## Topic for Implementation:

Implementation of a new transport protocol using ns-3 simulator (used C++) running on Oracle VM in Linux Operating System and comparison of the new transport protocol with original TCP and UDP protocols in terms of throughput and number of dropped packets.

## Objective of the project:

In this project, I have tried to implement a delay-based Transmission Control Protocol (TCP) and compared it with the original TCP (TCP New Reno), TCP Vegas and UDP in terms of throughput and number of dropped packets.

## Introduction:

With the rapid growth of Internet population and the intensive usage of TCP/IP protocol suite, the transmission control protocol (TCP) congestion control algorithm has become a key factor influencing the performance and behavior of the Internet. Several studies have reported that delay-based TCP protocols provides better performance than loss-based ones with respect to overall network utilization, stability, fairness, throughput, packet loss, and burstiness. Since delay-based TCP uses the difference between the expected and actual flow rates to infer the congestion window adjustment policy from throughput measurements, it usually reduces the sending rate before the connection experiences a packet loss.

## TCP New One:

Our delay-based TCP protocol is known as **TCP New One**. It is loosely based on TCP Vegas. TCP Vegas adjusts its congestion window by following algorithm-

$$CWND = \begin{cases} CWND+1 & \text{if} \quad delta<\alpha \\ CWND & \text{if} \quad \alpha<delta<\beta \\ CWND-1 & \text{if} \quad \beta<delta \end{cases}$$

Where

- Delta = Expected_rate - Actual_rate
- Expected_rate = cwnd/base_rtt, cwnd is the current congestion window size and base_rtt is the minimum RTT of related connection
- Actual_rate = cwnd/rtt, rtt is actual roundtrip time

**TCP New One** adjusts its congestion window by following algorithm-

$$\text{CWND} = \begin{cases} \text{CWND} + [(0.4+\alpha\text{-delta})/\text{cwnd}] & \text{if} \quad \text{delta} < \alpha \\ \text{CWND} & \text{if} \quad \alpha < \text{delta} < \beta \\ \text{CWND}-1 & \text{if} \quad \text{delta} > \beta \end{cases}$$

α and β are parameters whose value is set at 4 and 7 respectively.

## Observations:

To measure the performance of our new transport protocol I have made use of the Flow Monitor module of ns-3. The statistics are collected for each flow and exported in XML format.

The following observations were made-

1. TCP New Reno without congestion


```
<Flow timesForwarded="10980" lostPackets="0" rxPackets="5490" txPackets="5740" rxBytes="3139244" txBytes="3282244"
    lastDelay="+4726159996.0ns" jitterSum="+135340986512.0ns" delaySum="+16131353612904.0ns"
    timeLastRxPacket="+99975117165.0ns" timeLastTxPacket="+99974717027.0ns" timeFirstRxPacket="+1008784000.0ns"
    timeFirstTxPacket="+1000000000.0ns" flowId="1"> </Flow>
```


2. TCP New Reno with congestion


```
<Flow timesForwarded="2260" lostPackets="984" rxPackets="1130" txPackets="2187" rxBytes="656764" txBytes="1263304"
lastDelay="+10912239200.0ns" jitterSum="+16712223428.0ns" delaySum="+6040922839651.0ns" timeLastRxPacket="+40000558357.0ns"
timeLastTxPacket="+96769915488.0ns" timeFirstRxPacket="+1008784000.0ns" timeFirstTxPacket="+1000000000.0ns" flowId="1">
```


3. UDP without congestion


```
<Flow timesForwarded="5666" lostPackets="0" rxPackets="2833" txPackets="2975" rxBytes="3025644" txBytes="3177300"
    lastDelay="+4725423996.0ns" jitterSum="+7191327983.0ns" delaySum="+8514206308668.0ns" timeLastRxPacket="+99974381164.0ns"
    timeLastTxPacket="+99974717026.0ns" timeFirstRxPacket="+1057359997.0ns" timeFirstTxPacket="+1000000000.0ns" flowId="2"> </Flow>
```


4. UDP with Congestion


```
<Flow timesForwarded="3096" lostPackets="5101" rxPackets="1548" txPackets="7140" rxBytes="1653264"
txBytes="7625520" lastDelay="+16248538116.0ns" jitterSum="+16191178119.0ns"
delaySum="+13053652628516.0ns" timeLastRxPacket="+39989870356.0ns" timeLastTxPacket="+99994128574.0ns"
timeFirstRxPacket="+1057359997.0ns" timeFirstTxPacket="+1000000000.0ns" flowId="2">
```


