
      Config::Connect(sink, MakeBoundCallback(&ReceivedPacket, stream3GP, 0));

      sink_ = "/NodeList/7/ApplicationList/0/$ns3::PacketSink/Rx";
      Config::Connect(sink_, MakeBoundCallback(&ReceivedPacket, stream3TP,
otherFlowStart));



      //Turning on Static Global Routing
      Ipv4GlobalRoutingHelper::PopulateRoutingTables();

      Ptr<FlowMonitor> flowmon;
      FlowMonitorHelper flowmonHelper;
      flowmon = flowmonHelper.InstallAll();
      Simulator::Stop(Seconds(durationGap+otherFlowStart));

      //animation("anim4.xml", routers, senders, recievers);    //animation

      Simulator::Run();
      flowmon->CheckForLostPackets();

    cout << "Part b throughput into app6_b.tp" << endl;
      //Ptr<OutputStreamWrapper> streamTP =
asciiTraceHelper.CreateFileStream("app6_b.tp");
      Ptr<Ipv4FlowClassifier> classifier =
DynamicCast<Ipv4FlowClassifier>(flowmonHelper.GetClassifier());
      std::map<FlowId, FlowMonitor::FlowStats> stats = flowmon->GetFlowStats();
      for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i =
stats.begin(); i != stats.end(); ++i)
      {
            Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);
            if(t.sourceAddress == "10.1.0.1")
            {
                  if(mapDrop.find(1)==mapDrop.end())
                        mapDrop[1] = 0;

                  *stream1PD->GetStream() << "TcpReno Flow " << i->first  << "
(" << t.sourceAddress << " -> " << t.destinationAddress << ")\n";
                  *stream1PD->GetStream()  << "Net Packet Lost: " << i-
>second.lostPackets << "\n";
                  *stream1PD->GetStream()  << "Packet Lost due to buffer
overflow: " << mapDrop[1] << "\n";
                  *stream1PD->GetStream()  << "Packet Lost due to Congestion: "
<< i->second.lostPackets - mapDrop[1] << "\n";
                  *stream1PD->GetStream() << "Max throughput: " <<
mapMaxThroughput["/NodeList/5/$ns3::Ipv4L3Protocol/Rx"] << std::endl;
            }
            else if(t.sourceAddress == "10.1.1.1")
            {
                  if(mapDrop.find(2)==mapDrop.end())
                        mapDrop[2] = 0;

                  *stream2PD->GetStream() << "TcpTahoe Flow " << i->first  << "
(" << t.sourceAddress << " -> " << t.destinationAddress << ")\n";
                  *stream2PD->GetStream()  << "Net Packet Lost: " << i-
>second.lostPackets << "\n";
                  *stream2PD->GetStream()  << "Packet Lost due to buffer
overflow: " << mapDrop[2] << "\n";
```

```
                *stream2PD->GetStream()  << "Packet Lost due to Congestion: "
<< i->second.lostPackets - mapDrop[2] << "\n";
                *stream2PD->GetStream() << "Max throughput: " <<
mapMaxThroughput["/NodeList/6/$ns3::Ipv4L3Protocol/Rx"] << std::endl;
            }
            else if(t.sourceAddress == "10.1.2.1")
            {
                  if(mapDrop.find(3)==mapDrop.end())
                        mapDrop[3] = 0;

                  *stream3PD->GetStream() << "TcpWestWood Flow " << i->first  <<
" (" << t.sourceAddress << " -> " << t.destinationAddress << ")\n";
                  *stream3PD->GetStream()  << "Net Packet Lost: " << i-
>second.lostPackets << "\n";
                  *stream3PD->GetStream()  << "Packet Lost due to buffer
overflow: " << mapDrop[3] << "\n";
                  *stream3PD->GetStream()  << "Packet Lost due to Congestion: "
<< i->second.lostPackets - mapDrop[3] << "\n";
                  *stream3PD->GetStream() << "Max throughput: " <<
mapMaxThroughput["/NodeList/7/$ns3::Ipv4L3Protocol/Rx"] << std::endl;
            }
      }

      //flowmon->SerializeToXmlFile("app6_b.flowmon", true, true);
      Simulator::Destroy();

}



int main (int argc, char *argv[])
{
    part1();
    part2();

}
```