5. TCP Vegas without congestion


```
<Flow timesForwarded="10184" lostPackets="0" rxPackets="5092" txPackets="5097" rxBytes="2992612" txBytes="2995552"
    lastDelay="+125967999.0ns" jitterSum="+82120061945.0ns" delaySum="+475869311722.0ns" timeLastRxPacket="+99979341027.0ns"
    timeLastTxPacket="+99978493024.0ns" timeFirstRxPacket="+1008784000.0ns" timeFirstTxPacket="+1000000000.0ns" flowId="1"> </Flow>
```

6. TCP Vegas with congestion

<Flow timesForwarded="612" lostPackets="0" rxPackets="306" txPackets="312" rxBytes="178860" txBytes="182388" lastDelay="+2912527824.0ns" jitterSum="+4124864166.0ns" delaySum="+449605318678.0ns" timeLastRxPacket="+98234378488.0ns" timeLastTxPacket="+98242970485.0ns" timeFirstRxPacket="+1008784000.0ns" timeFirstTxPacket="+1000000000.0ns" flowId="1"> </Flow>

7. TCP New One without congestion

<Flow timesForwarded="10184" lostPackets="0" rxPackets="5092" txPackets="5098" rxBytes="2992612" txBytes="2996140" lastDelay="+125967999.0ns" jitterSum="+91187261465.0ns" delaySum="+592451868267.0ns" timeLastRxPacket="+99979341027.0ns" timeLastTxPacket="+99978493024.0ns" timeFirstRxPacket="+1008784000.0ns" timeFirstTxPacket="+1000000000.0ns" flowId="1"> </Flow>

8. TCP New One with congestion

<Flow timesForwarded="676" lostPackets="0" rxPackets="338" txPackets="344" rxBytes="197676" txBytes="201204" lastDelay="+3203567807.0ns" jitterSum="+4274784321.0ns" delaySum="+547375168817.0ns" timeLastRxPacket="+99563658428.0ns" timeLastTxPacket="+99772250425.0ns" timeFirstRxPacket="+1008784000.0ns" timeFirstTxPacket="+1000000000.0ns" flowId="1"> </Flow>

## Results:

$$Throughput = 8.txBytes/(timeLastRxPacket - TimeFirstRxPacket)$$

| Transport Layer Protocol | Throughput (kbps) | Number of dropped Packets |
|---|---|---|
| TCP New Reno without congestion | 253.98 | 0 |
| TCP New Reno with congestion | 131.35 | 984 |
| UDP without congestion | 244.67 | 0 |
| UDP with congestion | 244.67 | 5101 |
| TCP Vegas without congestion | 242.07 | 0 |
| TCP Vegas with congestion | 14.31 | 0 |
| TCP New One without congestion | 242.07 | 0 |
| TCP New One with congestion | 16.05 | 0 |

## Inference:

From the results, I can infer that the number of packets dropped by our proposed TCP protocol, **TCP New One** are 0 because it is delay based and not loss based. Also, comparing the throughput, **TCP New One** matches TCP Vegas in no congestion scenario and betters it when there is congestion.

The reason why the throughput for **TCP New One** drops significantly during congestion phase is because, when it is sharing the link with other connections this connection fails to get a fair share of link bandwidth. This happens because loss-based TCP protocol try to utilize the available bandwidth until facing a packet loss. Whereas **TCP New One** will cautiously increase its sending rate to prevent congestion and consequently will fail at competing with loss-based TCP connection. This gives rise to a fairness problem and it remains a work in progress.

## Appendix:

1. ***Code for the Network Topology***

```
// Network topology
//
//       n0 ---+      +--- n2
//             |      |
//             n4 -- n5
//             |      |
//       n1 ---+      +--- n3
//
// - All links are P2P with 500kb/s and 2ms
// - TCP flow form n0 to n2
// - UDP flow from n1 to n3

#include <fstream>
#include <string>
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/netanim-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/tcp-new-own.h"


using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("Lab2");

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;
}

static void
CwndChange (uint32_t oldCwnd, uint32_t newCwnd)
{
  std::cout << Simulator::Now ().GetSeconds () << "\t" << newCwnd <<"\n";
}

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


int main (int argc, char *argv[])
{
  std::string lat = "2ms";
  std::string rate = "500kb/s"; // P2P link
  bool enableFlowMonitor = false;


  CommandLine cmd;
  cmd.AddValue ("latency", "P2P link Latency in miliseconds", lat);
  cmd.AddValue ("rate", "P2P data rate in bps", rate);
  cmd.AddValue ("EnableMonitor", "Enable Flow Monitor", enableFlowMonitor);

  cmd.Parse (argc, argv);
Config::SetDefault("ns3::TcpL4Protocol::SocketType",TypeIdValue(TypeId::LookupByName("ns3::Tc
pNewOwn")));
//
// Explicitly create the nodes required by the topology (shown above).
//
  NS_LOG_INFO ("Create nodes.");
  NodeContainer c; // ALL Nodes
  c.Create(6);

  NodeContainer n0n4 = NodeContainer (c.Get (0), c.Get (4));
  NodeContainer n1n4 = NodeContainer (c.Get (1), c.Get (4));
  NodeContainer n2n5 = NodeContainer (c.Get (2), c.Get (5));
  NodeContainer n3n5 = NodeContainer (c.Get (3), c.Get (5));
  NodeContainer n4n5 = NodeContainer (c.Get (4), c.Get (5));

//
// Install Internet Stack
//
```

```
  InternetStackHelper internet;
  internet.Install (c);

  // We create the channels first without any IP addressing information
  NS_LOG_INFO ("Create channels.");
  PointToPointHelper p2p;
  p2p.SetDeviceAttribute ("DataRate", StringValue (rate));
  p2p.SetChannelAttribute ("Delay", StringValue (lat));
  NetDeviceContainer d0d4 = p2p.Install (n0n4);
  NetDeviceContainer d1d4 = p2p.Install (n1n4);
  NetDeviceContainer d4d5 = p2p.Install (n4n5);
  NetDeviceContainer d2d5 = p2p.Install (n2n5);
  NetDeviceContainer d3d5 = p2p.Install (n3n5);

    // Later, we add IP addresses.
  NS_LOG_INFO ("Assign IP Addresses.");
  Ipv4AddressHelper ipv4;
  ipv4.SetBase ("10.1.1.0", "255.255.255.0");
  Ipv4InterfaceContainer i0i4 = ipv4.Assign (d0d4);

  ipv4.SetBase ("10.1.2.0", "255.255.255.0");
  Ipv4InterfaceContainer i1i4 = ipv4.Assign (d1d4);

  ipv4.SetBase ("10.1.3.0", "255.255.255.0");
  Ipv4InterfaceContainer i4i5 = ipv4.Assign (d4d5);

  ipv4.SetBase ("10.1.4.0", "255.255.255.0");
  Ipv4InterfaceContainer i2i5 = ipv4.Assign (d2d5);

  ipv4.SetBase ("10.1.5.0", "255.255.255.0");
  Ipv4InterfaceContainer i3i5 = ipv4.Assign (d3d5);

  NS_LOG_INFO ("Enable static global routing.");
  //
  // Turn on global static routing so we can actually be routed across the network.
  //
  Ipv4GlobalRoutingHelper::PopulateRoutingTables ();


  NS_LOG_INFO ("Create Applications.");

  // TCP connfection from N0 to N2

  uint16_t sinkPort = 8080;
  Address sinkAddress (InetSocketAddress (i2i5.GetAddress (0), sinkPort)); // interface of n2
  PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory", InetSocketAddress
(Ipv4Address::GetAny (), sinkPort));
  ApplicationContainer sinkApps = packetSinkHelper.Install (c.Get (2)); //n2 as sink
  sinkApps.Start (Seconds (0.));
  sinkApps.Stop (Seconds (100.));

  Ptr<Socket> ns3TcpSocket = Socket::CreateSocket (c.Get (0), TcpSocketFactory::GetTypeId
()); //source at n0

  // Trace Congestion window
  ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow", MakeCallback (&CwndChange));

  // Create TCP application at n0
  Ptr<MyApp> app = CreateObject<MyApp> ();
  app->Setup (ns3TcpSocket, sinkAddress, 1040, 100000, DataRate ("250Kbps"));
  c.Get (0)->AddApplication (app);
  app->SetStartTime (Seconds (1.));
  app->SetStopTime (Seconds (100.));
```

```
  // UDP connfection from N1 to N3

  uint16_t sinkPort2 = 6;
  Address sinkAddress2 (InetSocketAddress (i3i5.GetAddress (0), sinkPort2)); // interface of
n3
  PacketSinkHelper packetSinkHelper2 ("ns3::UdpSocketFactory", InetSocketAddress
(Ipv4Address::GetAny (), sinkPort2));
  ApplicationContainer sinkApps2 = packetSinkHelper2.Install (c.Get (3)); //n3 as sink
  sinkApps2.Start (Seconds (0.));
  sinkApps2.Stop (Seconds (100.));

  Ptr<Socket> ns3UdpSocket = Socket::CreateSocket (c.Get (1), UdpSocketFactory::GetTypeId
()); //source at n1

  // Create UDP application at n1
  Ptr<MyApp> app2 = CreateObject<MyApp> ();
  app2->Setup (ns3UdpSocket, sinkAddress2, 1040, 100000, DataRate ("250Kbps"));
  c.Get (1)->AddApplication (app2);
  app2->SetStartTime (Seconds (20.));
  app2->SetStopTime (Seconds (100.));

// Increase UDP Rate
  Simulator::Schedule (Seconds(30.0), &IncRate, app2, DataRate("500kbps"));


AnimationInterface anim ("l-2-netanim.xml");

  anim.SetConstantPosition (c.Get(0),0.07,24.6);
  anim.SetConstantPosition (c.Get(1),0.223,73.35);
  anim.SetConstantPosition (c.Get(2),73.62,24.40);
  anim.SetConstantPosition (c.Get(3),73.030,73.328);

  anim.SetConstantPosition (c.Get(4),24.40,49.0);
  anim.SetConstantPosition (c.Get(5),49.152,48.93);


Ptr<FlowMonitor> flowmon;
FlowMonitorHelper flowHelper;
flowmon = flowHelper.InstallAll();

//
// Now, do the actual simulation.
//
 Simulator::Stop (Seconds(100));
  Simulator::Run ();
flowmon->CheckForLostPackets ();
flowmon->SerializeToXmlFile("l-2_own_noconges2.xml", false, true);


  NS_LOG_INFO ("Run Simulation.");
  Simulator::Destroy ();
  NS_LOG_INFO ("Done.");
}
```

### 2. *Code for New One TCP protocol*

```
#include "tcp-new-one.h"
#include "ns3/tcp-socket-base.h"
#include "ns3/log.h"

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("TcpNewOne");
NS_OBJECT_ENSURE_REGISTERED (TcpNewOne);

TypeId
TcpNewOne::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::TcpNewOne")
    .SetParent<TcpNewReno> ()
    .AddConstructor<TcpNewOne> ()
    .SetGroupName ("Internet")
    .AddAttribute ("Alpha", "Lower bound of packets in network",
                   UintegerValue (4),
                   MakeUintegerAccessor (&TcpNewOne::m_alpha),
                   MakeUintegerChecker<uint32_t> ())
    .AddAttribute ("Beta", "Upper bound of packets in network",
                   UintegerValue (7),
                   MakeUintegerAccessor (&TcpNewOne::m_beta),
                   MakeUintegerChecker<uint32_t> ())
    .AddAttribute ("Gamma", "Limit on increase",
                   UintegerValue (1),
                   MakeUintegerAccessor (&TcpNewOne::m_gamma),
                   MakeUintegerChecker<uint32_t> ())
  ;
  return tid;
}

TcpNewOne::TcpNewOne (void)
  : TcpNewReno (),
    m_alpha (4),
    m_beta (7),
    m_gamma (1),
    m_baseRtt (Time::Max ()),
    m_minRtt (Time::Max ()),
    m_cntRtt (0),
    m_doingNewOneNow (true),
    m_begSndNxt (0)
{
  NS_LOG_FUNCTION (this);
}

TcpNewOne::TcpNewOne (const TcpNewOne& sock)
  : TcpNewReno (sock),
    m_alpha (sock.m_alpha),
    m_beta (sock.m_beta),
    m_gamma (sock.m_gamma),
    m_baseRtt (sock.m_baseRtt),
    m_minRtt (sock.m_minRtt),
    m_cntRtt (sock.m_cntRtt),
    m_doingNewOneNow (true),
    m_begSndNxt (0)
{
  NS_LOG_FUNCTION (this);
}

TcpNewOne::~TcpNewOne (void)
{
  NS_LOG_FUNCTION (this);
}

Ptr<TcpCongestionOps>
TcpNewOne::Fork (void)
{
  return CopyObject<TcpNewOne> (this);
}
```

```
void
TcpNewOne::PktsAcked (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked,
                      const Time& rtt)
{
  NS_LOG_FUNCTION (this << tcb << segmentsAcked << rtt);

  if (rtt.IsZero ())
    {
      return;
    }

  m_minRtt = std::min (m_minRtt, rtt);
  NS_LOG_DEBUG ("Updated m_minRtt = " << m_minRtt);

  m_baseRtt = std::min (m_baseRtt, rtt);
  NS_LOG_DEBUG ("Updated m_baseRtt = " << m_baseRtt);

  // Update RTT counter
  m_cntRtt++;
  NS_LOG_DEBUG ("Updated m_cntRtt = " << m_cntRtt);
}

void
TcpNewOne::EnableNewOne (Ptr<TcpSocketState> tcb)
{
  NS_LOG_FUNCTION (this << tcb);

  m_doingNewOneNow = true;
  m_begSndNxt = tcb->m_nextTxSequence;
  m_cntRtt = 0;
  m_minRtt = Time::Max ();
}

void
TcpNewOne::DisableNewOne ()
{
  NS_LOG_FUNCTION (this);

  m_doingNewOneNow = false;
}

void
TcpNewOne::CongestionStateSet (Ptr<TcpSocketState> tcb,
                               const TcpSocketState::TcpCongState_t newState)
{
  NS_LOG_FUNCTION (this << tcb << newState);
  if (newState == TcpSocketState::CA_OPEN)
    {
      EnableNewOne (tcb);
    }
  else
    {
      DisableNewOne ();
    }
}

void
TcpNewOne::IncreaseWindow (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked)
{
  NS_LOG_FUNCTION (this << tcb << segmentsAcked);

  if (!m_doingNewOneNow)
    {
      // If NewOne is not on, we follow NewReno algorithm
      NS_LOG_LOGIC ("NewOne is not turned on, we follow NewReno algorithm.");
      TcpNewReno::IncreaseWindow (tcb, segmentsAcked);
      return;
    }

  if (tcb->m_lastAckedSeq >= m_begSndNxt)
```

```
{ // A NewOne cycle has finished, we do NewOne cwnd adjustment every RTT.

  NS_LOG_LOGIC ("A NewOne cycle has finished, we adjust cwnd once per RTT.");

  // Save the current right edge for next NewOne cycle
  m_begSndNxt = tcb->m_nextTxSequence;

  /*
   * We perform NewOne calculations only if we got enough RTT samples to
   * insure that at least 1 of those samples wasn't from a delayed ACK.
   */
  if (m_cntRtt <= 2)
    {  // We do not have enough RTT samples, so we should behave like Reno
      NS_LOG_LOGIC ("We do not have enough RTT samples to do NewOne, so we behave like NewReno.");
      TcpNewReno::IncreaseWindow (tcb, segmentsAcked);
    }
  else
    {
      NS_LOG_LOGIC ("We have enough RTT samples to perform NewOne calculations");
      /*
       * We have enough RTT samples to perform NewOne algorithm.
       * Now we need to determine if cwnd should be increased or decreased
       * based on the calculated difference between the expected rate and actual sending
       * rate and the predefined thresholds (alpha, beta, and gamma).
       */
      uint32_t diff;
      uint32_t targetCwnd;
      uint32_t segCwnd = tcb->GetCwndInSegments ();

      /*
       * Calculate the cwnd we should have. baseRtt is the minimum RTT
       * per-connection, minRtt is the minimum RTT in this window
       *
       * little trick:
       * desidered throughput is currentCwnd * baseRtt
       * target cwnd is throughput / minRtt
       */
      double tmp = m_baseRtt.GetSeconds () / m_minRtt.GetSeconds ();
      targetCwnd = segCwnd * tmp;
      NS_LOG_DEBUG ("Calculated targetCwnd = " << targetCwnd);
      NS_ASSERT (segCwnd >= targetCwnd); // implies baseRtt <= minRtt

      /*
       * Calculate the difference between the expected cWnd and
       * the actual cWnd
       */
      diff = segCwnd - targetCwnd;
      NS_LOG_DEBUG ("Calculated diff = " << diff);

      if (diff > m_gamma && (tcb->m_cWnd < tcb->m_ssThresh))
        {
          /*
           * We are going too fast. We need to slow down and change from
           * slow-start to linear increase/decrease mode by setting cwnd
           * to target cwnd. We add 1 because of the integer truncation.
           */
          NS_LOG_LOGIC ("We are going too fast. We need to slow down and "
                        "change to linear increase/decrease mode.");
          segCwnd = std::min (segCwnd, targetCwnd + 1);
          tcb->m_cWnd = segCwnd * tcb->m_segmentSize;
          tcb->m_ssThresh = GetSsThresh (tcb, 0);
          NS_LOG_DEBUG ("Updated cwnd = " << tcb->m_cWnd <<
                        " ssthresh=" << tcb->m_ssThresh);
        }
      else if (tcb->m_cWnd < tcb->m_ssThresh)
        {     // Slow start mode
          NS_LOG_LOGIC ("We are in slow start and diff < m_gamma, so we "
                        "follow NewReno slow start");
          TcpNewReno::SlowStart (tcb, segmentsAcked);
        }
      else
```

```
            {     // Linear increase/decrease mode
              NS_LOG_LOGIC ("We are in linear increase/decrease mode");
              if (diff > m_beta)
                {
                  // We are going too fast, so we slow down
                  NS_LOG_LOGIC ("We are going too fast, so we slow down by decrementing cwnd");
                  segCwnd--;
                  tcb->m_cWnd = segCwnd * tcb->m_segmentSize;
                  tcb->m_ssThresh = GetSsThresh (tcb, 0);
                  NS_LOG_DEBUG ("Updated cwnd = " << tcb->m_cWnd <<
                                " ssthresh=" << tcb->m_ssThresh);
                }
              else if (diff < m_alpha)
                {
                  // We are going too slow (having too little data in the network),
                  // so we speed up.
                  NS_LOG_LOGIC ("We are going too slow, so we speed up by incrementing cwnd");
                  segCwnd+=((0.4+m_alpha-diff)/segCwnd);
                  tcb->m_cWnd = segCwnd * tcb->m_segmentSize;
                  NS_LOG_DEBUG ("Updated cwnd = " << tcb->m_cWnd <<
                                " ssthresh=" << tcb->m_ssThresh);
                }
              else
                {
                  // We are going at the right speed
                  NS_LOG_LOGIC ("We are sending at the right speed");
                }
            }
          tcb->m_ssThresh = std::max (tcb->m_ssThresh, 3 * tcb->m_cWnd / 4);
          NS_LOG_DEBUG ("Updated ssThresh = " << tcb->m_ssThresh);
        }

      // Reset cntRtt & minRtt every RTT
      m_cntRtt = 0;
      m_minRtt = Time::Max ();
    }
  else if (tcb->m_cWnd < tcb->m_ssThresh)
    {
      TcpNewReno::SlowStart (tcb, segmentsAcked);
    }
}

std::string
TcpNewOne::GetName () const
{
  return "TcpNewOne";
}

uint32_t
TcpNewOne::GetSsThresh (Ptr<const TcpSocketState> tcb,
                        uint32_t bytesInFlight)
{
  NS_LOG_FUNCTION (this << tcb << bytesInFlight);
  return std::max (std::min (tcb->m_ssThresh.Get (), tcb->m_cWnd.Get () - tcb->m_segmentSize), 2 * tcb-
>m_segmentSize);
}

} // namespace ns3
```

## References:


1. https://www.nsnam.org/docs/models/html/flow-monitor.html
2. https://www.nsnam.org/docs/tutorial/html/

3. L. Brakmo and L. Peterson, “TCP Vegas: End-to-end congestion avoidance on a global Internet,” *IEEE J. Select. Areas Commun.*, vol. 13, no. 8, pp. 1465–1480, Oct. 1995.